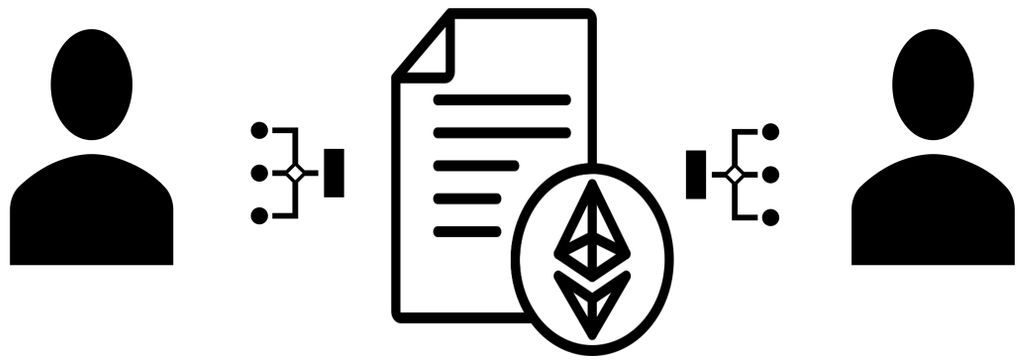
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

LICENTIATE THESIS

Smart contracts are programs deployed on a blockchain platform that automatically execute stated rules when specified conditions are met. They are used to manage distribution of digital assets and to automate transactions without relying on a central intermediary party. Smart contracts are used across various industries such as finance, supply chain, and healthcare, to automate agreements and transactions. Once a smart contract is deployed it cannot be modified, so that if a bug or a vulnerability is discovered post deployment this cannot be corrected, which exposes the users of the contract to unmitigable risks. Thus, ensuring correct behavior of smart contracts is of the utmost importance. The thesis presents an approach to model smart contracts as discrete event systems. The principles for this conversion are outlined and it is shown how liveness properties can be verified. One application for this is detecting whether a smart contract is vulnerable to malicious attacks that can prevent the contract from releasing its assets of value. In addition, the thesis proposes a framework for detecting bias within smart contracts. Using supervisory control theory, we are able to detect strategies that might provide unfair advantage to some users or place specific users at a disadvantage.

CHALMERS
UNIVERSITY OF TECHNOLOGY

NISHANT PAREKH • Behavioral Analysis of Smart Contracts Using Supervisory Control Theory • 2026



Behavioral Analysis of Smart Contracts Using Supervisory Control Theory

NISHANT PAREKH

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Behavioral Analysis of Smart Contracts Using Supervisory Control Theory

Nishant Parekh

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2026

Behavioral Analysis of Smart Contracts Using Supervisory Control Theory

Nishant Parekh

Acknowledgements, dedications, and similar personal statements in this thesis, reflect the author's own views.

Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
Phone: +46 (0)31 772 1000

Cover:
The cover image shows users exchanging information or performing transactions with a smart contract deployed on the Ethereum blockchain.

Note: Large language models were used solely as assistive tools for grammar checking and language refinement, while the content and analysis in the thesis remain the author's own work.

Printed by Chalmers Digital Printing
Gothenburg, Sweden, August 2026

Behavioral Analysis of Smart Contracts Using Supervisory Control Theory

Nishant Parekh
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Smart contracts are programs deployed on blockchains that are typically used to manage distribution of digital assets and cryptocurrencies. Smart contracts are used in applications such as auctions, games, financial services and decentralized applications. It is not possible to change a smart contract after it has been deployed on the blockchain network. Since smart contracts can hold and manage transfer of digital assets, even a small error in their behavior can lead to serious consequences. This thesis explores how smart contracts can be modeled as event driven systems and be analyzed for certain properties.

We present a framework for automatically converting Solidity smart contracts into Extended Finite State Machine models that describe the execution of the smart contract. Using these models, we show how non-blocking verification can detect situations where a contract may fail to reach its intended completion. In particular, we study cases where failed transfers may result in funds being locked or prevent the expected progress of the contract.

Furthermore, this thesis investigates *bias* in smart contracts. *Bias* here refers to an advantage/disadvantage allowed by the behavior of the contract to a user, or a group of users. Such an advantage/disadvantage reveals the ability to force an outcome that either benefits the user(s) or harms others. By analyzing the contract from perspectives of different users, we show how such a *bias* can be detected.

Keywords: Smart Contracts, Behavioral Analysis, Formal Verification, Supervisory Control Theory, Non-Blocking, Extended Finite State Machines

To the world

List of Publications

This thesis is based on the following publications:

[A] **Nishant Parekh**, Wolfgang Ahrendt, Martin Fabian, "Smart Contract Denial-of-Service Analysis Using Non-Blocking Verification". Published in Discrete Event Dynamic Systems, Dec 2025.

[B] Martin Fabian, **Nishant Parekh**, Wolfgang Ahrendt , "On Bias in User-Centric Discrete-Event Systems". *Proc. 18th Workshop on Discrete Event Systems, Eindhoven, Netherlands, May, 2026.*

[C] **Nishant Parekh**, Wolfgang Ahrendt, Martin Fabian, "On Detecting Bias in Smart Contracts Using Supervisor Synthesis". *Proc. 18th Workshop on Discrete Event Systems, Eindhoven, Netherlands, May, 2026.*

Other publications by the author, not included in this thesis, are:

[D] **Nishant Parekh**, Wolfgang Ahrendt, Martin Fabian, "Automatic Conversion of Smart Contracts for Non-Blocking Verification". *Proc. 17th Workshop on Discrete Event Systems, May, 2024, Rio de Janeiro, Brazil.*

Contents

Acknowledgments

A thesis may carry one name on the cover, but it is never the work of one person alone. It is made possible by conversations, encouragement, criticism, shared meals, support, laughter in tough times, and the many people who make the journey worth taking.

I have been fortunate to have Martin and Wolfgang as my supervisors. Your guidance has shaped much more than this thesis. You have taught me to approach difficult questions patiently and examine ideas critically. Thank you for your time, guidance, trust, encouragement, and for always challenging me to improve.

The years behind this thesis have also been filled with friendships that made life richer than research alone ever could. Sabino and Ying, sharing an office with you turned ordinary workdays into something much better and way more fun. Alvin, our many discussions and brainstorming sessions have always been enjoyable and an insightful part of this journey.

I am deeply grateful as well to Ola, Erik, Chiara, Adam, Sondre, Lasse, Daniel, Nanami, Zhitao, Gabriel, Bethlehem, Wenhao, Kilian, Godwin, Ahmet, Francesco, Martina, Mattia, Kristian, Sopan, Aniruddh, Yash, Parag, Prayag, and Parul for the conversations, laughter, encouragement, and the memories we have shared. To everyone whose names I may have unintentionally missed, please know that your presence and support have been deeply appreciated and have made these years more meaningful.

My deepest gratitude goes to my family. To my father, Ajay, and my mom, Saroj, thank you for your love, support, steady presence and the encouragement you have given me throughout this journey. To my elder brother, Raunak, who has stood like a rock throughout my life and this PhD journey. To Disha, my wife and my closest companion. Your patience, love, and unwavering faith in me have been a source of strength far beyond what words can fully express.

Each of you has contributed, in your own beautiful way, to the person I have become along the way. I will always look back on this journey with gratitude, not only for what I learned, but for the people with whom I was fortunate enough to share it.

Acronyms

DES:	Discrete Event Systems
UCDES:	User Centric Discrete Event Systems
SCT:	Supervisory Control Theory
FSM:	Finite State Machine
EFSM:	Extended Finite State Machine
AST:	Abstract Syntax Tree

Part I

Overview

CHAPTER 1

Introduction

Digital systems often require several parties to record, verify and update shared information. In traditional systems, this coordination is usually performed by a trusted intermediary, such as a bank or a central authority. The intermediary is responsible for maintaining the liveness of the system, verification of flowing information and settling disputes amongst participants. However, the dependence on such an intermediary also creates a single point of control, and all involved in the system must have faith that the intermediary will act properly.

This reliance introduces several challenges. Not only does the intermediary become a potential bottleneck, increasing operational costs due to resources required to manage and secure the system, it also introduces a single point of failure where technical faults, malicious attacks, or internal misconduct can disrupt the way system operates. Lastly, the participants of the system must place implicit trust in the intermediary's integrity even though they may have limited visibility into its internal process.

In response to these limitations, alternative approaches have emerged that aim to distribute trust across the system rather than to centralize it. These systems rely on decentralized protocols and consensus mechanisms that al-

low multiple participants to independently verify and agree upon the state of shared data. By reducing reliance on a central authority, such approaches can improve robustness and enable more transparent coordination among participants [1]. Blockchain systems provide a way for multiple parties to maintain a shared record of transactions in a decentralized manner [2, 3], without relying on a central intermediary. While early blockchain systems mainly support the transfer of digital currency, later platforms such as Ethereum [4] extend this model by enabling executable code on the blockchain in the form of smart contracts. Smart contracts make it possible to automate agreements, asset transfers, and other application logic in a decentralized and trustless setting.

Applications built using smart contracts, known as decentralized applications [5] (DApps), support a wide range of applications such as token systems, decentralized finance [6] (DeFi), digital marketplaces, governance mechanisms, decentralized autonomous organizations [7] (DAOs) and other forms of multi-party coordination. Smart contracts operate as the direct execution layer for exchanges, protocols, asset issuance, game rules and collective decision making. The Organization for Economic Co-operation and Development (OECD) observes strong interest among market participants and policy makers in distributed-ledger-based financial applications [8]. Future predictions suggest substantial growth potential. For instance, McKinsey estimates that tokenized financial assets may reach around USD 2 trillion by 2030 in a base-case scenario [9], indicating the growing economic relevance of programmable blockchain infrastructure in which smart contracts serve as a key mechanism for automation and execution.

Smart contracts can enforce agreements among mutually distrusting parties without relying on a trusted intermediary. Although the idea predates blockchains and goes back to Nick Szabo's early work in the 1990s [10], Ethereum was the first major platform to realize smart contracts as a general purpose programmable infrastructure for decentralized applications [11]. Smart contracts are an important foundation for decentralized systems, as they allow rules of interaction to be expressed directly in code and executed in a transparent and verifiable manner across a distributed network.

A key characteristic of smart contracts is that they often directly govern digital assets. They may determine who is allowed to transfer funds, who can participate in a protocol, when an auction can close, or how collective decisions are made in decentralized organizations. Another important feature

of smart contracts is that they execute in an open and possible an adversarial environment [12]. Multiple users can interact with the same contract and often with distinct goals. In this setting, correctness cannot be reduced to the absence of programming bugs alone. A smart contract may be free from bugs but still show undesirable behavior. An undesired behavior may include a contract that is unfair to some participants, or a contract that allows executions where the expected progress does not happen, or where an attacker is able to make the contract behave in ways unintended by a regular user.

The growing adoption of DApps has made security and correctness increasingly critical, especially as these systems manage large amounts of funds. To deploy a DApp, its smart contracts must be published to a blockchain network, at which point the Dapp goes "live". Once a smart contracts has been deployed on the blockchain, it cannot be modified, which makes verifying its correctness before deployment essential. Despite improving safety standards and more rigorous security audits, attacks on smart contracts are common. Chainalysis reported $2.2 billion stolen in 2024 [13] and more than $3.4 billion stolen in 2025 [14]. To prevent such losses, stronger assurances other than conventional testing are needed. In this context, formal verification [15] is particularly relevant, as it establishes whether a system satisfies a formal specification over all possible executions.

Many important smart contract concerns are not limited to classical code defects. Existing verification tools and approaches focus primarily on low-level correctness conditions, including arithmetic errors, division by zero, out-of bounds accesss, and insufficient funds [16]. This motivates the need for modeling frameworks that can capture both the internal logic of a contract and the effects of interaction with multiple external actors.

Discrete event systems (DES) [17] form a broad class of systems whose behavior evolves through discrete occurrences rather than continuously over time. The system changes its state in relation to an event, and the resulting evolution is determined by the order in which event occur. This makes the DES framework well suited for many kinds of reactive systems, including software systems, where possible sequences of events and system behaviors are to be analyzed. Modeling a system as DES makes it possible to analyze how system behavior emerges from interaction of events and enables us to investigate where certain states of the system can be reached or avoided.

Extended finite state machines [18, 19] (EFSM) provide a convenient way to

model systems within the discrete event systems framework, especially when the system progression is data dependent. Evolution of a system may depend on certain conditions and might also update certain attributes associated with the system. EFSMs allow for a compact representation of such systems as they can capture system behavior through *guards*, which act as preconditions for a system to progress from a specific state to another and *actions*, which update values associated with the state change. For this reason, EFSMs are an appropriate modeling formalism to analyze smart contract behavior. Smart contracts are driven by events, such as a participant calling a certain function, but the effect of those events often depends on contract state, stored values, and conditions on incoming interactions.

Supervisory control theory [20] (SCT) develops this general DES perspective further by studying how system behavior can be examined with respect to a desired specification. While DES provides the formalism to model system behavior, that is what the system can do, SCT provides tools to answer questions about whether the system can be made to behave in a way that satisfies intended requirements.

In particular, it is useful to view smart contracts as an event-driven system, where contract execution steps can be represented as transitions [21]. Such a perspective makes it possible to not only study whether a contract can reach an erroneous state, but also to examine contract behavior against its expectations. This behavioral viewpoint forms the basis of this thesis, investigating smart contracts through the lens of discrete event systems. It should be noted that *state transition* in smart contracts and *transitions* in a DES model are not identical. A state transition with respect to smart contract is the process of updating the blockchain's current state, such as account balances or contract variables. Whereas, in a DES model, a transition is the process by which the system moves from one state to another.

1.1 Behavioral Analysis of Smart Contracts

The behavior of a smart contract is determined by the sequence of interactions, allowed between users and the contract. Each function call may enable new actions, disable other, update variables and transfer digital assets. As a result, the correctness of a smart contract also depends on how the users interact with the contract over time, and whether the resulting executions lead to

intended outcomes. This thesis focuses on two behavioral aspects: bias in smart contract behavior, and the effect of malicious users on contract progress.

Analysis based on Bias

The first concern, bias in smart contract behavior, looks into whether the contract gives some user, or group of users, an advantage in reaching a desired outcome. Interpreting the behavior of smart contracts in terms of bias is useful because it separates the contract design from the intentions of the participants. A biased behavior does not necessarily mean that the contract contains an explicit malicious condition. Rather it means that the structure of the contract permits the user(s) to achieve a desired objective regardless of the choices made by other participants. When it comes to the behavior of the contract, bias is not limited to user(s) achieving their desired goals. Bias could also emerge where users are able to prevent other users from achieving their goals or putting them in a detrimental position. The same contract can therefore be analyzed from different user perspectives. Bias, is analyzed with respect to a chosen user along with their chosen objective. For instance, in a game-like contract, the objective may be to reach a winning state.

Identifying bias is important because smart contracts often manage valuable digital assets. Their behavior is determined by their code, and users may intentionally search for strategies that allow them to influence or control the outcome of an interaction. A contract may appear fair when examined at the level of its code, but still allow the user(s) to force a favorable result.

Analysis under Malicious User

The second form of behavioral analysis studies what happens when a user acts maliciously, when interacting with smart contracts. In a trustless environment, a contract cannot assume that all participants will cooperate or act with benevolent intent. A user may refuse to perform an expected action, repeatedly call enabled functions, exploit timing or state dependencies, or interact with the contract through another contract designed to not cooperate under certain conditions. Therefore analysis must consider adversarial behaviors that are still permitted by the contract.

The issue investigated in the thesis is one of progress. A contract is designed to move toward some intended outcome, such as completing an interaction or

distributing digital assets. If a malicious user can lead it into a state from which the intended outcome can no longer be reached, then the contract is vulnerable. The problem is not only that one action fails, such as a transfer, but that the failure prevents further progress. This kind of analysis shows that a contract may be correct under the assumption that participants are cooperative, while still being problematic under adversarial interaction.

1.2 Research Questions

The research presented in this thesis is guided by three research questions. These questions arise from identified gaps in the existing literature on smart contract verification. In particular, this thesis builds on the view that smart contracts can be modeled as discrete event systems and analyzed using supervisory control theory. This perspective makes it possible to study behavioral properties that are difficult to capture through conventional program analysis alone.

The first research question concerns how smart contracts can be modeled as DES. In order to apply methods from supervisory control theory, smart contracts must first be represented by a formal model that captures control flow, state changes, guards, and interactions between different participants. Although verification approaches for smart contracts exist, the literature offers limited support for a systematic translation from smart contract code to behavioral models. This creates a methodological gap between smart contracts and formal verification techniques based on finite state modeling and control. The first research question therefore addresses this issue:

RQ1: *How can smart contracts be automatically converted to extended finite state machines?*

To answer **RQ1**, presented in this thesis is a framework for converting smart contracts to extended finite state machines by abstracting their source code into EFSM components. At a high level, the conversion identifies the parts of the contract that determine its possible executions and organizes these parts into an EFSM structure. The contract's interactions, internal executions, and user actions are represented in a way that preserves the logic of the smart contracts. The automatic conversion process is presented in Chapter 4 and

Paper A.

The next question focuses on progress-related correctness properties of smart contracts. In blockchain applications, correctness is often discussed in terms of security vulnerabilities, access control, or functional correctness [22–24]. However, smart contracts may also exhibit failures related to progress, where funds or contract execution can become stuck. Such situations are especially important in contracts involving external calls such as transfers. Existing verification techniques do not adequately address these issues and there is a clear need for methods that can detect whether a contract may enter blocking situations that compromise its intended operation. This motivates the second research question:

RQ2: *How can progress-related vulnerabilities in smart contracts, such as Denial-of-Service, be detected and verified?*

To address this question, the thesis studies smart contracts through the lens of non-blocking verification. By modeling contract behavior as a discrete event system, it becomes possible to analyze whether executions can always progress toward desired outcomes or whether certain behaviors may cause the contract to become stuck. The thesis shows how supervisory control theory can be used to detect such sequence of events leading up to a situation from which no progress can happen and thereby reveal vulnerabilities. This research question is addressed in Paper A.

The third and final research question concerns fairness of a smart contract with respect to its users. Smart contracts often govern interactions among multiple users with potentially conflicting interests. In such settings, it is important to ask whether the contract behavior gives one participant an inherent advantage over another. The notion of fairness has been sparingly discussed in the smart contract literature [25], and there is still no well-established behavioral framework for formalizing and analyzing different forms of bias in a smart contract. The lack of formal methods for determining whether a user can force a beneficial outcome, or whether a coalition of users can strategically disadvantage another participant motivates the third research question:

RQ3: *How can bias in smart contracts be formally defined and be analyzed?*

To answer this question, the thesis presents a formal framework in which bias is captured in terms of the existence of enforceable strategies over smart contract behavior. This makes it possible to analyze whether the structure of a contract gives a participant, or a group of participants, an advantage where they can achieve their goals irrespective of what other users do. The question is addressed by defining bias within the discrete event systems framework and applying it to smart contracts, thereby showing how supervisory control theory can be used to reveal strategic imbalance in contract behavior. Paper B and Paper C address **RQ3**.

Research questions addressed in the thesis, their main focus and the relevant papers that address these questions are presented in table 1.1.

Table 1.1: Coupling between the research questions, their main focus, and the corresponding paper.

Research Question	Main Focus	Paper
RQ1	Modelling smart contracts as extended finite state machines for formal verification	Paper A
RQ2	Detection of progress-related vulnerabilities and compromised liquidity	Paper A
RQ3	Formalisation and analysis of bias using supervisory control theory	Paper B, Paper C

1.3 Thesis Contributions

This thesis contributes to the formal analysis of smart contracts by developing methods for modeling and verifying properties that arise from interaction among multiple smart contract users. While existing verification approaches have largely focused on safety properties such as runtime errors, arithmetic issues, reentrancy, they are less suited to reasoning about strategic behavior between users. This thesis addresses that gap by bringing discrete event systems and supervisory control theory into analysis for smart contracts.

The *first* contribution of the thesis is a formal modeling framework for automatically converting smart contracts to EFSMs. Principles describing how smart contracts can be converted to EFSMs are presented in which the behavior of the smart contract is modeled explicitly in terms of states, events, guards, and variable updates. This provides a structured way to obtain a behavioral model directly from the contrct source code and forms the basis for applying discrete event systems methods to smart contracts. The *second* contribution is the use of generated EFSM models to analyze progress related properties of smart contracts. Specifically, the thesis provides a framework for studying whether the progress of the contract towards an intended outcome can be blocked due to malicious behavior of a contract user. The *third* contribution is a framework for analyzing bias in smart contract behavior. The thesis introduces a way to study whether the behavior of the contract can give some user(s), an advantage in reaching a desired outcome, that the user(s) is able to force a specific outcome. Together, these contributions show how smart contracts can be modeled and analyzed as discrete event systems, with the goal of understanding their behavior beyond what is immediately visible from the source code.

1.4 Thesis Outline

This thesis is divided into two main parts. Part I provides the necessary background, introduces the research questions and contributions, and summarizes the scientific work included in the thesis. Part II contains the research papers on which the thesis is based.

Chapter 1 (this chapter) introduces the overall research area and motivates the study. It presents the research questions that guide the thesis, followed by a description of the main contributions. The present section concludes the introductory chapter by outlining the structure of the thesis. Chapter 2 provides the blockchain and smart contract background required for the remainder of the thesis. It begins with an introduction to Ethereum and then discusses accounts, transactions, and the blockchain state. The chapter also presents the Ethereum Virtual Machine and the Solidity programming language, followed by a discussion of calls, transfers, and reversion. Finally, the chapter introduces the abstract syntax tree, which is the basis for modeling smart contracts as EFSMs. Chapter 3 introduces the theoretical foundation

from discrete event systems that underpins the analysis carried out in this thesis. It presents formal languages before introducing finite state machines and extended finite state machines. The chapter concludes with the supervisory control theory, which forms the main analytical framework used in the thesis. Chapter 4 elaborates in detail on the principles used for automatically modeling smart contracts as extended finite state machines. Chapter 5 summarizes the papers included in the thesis and explains their main ideas and contributions in relation to the overall research objectives. This chapter serves as a bridge between the introductory part of the thesis and the appended papers. Chapter 6 concludes the thesis by discussing the main findings and reflecting on possible directions for future work. Part II contains the full versions of the papers included in the thesis. These papers provide the detailed methodological development, analysis, and results on which the thesis is based. The papers have been reformatted to fit the format of this thesis, but are not otherwise altered.

CHAPTER 2

Smart Contracts

Smart contracts have emerged as one of the most significant applications of blockchain technology, extending distributed ledgers beyond value transfer to the execution of programmable logic. They enable rules, conditions, and interactions between parties to be encoded directly in the smart contract and executed in a decentralized environment without relying on a central authority. This makes them a powerful foundation for decentralized applications, but at the same time introduces new challenges. Since smart contracts often manage assets, errors in their design or execution may have devastating consequences. This chapter elaborates on key knowledge related to smart contracts.

2.1 Ethereum

While the idea of smart contract predates blockchain technology, Ethereum [3] was the first major blockchain to provide a general purpose infrastructure for deploying and executing smart contracts, establishing itself as a decentralized platform for programmable applications.

Ethereum can be understood as a transaction based state machine. Its *global state* contains the current status of all accounts including balances, contract

code, and contract storage [3]. The *world state* is updated only through valid transactions. These transactions are grouped into blocks and each block is linked cryptographically to the previous one, thereby forming a blockchain. Because each block contains a reference to its predecessor, modifying old data such as contract code or transaction details would require changing all subsequent blocks as well, which is computationally and economically extremely expensive. As a consequence once a smart contract has been deployed, its behavior cannot be changed, making vulnerabilities in the contract particularly costly. Ethereum also has a native cryptocurrency, Ether [3], which is used both as a transferable digital asset and also as a means of paying for computation. A user submits transactions to the Ethereum network and pays for the resources consumed during execution.

2.2 Accounts, transactions, and the blockchain state

Ethereum distinguishes between two main types of accounts, namely, Externally Owned Accounts (EOAs) and Contract Accounts (CAs). Both are identified by addresses and both may hold Ether. However, EOAs are controlled by private keys and do not contain executable code, whereas CAs contain code and persistent storage and are controlled by that code. An EOA can initiate a transaction, for example to transfer Ether or invoke a contract function. A contract account, in contrast, cannot initiate transactions on its own, since its code executes only when triggered by an external transaction or by a call from another contract.

Transactions are the mechanism by which Ethereum's state changes. At a high level, a transaction may transfer Ether between accounts, deploy a new contract, or invoke a function of an already deployed contract. When a transaction is accepted and executed, the global state is updated to reflect the resulting balance changes and storage updates. In this way, the blockchain stores not only a history of transfers, but also a history of state transitions induced by contract execution.

In Ethereum, a transfer is the movement of Ether from a sender account to a recipient account as part of a transaction or contract call. Such a transfer changes the balances recorded in the blockchain state. If the recipient is a contract account, the transfer may also trigger contract logic through

the special *receive* function or, in certain cases, the *fallback* function. Consequently, value transfer in Ethereum is not always a passive balance update; it may also induce further computation and interaction. This is one reason why Ether transfers are highly relevant in security analysis and in formal models of smart contract behavior.

2.3 Ethereum Virtual Machine

The Ethereum Virtual Machine [4] (EVM) is a stack based virtual machine that is responsible for running and executing smart contracts on Ethereum. Smart contracts are usually written in high level languages and are then compiled into EVM bytecode before deployment. The EVM is platform independent, enabling smart contracts to be developed in well known programming languages such as Solidity [26], Vyper [27], and others.

To account for computational effort and to prevent non-terminating execution, Ethereum uses the notion of *gas*. Each EVM instruction consumes a predefined amount of *gas*, and the sender of the transaction specifies the *gas* limit that bounds the total amount of computation. If the gas limit is exhausted, the execution is aborted and the state-changes are reverted.

2.4 Solidity

Although smart contracts ultimately execute as EVM bytecode, they are usually written in high-level languages, most notably Solidity [26, 28]. Solidity is an object-oriented language that not only combines elements similar to C++, JavaScript, and Python, it also includes blockchain-specific constructs for interacting with transaction data, block data, addresses, Ether transfers, and contract storage.

A Solidity contract resembles a class in object-oriented programming, typically consisting of state variables, functions, and possibly a constructor. Functions define the operations that users or other contracts may invoke. State variables are stored on the blockchain and represent the long-term state of the contract. By contrast, local variables used within functions exist only during execution and do not form part of the long-term blockchain state. An example smart contract, the SimpleBank contract is shown in Fig 2.1.

```
1  contract SimpleBank {
2      mapping(address => uint256) private balances;
3
4      function deposit() public payable {
5          balances[msg.sender] += msg.value;
6      }
7
8      function withdraw(uint256 amount) public {
9          require(balances[msg.sender] >= amount, "Insufficient
               balance");
10         balances[msg.sender] -= amount;
11         payable(msg.sender).transfer(amount);
12     }
13
14     function getBalance() public view returns (uint256) {
15         return balances[msg.sender];
16     }
17 }
```

Figure 2.1: The SimpleBank smart contract

The SimpleBank smart contract is intended to act as a simple bank-like wallet that allows users to deposit Ether, withdraw part of their stored balance, and check how much they have. The contract begins by defining a mapping `balances`, on line 2, that associates each address with its deposited Ether balance. The `deposit()` function on line 4 is labeled `payable`, meaning that it can receive Ether, and it increases the stored balance of the caller by the amount sent in the transaction. The `withdraw(uint256 amount)` function on line 8 takes `amount` as an input and first checks, using `require` on line 9, that the caller has sufficient funds. If the check succeeds, the requested amount is subtracted from the callers balance and transferred back to the caller's address, whereas, if the check fails the function reverts. Finally, function `getBalance` on line 14 provides a read-only way to look at the caller's stored balance. Details on how transfers and reverts work are presented in the next section.

2.5 Calls, transfers, reversion

A smart contract function may be invoked either by an externally owned account or by another contract. Execution of the smart contract is inherently reactive as it only proceeds when triggered by a transaction or a function

call. Calls can be of two types; *external* and *internal*. A call within the same contract is an internal call, whereas a call to another contract is an external call. An external call transfers control to another contract, allowing the other contract to execute before returning control to the caller.

Ether transfers are closely related to the calling structure. A contract may send Ether to another address as part of its execution, and if the recipient is itself a contract, its *receive* or *fallback* function may be triggered automatically. In that case, what appears to be a simple transfer of value also becomes a transfer of control, since execution may continue in code outside the calling contract. This feature is important for both correctness and security [12, 29]. The receiving contract may update its state, emit events, invoke other contracts, revert (see below), or even call back into the original sender before the initial execution context has fully completed. As a result, value transfers are not merely accounting operations, they can shape the dynamic behavior of a transaction and influence the overall sequence of interactions between contracts.

One of the most important semantic features of smart contract execution is *reversion*, which can be caused by several factors such as a *require* condition being violated, an exception occurring, insufficient funds, or attempts for unauthorized access. *Reversion* refers to the mechanism by which the effect of a transaction or internal call are rolled back to the original state before the call. When a *reversion* occurs, all state changes made during the current call, including updates to storage and balance transfers, are undone. This behavior is essential in preventing partial execution from leaving the system in an inconsistent configuration. *Reversion* has significant implication when designing a contract and its security analysis. A robust smart contract design requires careful handling of reversion conditions and use of programming practices that reduce unintended denial-of-service risks [12, 29].

2.6 Abstract Syntax Tree

An abstract syntax tree (AST) is a tree structured representation of a program representing its syntactic structure [30]. The structure is called *abstract* because it omits surface-level details of the source text, such as punctuation and formatting, and retains only the essential programming contructs and their hierarchical relationships.

The AST represents a program as a hierarchy of nodes. The root of the tree typically corresponds to the entire source unit, while its children represent major constructs such as contract definitions, variable declarations, function definitions, statements, and expressions. In this way, the AST makes the nested structure of the program explicit. For example, a function definition node may contain child nodes representing its parameter list, modifiers, local declarations, conditional statements, assignments, and return statements. The AST provides a normalized structural representation in which each node has a specific role. This makes it easier to define translation rules specific to node types. Rather than scanning source text line by line, the translation procedure can inspect each node and directly determine how that construct should be interpreted. As a result, the translation becomes more robust and easier to automate.

The abstract syntax tree for a Solidity smart contract can be generated directly by the Solidity compiler [26], *solc*. In the conversion of smart contracts to EFSMs, the AST plays a central role as an intermediate representation between the high level source code and the formal behavioral model. Each node in the AST corresponds to a specific language construct, the behavior of which is interpreted and translated into an EFSM component. For instance, contract variables are mapped to EFSM variables, and conditional statements such as *if* clauses or *require* expressions can be translated into guards on transitions. Similarly, assignments in the smart contract can be represented as updated actions in the EFSM.

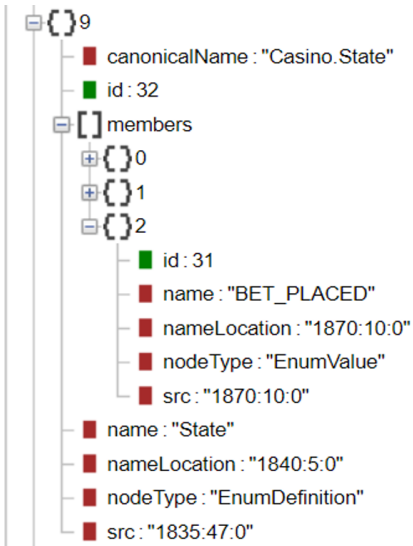


Figure 2.2: A snapshot of the abstract syntax tree for a smart contract.

Fig 2.2 shows a snapshot of the abstract syntax tree for a smart contract. The snapshot does not show the complete tree, rather captures a small section of the entire AST. As can be seen in the snapshot, Solidity constructs appear as nested nodes in the AST.

This is part of the AST for the Casino smart contract of Paper A, whose Solidity code is presented in Fig A.2. Line 3 in the Casino defines an enumeration variable `State`, with three possible values `IDLE`, `GAME_AVAILABLE`, and `BET_PLACED`. In the snapshot of Fig 2.2 these three values appear under the `members` node, and the `BET_PLACED` value has index 2. In the AST, each element is given a unique id-number, here 31 for `BET_PLACED`, and a type, denoted by `nodeType` in the AST, and for `BET_PLACED` the `nodeType` is `EnumValue`. It can also be seen in Fig 2.2 that the `State` variable has `nodeType` as `EnumDefinition` with id-number 32. In addition, there are in the AST node references to where in the source code the element appears, and some other info not relevant to the conversion from AST to EFSMs.

CHAPTER 3

Discrete Event Systems

Discrete Event Systems [17], abbreviated as DES are dynamical systems whose evolution is driven by the occurrence of events. An update in a DES, represented by a state change happens only when a relevant event takes place. Many types of systems can be modeled as DES. Examples include manufacturing systems, communication protocols, traffic systems, software processes, and in the context of this thesis, smart contracts. Modeling software systems as DES for analysis is particularly useful where program execution depends on ordered interactions between components of the software. In this thesis, DES provides the theoretical basis for modeling the behavior of smart contracts and for reasoning about what the users and the environment can or cannot force a contract to do.

This chapter introduces the main concepts needed for the remainder of the thesis. It begins with formal languages, which provide the mathematical basis for describing event sequences. The next chapters present Finite State Machines (FSMs) and Extended Finite State Machines (EFSMs). Finally Supervisory Control Theory (SCT) is introduced, which provides the main analysis framework.

3.1 Formal Languages

A DES is commonly described in terms of the events that may occur and the event sequences that the system can generate. This leads naturally to the language-theoretic view of system behavior.

Let Σ be a finite set of symbols, called the *alphabet*. In DES, the elements of Σ represent events. A *string* over Σ is a finite sequence of events drawn from Σ. The empty string is denoted by ε, and represents the sequence of length zero. The set of all finite strings over Σ is denoted by Σ^*. A language L over Σ is any subset of Σ^*. In DES, a language represents the set of event sequences that are possible in a system. This view is useful because many behavioral properties can be expressed in terms of allowed or disallowed strings. Instead of reasoning only about isolated states, one can reason about sets of possible executions.

An important distinction is whether a language is *prefix-closed* or *non-prefix-closed*. A string $s \in \Sigma^*$ is a prefix of another string $t \in \Sigma^*$ if there exists a string $u \in \Sigma^*$ such that $t = su$. A language $L \subseteq \Sigma^*$ is prefix-closed, denoted $\overline{L}$, if for each string belonging to L, all of its prefixes also belong to L. Intuitively, this means that if an execution is included in the language, then every partial execution leading up to it is also included in the language. Prefix-closed languages are therefore useful for describing ongoing behavior, where all intermediate stages of an execution are part of the system behavior. On the other hand, a non-prefix-closed language does not contain all prefixes of its strings. Such a language can be used to describe specific executions, such as reaching a certain state.

The distinction between generated and *marked* behavior is fundamental in DES. A system may be able to execute many event sequences, but only some of them may correspond to meaningful or desirable outcomes. For instance, in a contract-based application, the generated language may include all possible interaction sequences with a contract, while the marked language may represent those sequences that reach a completed transaction, a winning condition, or another relevant situation.

3.2 Finite State Machines

Finite-state machines are a common modeling framework for discrete-event systems [17, 31]. In this setting, a system is described in terms of a finite collection of states and the events associated to changes between the states. Each state corresponds to a particular condition or operating mode of the system, while each event represents an occurrence corresponding to the system moving from one state to another. This makes FSMs a natural tool for representing dynamic behavior in systems whose evolution depends on discrete actions rather than continuous change.

The behavior of an FSM is captured by transitions of the form $x \xrightarrow{s} y$, meaning that the system moves from state x to state y when the event s occurs. In addition to the transition structure, an FSM also includes a set of initial states, which specify where execution may begin, and a set of marked states, which identify states of special interest such as completion, success, or fulfillment of some task.

In many models, the FSM has exactly one initial state and, for each state and event, there is at most one possible successor state. In that case, the model is *deterministic*, since the current state can be uniquely inferred from the observed event sequence.

Formally, a finite-state machine is defined as a tuple

$$G = \langle \Sigma, Q, \rightarrow, Q^\circ, Q^m \rangle,$$

where

- Σ is the finite event alphabet,
- Q is the finite state set,
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation,
- $Q^\circ \subseteq Q$ is the set of initial states, and
- $Q^m \subseteq Q$ is the set of marked states.

The machine is deterministic when it has at most one initial state, when no state has two different outgoing transitions with the same event label.

For a plant G, two languages are especially important. The first is the generated language, denoted by $L(G)$, which contains all event sequences that the

model can execute from its initial state. The second is the marked language, denoted by $L_m(G)$, which contains those generated sequences that lead to marked, or accepted, states. The generated language captures what is possible, whereas the marked language captures represents all event sequences that reach a state of interest in the plant G.

3.3 Extended Finite State Machines

Extended finite-state machines (EFSMs) provide a natural extension of finite-state machines for systems whose behavior depends not only on events, but also on data [19, 32]. While an ordinary finite-state machine represents system behavior using states and event-labeled transitions, an EFSM extends this model with variables and expressions that control when transitions may occur and how data values evolve [19]. For this reason, EFSMs are well suited for describing systems in which the control flow and the data flow are closely connected.

As in the finite-state setting, the behavior of the system is described by locations and transitions associated with events. Transitions are of the form

$$x \xrightarrow{s:p} y,$$

where x is the source location, y is the target location, s is the event label, and that the system transits from location x to y if update p evaluates to *true*. The update p is a relation between *pre-transition* and *post-transition* variables and can be used as simple assignments or as conditions.When such a transition occurs, the EFSM moves from x to y, and the values of the variables are modified according to the update expression. Variables not affected by the update retain their values.

The expressions used in EFSM transitions are constructed from variables, constants, Boolean values, and the usual arithmetic and logical operators. Each variable is associated with a discrete domain that specifies the set of admissible values. In addition, every variable has an initial value that determines its value at the start of execution. Since variables values may change as transitions occur, the behavior of an EFSM cannot be described solely in terms of its locations. Instead, the state of the system is given by the combination of the current location and the current valuation of all variables.

To distinguish current values from values after a transition, it is common to

use a corresponding set of next-state variables, usually denoted by V'. These primed variables are used in update expressions to describe how the variables evolve when a transition is executed. In this way, EFSMs can capture both the structural behavior of the system and the evolution of its associated data in a single model.

Formally, an extended finite-state machine (EFSM) is defined as a tuple:

$$E = \langle \Sigma, Q, \rightarrow, Q^\circ, Q^m \rangle,$$

where

- Σ is the set of events,
- Q is the finite set of locations,
- $\rightarrow \subseteq Q \times \Sigma \times P_V \times Q$ is the transition relation,
- P_V is the set of all update expressions over the variable set V,
- $Q^\circ \subseteq Q$ is the set of initial locations, and
- $Q^m \subseteq Q$ is the set of marked locations.

In this thesis, the term *state machine* is used as a general term that may refer to either a finite-state machine or an extended finite-state machine depending on the context.

EFSMs often provide a more compact and intuitive representation than ordinary finite-state machines, especially when a system contains repeated behavioral patterns that differ only in the values of certain variables. However, many analysis techniques are formulated for finite-state machines rather than EFSMs. For this reason, it is often necessary to transform an EFSM into an equivalent FSM before performing analysis.

This transformation is commonly referred to as *flattening* [32]. The idea is to make the variable valuations explicit by unfolding them into the state space. The states of the resulting FSM are therefore pairs consisting of a location of the EFSM and a valuation of the variables. Initially, each initial location is combined with the initial valuation of the variables. Thereafter, transitions in the flattened FSM are generated by evaluating the update expressions of the EFSM and determining which target valuations are reachable. Consequently, each transition in the EFSM may correspond to several transitions in the flattened FSM, depending on the possible variable values for which the associated

update expression is enabled. Although flattening makes the model suitable for standard FSM-based analysis, it may also significantly increase the size of the state space. Thus, EFSMs offer a compact modeling formalism, whereas flattened FSMs provide a representation that is often more convenient for formal analysis.

In practice, complex systems are rarely described by a single monolithic machine. A more convenient approach is to model each component separately and then study the combined behavior of the collection. This modular representation simplifies modeling, since it is often easier to describe the local behavior of individual components than to construct the full system model directly.

Rather than modeling discrete event systems as one large automaton, it is often described as a collection of interacting subsystems. This modular way of modeling is useful as typically several real systems are too complicated to model cleanly with a single state machine. Instead, the overall behavior is built up from several smaller machines whose interaction captures the behavior of the full system. A standard way to represent this interaction is through *synchronous composition* [33].

Synchronous composition combines the behavior of the component subsystems into a global model. If an event labeling the transition is *shared* between components, it can occur only when the event is enabled in all the components, and the participating components move forward together when it occurs. If an event belongs to only one component, then it is local to that component and can occur independently of others. For extended finite state machines, transitions are associated with not only events but also with updates, so the state change associated with a transition includes both the control move and the update action. A formal definition of synchronous composition is presented in Definition 2 of Paper A.

Marking is a standard notion in supervisory control theory, used to distinguish states in the system that are significant for the behavior under study. Typically, a subset of the state set is designated to be marked, and these states are interpreted as representing important configurations such as completed tasks, successful termination, or other states of interest in the analysis. The presence of marked states allows the system to be analyzed in terms of whether its executions can lead to an acceptable state. In the context of this thesis, marking becomes an important part of the formal model, more so when

the objective is to reason about reachable and correct behaviors.

3.4 Supervisory Control Theory

The Supervisory Control Theory (SCT), introduced by Ramadge and Wonham [20, 34], provides a formal framework for the analysis and control of discrete event systems. The central objective of SCT is to restrict the behavior of a system so that it satisfies given requirements, preserving acceptable behavior to the greatest extent. In this framework, the *uncontrolled* system is represented by a *plant*, and the desired behavior is described by one or more *specifications* for the system. Based on these models, a *supervisor* is synthesized to restrict the plant such that the resulting closed-loop system satisfies the specified requirements.

The behavior generated by the plant includes event sequences that are feasible from the system point of view but undesirable with respect to safety, coordination, or progress. The purpose of supervision is therefore not to construct behavior independently, but to remove those evolutions of the plant that lead to violation of the specified requirements. Let the plant be denoted by G, the specification by K, and the supervisor by S. The supervisor observes the execution of the plant and determines which events remain enabled after a given sequence of previously executed events. Thus, the supervisor acts as a feedback mechanism: it does not generate events itself, but influences the future evolution of the system by restricting the set of events that may occur. Consequently, the closed-loop behavior is the set of executions that remain possible after the supervisor's restrictions have been applied. The controlled behavior can be represented by the synchronous composition of the plant G and the supervisor S. This gives a convenient representation of the closed-loop system and forms the basis for supervisor synthesis and verification.

The supervisor limits the behavior of the plant by preventing certain events from occurring. However, not all events within a system can be disabled as they may be beyond control. For supervisory control, the event set Σ, is therefore divided into two disjoint sets: *controllable events* and *uncontrollable events*. The supervisor may disable controllable events, whereas uncontrollable events must remain available whenever they are enabled in the plant.

Non-blocking, Controllability, and Minimal Restrictiveness

When synthesizing a supervisor, the aim is to construct a supervisor that preserves two fundamental properties: *controllability* and *non-blockingness*. These two properties are central because a supervisor must both respect that it cannot influence uncontrollable events and also ensure that the controlled system can still reach its intended behavior. However, the supervisor should only restrict the system as much as absolutely necessary, it should be *minimally restrictive*. This guarantees that under control of the supervisor, the system can always reach some desired state, while it retains the maximal possible behavior.

Controllability ensures that the supervisor respects the distinction between controllable and uncontrollable events. This is to ensure that if an uncontrollable event is enabled in a state of the plant reachable under control, this uncontrollable event must remain enabled in the controlled system, otherwise the supervisor would be attempting to prevent behavior that it has not authority to stop. Controllability is a fundamental property of the supervisor, since any method to control the system should remain implementable and consistent with the system's operational limits.

Non-blocking refers to that every reachable state of the supervised system must remain co-reachable with a marked state. The synthesis must identify states that prevent progress towards marked states and restrict the behavior of the plant so as to not reach those states. Non-blockingness guarantees that from every state reachable under control, some marked state can always be reached, and is thus an essential property of the supervisor.

The synthesis result is typically a *minimally restrictive* supervisor satisfying both properties. Minimal restrictiveness means that the synthesized supervisor excludes only those behaviors that must be removed to preserve controllability and non-blockingness. In other words, it keeps the maximal allowable behavior of the original system while still enforcing the specification.

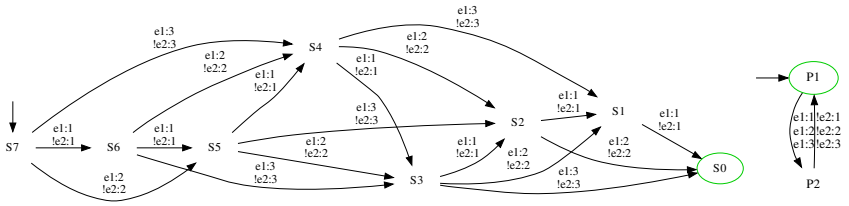


Figure 3.1: Stick-pile automaton (left) and player-choice automaton (right), for the stick-picking game. Ellipsed states are marked. Exclamation marks denote uncontrollable events.

Example 1 *Consider a stick-picking game with two players. The game consists of seven sticks with players taking turns picking sticks from a pile. On each turn, a player may pick one, two, or three sticks, provided that there are enough sticks. The player to pick the last stick loses the game. The system can be modeled by two automata, shown in Fig 3.1. One automaton (left) represents the possible stick picking, while the other (right) represents the players' choices and turn taking, with Player 1 starting the game. In the stick-pile automaton (left, Fig 3.1), the number of remaining sticks is tracked, and the transitions correspond to the progression of the game. In the player-choice automaton (right, Fig 3.1), an event ei : j represents that player i picks j number of sticks. The game is analyzed to determine whether a supervisor can be synthesized for Player 1 such that the system can be controlled by Player 1 to reach a winning state. Thus, events associated with Player 1, namely* $e1:1$, $e1:2$, *and* $e1:3$ *are controllable, whereas events of Player 2,* $e2:1$, $e2:2$, *and* $e2:3$ *are uncontrollable. The marked locations in the model represent the winning game state location* $S0$ *of the stick-pile, and the* $P1$ *location in the player-choice automaton, indicating that Player 1 wins the game. These marked states define the desired behavior against which the existence of a supervisor is assessed.*

The full game play is given by the synchronous composition (formally defined in Paper B, page B6) of the stick-pile and the player-choice automata, shown in Fig 3.2. The synchronous composition of the two automata presented in Fig 3.1 is obtained by forming the product of their state spaces and synchronizing all shared events; this then represents all possible behaviors in the composition of the two automata. A transition in the composed model occurs when the event is enabled in both component automata. The uncontrollable

events, that is events associated with Player 2, are denoted by an exclamation mark, $!e2:1$, $!e2:2$, *and* $!e2:3$. *The marked state in the synchronous composition,* $P1:S0$ *in Fig 3.2, is determined by the marking of the component EFSMs. That* $P1:S0$ *is marked means that when the system reaches* $P1:S0$, *all components of the composed EFSM have reached their respective marked states,* $P1$ *and* $S0$. *Thus,* $P1.S0$ *represents that it is Player 1's turn to pick but there are no sticks left, which means that Player 2 picked the last stick and hence lost the game.*

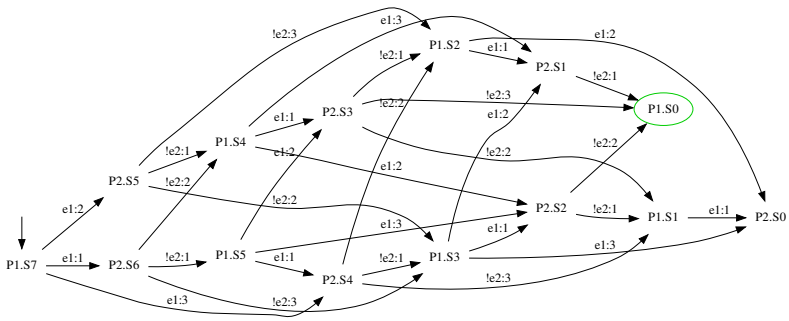


Figure 3.2: The synchronous composition of the stick-pile and the player-choice automata of Fig 3.1. This describes all possible ways to play the stick-picking game when Player 1 starts.

It is also noteworthy from Fig 3.2 that the state $P2.S0$ *is not marked. This is because this state represents that it is Player 2's turn to pick but there are no sticks left, hence Player 1 lost the game. This is possible in the game play, but not desired, and therefor this state is not marked. There is also no way to reach a marked states from* $P2.S0$, *which means that this state is blocking.*

During the synthesis, some states are removed because not every reachable state is acceptable under the desired specification. In supervisory control, the goal is to keep only those states from which the system can still behave correctly and eventually reach some marked state. In the synthesis process, states that violate the specification are eliminated from the full composed model. In particular, a state is removed if it is not co-reachable with a marked state, that is, reaching a marked state is no longer possible from there on, or if the state has an uncontrollable transition that leads to a blocking state. The elimination

is carried out backwards by repeatedly removing states from which no marked state remains reachable, or states that through uncontrollable events lead up to them.

For instance, in Fig 3.3, from states $P2 : S0$ *and* $P1 : S1$*, highlighted by red boxes, no marked state can be reached and thus these states have to be eliminated. On the following iterations, states that have uncontrollable events leading up to* $P2 : S0$ *and* $P1 : S1$ *must also be removed.*

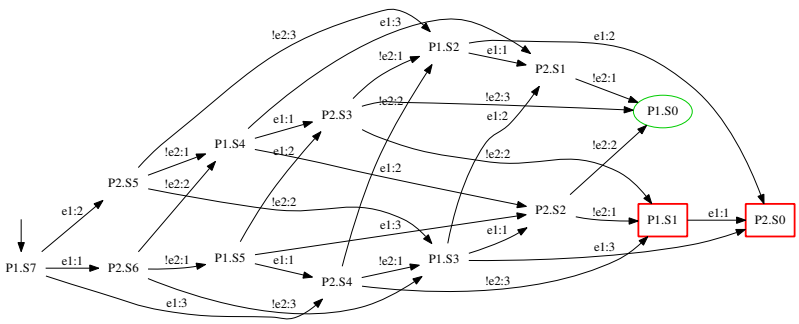


Figure 3.3: Full synchronous composition of stick-pile and player-choice automata with blocking states boxed.

This is shown in Fig 3.4, where the next set of states to be removed are $P2 : S3$*,* $P2 : S2$*, and* $P2 : S4$*, as they have the uncontrollable events* $!e2 : 2$*,* $!e2 : 1$*, and* $!e2 : 3$*, respectively, leading to the blocking state* $P2 : S1$*. Note that the transition labeled by* $e1 : 2$ *from* $P1.S2$ *to the blocking state* $P2.S0$ *does not make it necessary to remove* $P1.S2$*, since the event* $e1 : 2$ *is controllable, and can thus be disabled by the supervisor to control the system not to reach* $P2.S0$*.*

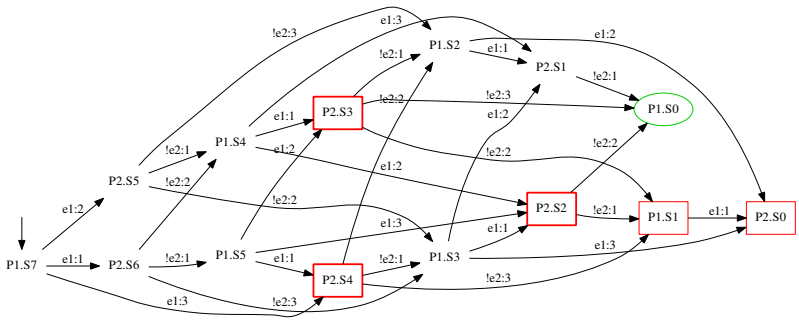


Figure 3.4: Full synchronous composition of stick-pile and player-choice automata with blocking and uncontrollable boxed.

From Fig 3.4, the synthesis continues for two more iterations, as from $P1.S5$ *no marked state can be reached without passing through boxed states, and then from* $P2.S6$ *the uncontrollable event* $!e2:1$ *reaches* $P1.S5$*. Thereafter, no more states have to be removed.*

After synthesis, the resulting supervisor, shown in Fig 3.5 is obtained. The supervisor disables events $e1:1$ *and* $e1:3$*, allowing Player 1 to only pick 2 sticks on the first turn. Subsequently, all uncontrollable events* $!e2:1$*,* $!e2:2$*, and* $!e2:3$ *are allowed at location* $P2:S5$*. From there on, whatever Player 2 chooses, Player 1 can then force Player 2 to take the last stick and lose the game.*

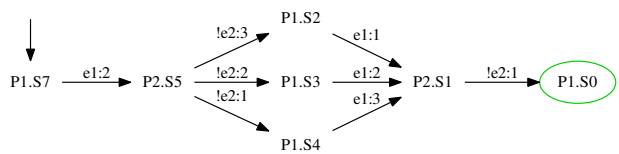


Figure 3.5: Supervisor generated for the stick-picking game.

CHAPTER 4

Conversion of Smart Contracts to Extended Finite State Machines

For the purpose of our formal analysis, smart contract behavior needs to be represented as a discrete event model. The model should be expressive enough to capture the important behavioral aspects of the smart contract, but at the same time abstract enough to remain finite and compact enough so that it can be analyzed. In this thesis, Solidity smart contracts are converted into extended finite-state machines (EFSMs). The EFSM model acts as a model that represents Solidity smart contract and allows formal analysis methods used later in the thesis.

This chapter describes the conversion mechanism from Solidity smart contracts to EFSMs. The chapter first motivates the choice of EFSMs as the modeling formalism. It then explains the overall conversion pipeline, the representation of functions, parameters, caller identity, variables, guards, modifiers, internal function calls, nondeterminism, and function failure. Finally, the chapter discusses the abstractions made by the conversion and explains why these abstractions are necessary.

4.1 Smart Contracts as EFSMs

Extended finite-state machines are a suitable modeling formalism for Solidity smart contracts because they combine control-flow structure with data-dependent behavior. A standard finite-state machine represents behavior using states and events. However, a Solidity smart contract also contains variables and conditions over those variables. The execution of a function often depends on whether a condition evaluates to true, and the execution of statements often changes the values of variables. EFSMs are designed to represent precisely this type of behavior.

An EFSM transition may contain three components: an event, a guard, and an action. The event represents the occurrence of something in the system, such as a function call or an internal step in the function execution. The guard represents a condition that must hold for the transition to be enabled. The action represents the update of variables when the transition is taken.

This structure corresponds closely to Solidity. A Solidity function is guarded by preconditions, often written as`require` statements or modifiers. These preconditions can be represented as EFSM guards. Assignments to state variables and local variables can be represented as EFSM actions. Function invocations and important execution steps can be represented as events. Thus, EFSMs provide a natural way to translate Solidity programs into discrete-event models. For instance, the Solidity statement

$$\texttt{x} = \texttt{x} + 1;$$

can be represented as an EFSM transition whose action updates the variable x. Similarly, the Solidity statement

$$\texttt{require}(\texttt{x} > 0);$$

can be represented as a guard that must be satisfied before the corresponding transition is enabled.

4.2 Overview of the Conversion Pipeline

The conversion begins with the Solidity source code. The Solidity compiler is used to obtain an abstract syntax tree (AST) representation of the contract [26]. The AST contains structured information about the contract as mentioned in Section 2.6.

The conversion framework traverses the AST and identifies the Solidity constructs that are relevant for EFSM generation. In particular, the conversion extracts:

- contract-level state variables,
- local variables declared inside functions,
- function definitions,
- function parameters,
- assignment statements,
- expression statements,
- `require` statements,
- modifiers,
- transfer-related statements,
- internal function calls, and
- global variables such as `msg.sender` and `msg.value`.

Each Solidity function is then converted into a separate EFSM. Additional EFSMs are generated for preconditions, modifiers, parameter choices, and global assignments when needed. The overall conversion can be summarized as follows:

(i) Obtain the AST representation of the Solidity contract.

(ii) Extract state-variable declarations and create corresponding EFSM variables.

(iii) Identify function definitions in the AST.

(iv) Generate EFSM for each function.

(v) Translate statements inside each function into EFSM transitions.

(vi) Translate assignments into transition actions.

(vii) Translate `require` conditions and modifiers into guards.

(viii) Generate auxiliary EFSMs for parameter choices and caller modeling.

(ix) Model function failure and state reversion.

This pipeline makes the conversion systematic. Each relevant Solidity construct has a corresponding representation in the EFSM model.

4.3 Modeling Functions

Functions are the main entry points through which users interact with a smart contract. Therefore, the conversion treats each Solidity function as a separate behavioral component. For every AST node of type `FunctionDefinition`, the conversion creates an EFSM representing the execution of that function.

Let f be a function in a Solidity contract. The EFSM generated for f is denoted by G_f. The transitions in G_f follow the order of statements in the Solidity function.

The first transition(s) of G_f represents the invocation of the function. After this transition, the EFSM proceeds through the statements of the function in the same order as the Solidity source code. A simple assignment statement usually corresponds to a transition with an action. A condition corresponds to a guarded transition. A `require` statement can act as a precondition for the function, the modeling of which is discussed later in Section 4.3.

For example, Fig 4.1 shows a simple Solidity function with its converted EFSM.

```
1  functin inc() public {
2      require(x<5);
3      x = x + 1;
4  }
```

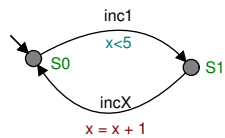


Figure 4.1: A function example in Solidity (left), with the simplified converted EFSM (right).

The EFSM model in Fig 4.1, right, has the initial state `S0`. The require statement `require(x < 5);` is converted to the guard, `x<5`, on the transition from location `S0` to `S1`, labeled by the invocation event `inc1`. The assignment

statement on line 3, `x = x + 1;` is converted to an action on the transition from `S1` to `S0`, associated with the event `incX`, where the suffix "X" at the end of `inc`, is a naming convention denoting the successful completion of the function `inc()`, and is used throughout the conversion framework. This example illustrates the idea: the EFSM follows the structure of the Solidity function while representing conditions and updates explicitly.

Function Invocation and Parameters

A function invocation in Solidity is not merely a jump to the function body. It also includes information about the caller and the parameter. In particular, the contract may inspect `msg.sender` and `msg.value`. Moreover, the function may receive parameter values from the caller.

The initial transition of the function EFSM is labeled with the invocation event. The same event may also appear in other EFSMs, such as modifier EFSMs, `require` EFSMs, or parameter-choice EFSMs. Through synchronization, the function can start only when all the associated conditions are satisfied.

Function parameters are modeled as part of the function invocation. This is important because the parameter value is chosen by the caller at the time the function is called. The EFSM model should therefore not treat parameter assignment as an ordinary internal assignment that occurs later in the function. Instead, the parameter assignment is associated directly with the invocation event.

Let f be a function with a parameter p. Suppose the declared users are U

$$U = \{u_1, u_2, \ldots, u_n\}.$$

the function model has an invocation event, corresponding to each user.

`fi`.

The occurrence of this event means that user u calls function f and passes a parameter.

The parameter choice is represented using an auxiliary EFSM. This auxiliary EFSM contains a self-loop transition with the event as the invocation event in which the parameter is to be passed. The event in the auxiliary EFSM synchronizes with the first event of the function EFSM. The parameter value

is non-deterministically assigned. For instance, an EFSM modeling parameter assignment is shown in Fig 4.2. As a result, the function invocation and the parameter assignment occur simultaneously in the model.

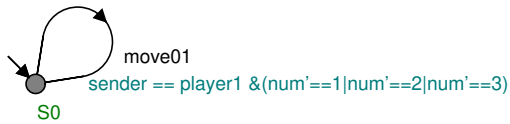


Figure 4.2: EFSM model for parameter assignment of function `move` by `player1`, and non-deterministic assignment of parameter `num`.

This approach has two advantages. First, it reflects the semantics of function calls more accurately, because parameters are passed at the moment of invocation. Second, it allows the model to reason about the caller identity and the parameter value at the same transition. This is useful when the function contains guards that depend on both the caller and the parameter value.

For example, consider a function

$$\texttt{move(uint num)}$$

where `num` can take values in $\{1, 2, 3\}$. If the users are `player1`, `player2`, and `player3`, then possible invocation events include

$$\texttt{move01}, \quad \texttt{move02}, \quad \texttt{move03}.$$

corresponding to each player respectively.

The generated EFSM for parameter assignment, shown in Fig 4.2 ties the event `move01` with the invocation event of the function `move`. The guard on the self-loop transition in Fig 4.2 checks if the function is called by `player1` and non-deterministically assigns the value of num to either 1, 2 or 3.

`require` Statements

The `require` statement is one of the most important Solidity constructs for controlling execution. A `require` statement checks whether a condition holds. If the condition is true, execution continues. If the condition is false, the function fails and the state changes made during the execution are reverted.The

revert sets back the system state to what it was before the `require` was executed. A common practice of using `require` statements is to use them as preconditions for a function call, specifying the conditions under which the function can be called.

Two types of `require` statements can be used in a function body: leading `require` statements and non-leading `require` statements. Leading `require` statements are those that appear at the beginning of a function, which are more generally used, and act as preconditions for function calls. Whereas non-leading `require` statements are the ones that appear later in the function body. For the purpose of keeping the generated model compact, and closer to common smart contract practices, only leading `require` statements are currently considered. Consider the following Solidity code:

```
1  function withdraw() public{
2      require(msg.sender == operator);
3      get_amount = true;
4  }
```

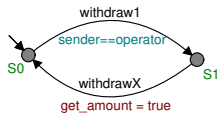


Figure 4.3: A function example with `require` on line 2 (left), with the simplified converted EFSM (right).

The condition `msg.sender` == `operator` for the `require` statement for function presented in Fig 4.3 determines whether the function is allowed to start or not.

Leading `require` statements are modeled in two distinct ways, depending on whether the `require` statement contains the function parameter or not. If the `require` statement does not contain the function parameters, it is modeled as a guard on the invocation transition of the function EFSM, as shown in Fig 4.1. However, if a parameter is present in the `require` condition, it has to be modeled differently. The reason being that an assignment to the parameter must happen at the time of the function call and cannot happen after the function has been called. So, any `require` condition involving the parameter has to be evaluated before the function is called. Modeling it as a separate EFSM addresses this issue and is explained in the remainder of this section.

Each such EFSM, associated with the `require` statement, contains a single location and a self-loop transition. The self-loop is labeled with the same

event as the invocation transition(s) of the function EFSM. The guard of the self-loop is the condition of the `require` statement. Since, multiple users can call the function and pass a parameter, invocation events associated with each user share the same self-loop transition. For instance, consider three users of a contract, `player1`, `player2` and `player3`, that they may call the `move` function presented in Fig 4.4 (left).

```
1  function move(uint num) public{
2      require(num >= 1 && num <= 3);
3      ...
4  }
```

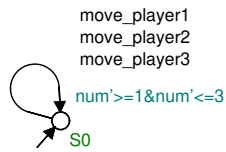


Figure 4.4: A function example with a `require` statement containing the parameter `num`.

The obtained EFSM for the `require` statement in Solidity code is shown to the right of Fig 4.4. This is a single location automaton with a self-loop transition with events corresponding to invocation events for each user. In Fig 4.4, right, the guard `num'>=1 & num'<=3` checks that the value passed to `num` must be between 1 and 3 after the transition occurs. An important thing to note here is that variable `num` is *primed* (`num'`) in the generated guard expression. Thus, the guard in the EFSM model in Fig 4.4 checks whether the variable `num` can be assigned a value between 1 and 3 once the transition has occurred. This representation has a modular structure and makes the generated model easier to represent and allows preconditions to be inspected independently.

Modifiers

Modifiers in Solidity are used to add reusable preconditions or additional behavior to functions. A common pattern is to define a modifier that checks a Boolean condition and then allows the function to execute only if the condition holds. For example consider the modifier `onlyOperator` in Fig 4.5, where there is also a function `withdraw` that uses this modifier to guarantee that the sender is always the `operator`. This is effectively the same behavior as using the leading `require` as in Fig 4.3.

```
1  modifier onlyOperator(){
2      require(msg.sender == operator);
3      _;
4  }
5  function withdraw() public onlyOperator{
6      ...
7  }
```

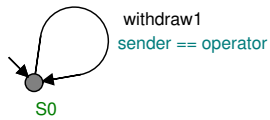


Figure 4.5: A modifier guaranteeing that only the `operator` can invoke a function.

In the conversion, modifiers are handled similarly to leading `require` statements. A modifier is treated as a precondition on the function invocation. For each modifier, the conversion creates a separate EFSM with a single location and a self-loop transition, see the EFSM to the right in Fig 4.5. The self-loop is labeled with the initial event of the function to which the modifier applies. The guard of the self-loop is the Boolean expression extracted from the modifier.

Several functions may use the same modifier as a precondition for the function to be called. In that case, the self-loop transition for the modifier is labeled by the initial events of all functions using the modifier. For instance, Fig 4.6 shows the EFSM for a modifier used in `createGame`, `addToPot`, `removeFromPot`, and `decideBet` functions, in the Casino smart contract of Paper A (see source code in Fig 2 of Paper A). The events `createGame1`, `addToPot1`, `removeFromPot1`, and `decideBet1` in Fig 4.6 are the invocation events within the generated EFSMs for functions `createGame`, `addToPot`, `removeFromPot`, and `decideBet` respectively, and thus having these invocation events in the EFSM model for the modifier ensures that function invocation can occur only when the guard `sender == operator` evaluates to true.

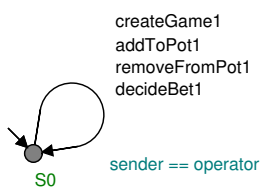


Figure 4.6: EFSM model of a modifier used by multiple fucntions.

More complex modifiers may include additional statements before or after the function body. Such modifiers require further modeling decisions. In the current conversion, the focus is on modifiers that act as Boolean guards on function invocation.

Representing modifiers as separate EFSMs has two advantages. First, it keeps the function EFSM compact. Second, it makes the modifier visible as an independent behavioral constraint. This is also useful when analyzing which conditions restrict the execution of a function.

Internal Function Calls

Solidity functions may call other functions within the same contract. These are internal function calls. The behavior of the calling function then depends on the behavior of the called function. If the called function completes successfully, the calling function continues. If the called function fails, the calling function also fails and its state changes are reverted.

The EFSM model represents internal function calls through synchronization between the EFSM of the calling function and the EFSM of the called function. Suppose function f calls function h. The EFSM G_f contains an event, for a self-loop transition, representing the invocation of h. This event synchronises with the first transition of G_h.

At the point of the internal call, the calling function has two possible outcomes: the called function succeeds, or the called function fails. Therefore, the calling function EFSM contains transitions representing both cases. If the called function succeeds, execution proceeds to the next statement in the calling function. If the called function fails, the calling function moves to its failure path and back to its initial location. The called function could fail for several reasons, one of them being that the called function contains a `.transfer()` function which might not be successful, or if it calls to another function and that results in a failure.

For example, consider a simple function `getAmount()` that has checks whether the boolean variable `getAll` is true or false. If the value of `getAll` is true, a `retrieve()` function is internally called which attempts to make a transfer. The generated EFSMs are shown in Fig 4.7 and Fig 4.8. The internal call to `retrieve()` is modeled by a self-loop transition at location $S1$ in the Fig 4.7. If the transfer in function `retrieve()` is successful, the event `retrieveX` occurs simultaneously in both the EFSMs. Whereas, if the transfer were to fail

due to some reason, then `retrieve()` also fails and consequently the event `retrieveFail` occurs in both the EFSMs bringing them back to their initial locations.

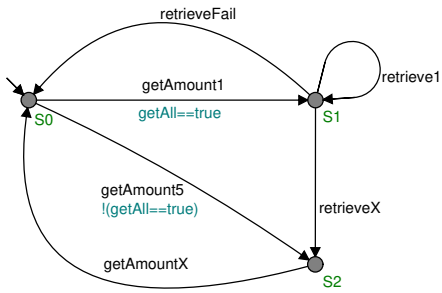


Figure 4.7: EFSM models for function `getAmount()`

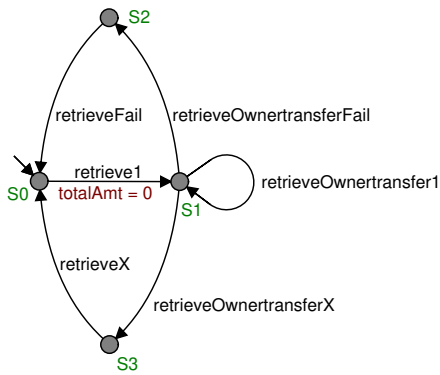


Figure 4.8: EFSM models for function `retrieve()`

Function Failure and State Reversion

One of the important semantic features of Solidity is reversion. If a function fails due to a violated `require` condition, a failed internal call, or another exception-like behavior, the state changes made during that function execution are reverted. The contract state after the failed execution should be the same as the state before the function was invoked.

The EFSM model captures this behavior by storing the values of global variables at the beginning of function execution. These stored values are kept in temporary variables. If the function later fails, the temporary values are assigned back to the global variables.

Let x be a global variable. At the beginning of a function execution, the model stores its value in a temporary variable x_{tmp}:

$$x_{tmp} := x.$$

During the execution of the function, the global variable x may be updated directly. If the function completes successfully, no restoration is needed. The current value of x is the committed value. If the function fails, the model executes an action

$$x := x_{tmp},$$

thereby restoring the value of x to the value it had at the start of the function call.

There are two possible modeling strategies for handling reversion. The first strategy is to execute the function on temporary copies of the global variables and copy the temporary values back to the global variables only when the function completes successfully. This strategy avoids the need to restore values on failure because the global variables are not updated during execution.

However, this approach causes problems when a function contains an internal function call. In Solidity, an internal function call should observe the updated state produced so far in the execution of the calling function. If the model works only on temporary copies and does not update the global variables until the end of the function, then the called internal function may not see the correct intermediate state.

For this reason, the conversion uses the second strategy. The function execution updates the global variables directly. The temporary variables are used only to restore the old values if the function fails. This strategy better reflects the behavior required for internal function calls. It allows intermediate updates to be visible during the execution, while still preserving Solidity-style reversion in case of failure.

4.4 Modeling Variables

Variables are a fundamental element of Solidity and form the basis for how smart contracts store and manage data. In the blockchain context, variables are used to represent values that may either persist within the contract's state or exist temporarily during the execution of a function. Because Solidity is a statically typed language, each variable must be declared with a specific data type. From a conceptual perspective, variables in Solidity can be understood in terms of scope and persistence. *State variables* are stored on the blockchain and remain available between transactions, whereas *local* variables are temporary and exist only during function execution.

State Variables

State variables define the persistent state of a Solidity contract. They are stored on the blockchain and retain their values between function calls. In the conversion, each relevant Solidity state variable is translated into an EFSM variable. For example, a Solidity declaration such as

```
uint total;
```

is translated into an EFSM variable `total`. A declaration such as

```
address whoseTurn;
```

is translated into an EFSM variable whose domain consists of the relevant addresses.

Since EFSMs used for analysis must have finite variable domains, Solidity types are bounded in the model. For instance, a Solidity variable of type `uint` can take a large range of values, and modeling the entire range is not practical due to state-space explosion. Therefore, the conversion initially assigns a low finite domain to such variables. This domain can later be customized depending on the contract and the property being analyzed.

For example, in a game contract where a variable `total` ranges only from 0 to 9, the EFSM variable can be given the finite domain

$$D(\texttt{total}) = \{0, 1, 2, \ldots, 9\}.$$

This is sufficient to capture the relevant behavior of the game. In other contracts, the domain may be selected differently depending on the property of interest.

Representation of `msg.value`

The value sent with a transaction is represented through `msg.value` in Solidity. A contract may use `msg.value` to check whether the caller has sent the required amount of Ether. For example, a function may contain a condition such as

$$\texttt{require}(\texttt{msg.value} == \texttt{entryFee});$$

or

$$\texttt{require}(\texttt{msg.value} > 0);$$

In the EFSM model, `msg.value` is treated as part of the environment of a function call. Similar to parameters, its value is assigned at the time of function invocation. By default, `msg.value` is modeled using the finite domain [0,1]. This domain may be refined or extended depending on the values relevant to the contract behavior. For example, instead of modeling all possible Ether values, the model may distinguish only between values that are relevant for the guard, such as whether the value is equal to a required amount or not.

The purpose of modeling `msg.value` is not to capture exact financial amounts. Rather, it is to preserve the control-flow effect of conditions involving `msg.value`. If the contract allows execution only when a certain value is sent, then the EFSM guard should reflect that condition.

Representation of `msg.sender` and Caller Identity

The identity of the caller is central to many smart contracts. Solidity exposes the caller through the global variable `msg.sender`. Contracts often use `msg.sender` to restrict access, enforce turn-taking, identify owners, or determine which participant receives a payment.

In the EFSM model, caller identity is represented by variables corresponding to the addresses declared or used in the contract. If the contract contains addresses such as `owner`, `player1`, `player2`, or `player3`, then corresponding variables are introduced in the EFSM model. The value of `msg.sender` is assigned as part of the external call modeling.

A guard such as

$$\texttt{require}(\texttt{msg.sender} == \texttt{owner});$$

is translated into an EFSM guard that compares the current caller with the variable representing `owner`.

The invocation event and the caller assignment together represent the external user action. This allows the model to distinguish between the same function being called by different users. Such a distinction is important, because behavioral properties analyzed in this thesis depend on which participant is able to perform which action. Details on how the non-deterministic assignment of msg.sender is modeled is presented in Paper A.

Local Variables

Local variables are variables declared inside a Solidity function. They exist only during the execution of the function. In the EFSM model, local variables may also be represented as variables, but their scope is associated with the function EFSM.

A local variable assignment is translated into an EFSM action in the same way as a state-variable assignment. However, the role of local variables is different. State variables affect future function calls because they persist across transactions. Local variables, on the other hand, usually influence only the current function execution.

Variable Updates

Variable updates in Solidity are represented as actions on EFSM transitions. Consider a transition

```
1  function counter() public{
2      count = count + 1;
3  }
```

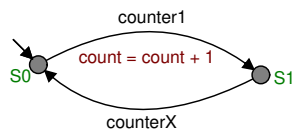


Figure 4.9: A simple function example with an update on variable `count`.

The conversion supports direct assignments, arithmetic updates, and expression statements that modify variables. In general, the right-hand side of the Solidity assignment is translated into an expression over EFSM variables. The left-hand side identifies the variable to be updated.

4.5 Modeling Transfers

Smart contracts often perform transfers or interact with external contracts. An external call may execute code in another contract, may fail, may consume gas, and may introduce additional behaviors that are not present in the original contract alone. In Solidity, the `address.transfer()` function is used to send funds from one contract to a payable address. If the receiver is an externally owned account, the transfer can be completed without additional contract logic. However, if the receiver is another smart contract, then its `receive` or `fallback` function is executed as part of the funds transfer. If the the receiving contract rejects the tarnsfer, by not having a receiving function, or by consuming more gas than is available, or by explicitly specifying conditions that revert the transfer, then the transfer fails. When `.transfer()` fails, it *reverts* the execution. This means that the function call in which the transfer occured is interrupted, and the state changes made during that function call execution are undone. For modeling purposes, the abstractions captures the aspect that is relevant for the analyzed contract, that is, whether the receiver accepts or rejects the transfer. Details of the modeling process and the resulting models are presented in Paper A.

CHAPTER 5

Summary of included papers

This chapter provides a summary of the included papers.

5.1 Paper A

Nishant Parekh, Wolfgang Ahrendt, Martin Fabian
Smart Contract Denial-of-Service Analysis Using Non-Blocking Verification
Published in Discrete Event Dynamic Systems, vol. 35, pp. 355–387, Dec 2025.

The paper investigates how denial-of-service vulnerabilities can arise in Solidity smart contracts when transfers fail or are rejected by a receiving party. In particular, the paper focuses on reachability-related denial-of-service, where desirable states of a contract become unreachable because a malicious party exploits reverting transfers. An important contribution of the paper is an automatic method for converting Solidity smart contracts to extended finite state machines (EFSM). The automatically generated model is treated as the

plant, to which two additional specification components are added: an attacker model and a progress specification. The attacker model captures the malicious behavior, attempting a denial-of-service attack, that once a party has rejected a transfer, it will continue to reject subsequent transfers. The progress specification identifies the behavior that should remain achievable despite such adversarial behavior. By combining these components, the paper shows how non-blocking verification can determine whether malicious transfer rejection can prevent the contract from reaching marked states. The method is demonstrated on two case studies: a Casino contract and an Auction contract. In the Casino case, the analysis shows that if a winning player rejects the transfer of their payout, the contract may fail to return to a state, from which the operator can retrieve funds, thus compromising the contract's liquidity. In the Auction case, the analysis also reveals a vulnerability where a malicious bidder can reject reimbursement transfers and thereby disrupt the normal bidding process to the detriment of the auction owner. To address these vulnerabilities, the paper adopts the standard pull-over-push design pattern. Instead of directly transferring funds within contract operations, the corrected contracts record the amount owed and require the receiving party to withdraw the funds as a separate external call. The verification results show that, once this correction is applied, the smart contract is free from the blocking behavior caused by malicious transfer rejection. The paper demonstrates not only that non-blocking verification can identify denial-of-service vulnerabilities in smart contracts, but also that it can be used to validate the proposed correction.

Contributions: The paper presents an automatic approach for translating Solidity smart contracts into EFSM models and shows that behavior of these models can be checked with non-blocking verification to detect denial-of-service vulnerabilities caused by failed external calls. Through the use cases, it demonstrates that the method can expose blocking scenarios, generate counterexamples that explain the source of the vulnerability, and verify that the vulnerability is relieved in the corrected contracts. More broadly, the work contributes to bridging formal verification and adversarial analysis by showing how discrete-event models can be used to study smart contracts under malicious behavior.

5.2 Paper B

Martin Fabian, **Nishant Parekh**, Wolfgang Ahrendt
On Bias in User-Centric Discrete-Event Systems
Proc. 18th Workshop on Discrete Event Systems, Eindhoven, Netherlands, May, 2026.

This paper introduces the notion of user-centric discrete event systems (UCDES), a class of discrete event systems in which multiple users interact with the system, with each user aiming to achieve a certain goal. In such systems it is common for users to have competing goals. Such goals could include either goals that are beneficial to the specific user or the user could be aiming to reach a goal which puts other users in a disadvantageous position. If the system behavior allows, the specific user could enforce a strategy to achieve their goals irrespective of actions by other users. The central question addressed in the paper is to find out whether the structure of the system contains *bias*, that is, behavior of the system that allows certain users to guarantee the achievement of their goals. A key point introduced in the paper is that an enforceable strategy is a special type of supervisor. While a supervisor ensures that marked states remain reachable, an enforceable strategy, revealed by the supervisor, must additionally be free from cycles of non-marked states, so that reaching marked states is actually guaranteed. The proposed method detects bias by configuring controllability of events and marked states according to the user perspective being analyzed. The paper demonstrates the method on stick-picking games and a Two-Machines-and-a-Buffer example. These examples show that along with individual users having an enforceable strategy, coalitions of user may also be able force a specific outcome detrimental to other users.

Contributions: The paper defines user-centric discrete-event systems, that are systems with which multiple users, each with their own intents and goals, interact; this is a generalization of things like multi-player games and smart contracts. Such systems can exhibit bias towards or against a user or set of users, and the paper shows how the existence of a supervisor can reveal the existence of bias.

5.3 Paper C

Nishant Parekh, Wolfgang Ahrendt, Martin Fabian
On Detecting Bias in Smart Contracts Using Supervisor Synthesis
Proc. 18th Workshop on Discrete Event Systems, Eindhoven, Netherlands, May, 2026.

This paper views smart contracts as user-centric discrete-event systems, and investigates whether smart contract behavior can be advantageous or disadvantageous for some users, even when such a bias is not immediately obvious from the source code. The work is motivated by the fact that smart contracts operate in a trustless environment, that the users trust the code of the smart contract, making it important to detect hidden favorable or unfavorable behavior before users interact with them. Building on earlier work on automatic conversion of smart contracts to EFSMs and on non-blocking verification, the paper extends the analysis from denial-of-service resilience to the broader question of bias detection in multi-user smart contracts. The paper uses a formal method, presented in Paper B, for detecting bias in smart contracts by combining EFSM-based modeling of smart contracts with supervisor synthesis. A central contribution is the formulation of the concepts of *positive bias* and *negative bias* in the context of smart contracts, together with their interpretation in terms of strategies available to users or coalitions of users. Building on these definitions, the paper shows how controllability assignments and state marking can be configured to detect whether a user can force beneficial outcomes for itself, or whether other users can force detrimental outcomes against that user. The results show that, in the three-player version of the NineGame case study, no positive bias exists toward any individual player, since no supervisor can be synthesized that gives a guaranteed winning strategy to a single player. However, the analysis also shows that negative bias does exist for each player such that the other two players can coordinate their actions so as to force that player to lose regardless of what the player does. Thus, although the game appears fair when only individual winning strategies are considered, the synthesized supervisor reveals that collusion introduces a disadvantage for the excluded player.

Contributions: The paper contributes a formal method for detecting bias in smart contracts through supervisor synthesis. It shows how controllability assignments and state marking can be configured to capture both positive

bias toward a user and negative bias against a user, and demonstrates on the NineGame case that even when no single player has a guaranteed winning strategy, colluding players may still force another player to lose. In this way, the paper extends smart contract verification beyond safety and security properties to the analysis of strategic advantage and collusion.

CHAPTER 6

Concluding Remarks and Future Work

This thesis explores how formal methods from discrete event systems and supervisory control theory can be used to analyze smart contracts. Smart contracts are software programs that manage digital assets and execute in a trustless environment. Once deployed, their behavior cannot be changed, which makes it important to develop methods that can reveal undesirable behavior before such contracts are used in practice.

The necessary background for this study is developed in the early chapters of the thesis. Chapter 2 introduces the smart contracts environment including Ethereum, transactions, the Ethereum Virtual Machine, Solidity, transfers, reversion and abstract syntax trees. These concepts provide the basis for understanding how smart contracts execute. Chapter 3 introduces the DES background required for the thesis, including formal languages, finite state machines, extended finite state machines, and supervisory control theory. Together, Chapter 2 and Chapter 3 establish the main foundation for the thesis, that is, the domain of smart contracts and the formal modeling framework used to analyze them, whereas Chapter 4 presents how smart contracts can be converted to extended finite state machines. As for future work, this work is only a starting point, and much remains to be explored in future.

References

[1] C. Christian and G. J. S., "Some simple economics of the blockchain," Tech. Rep. 22952, Dec. 2016. DOI: 10.3386/w22952. [Online]. Available: https://www.nber.org/papers/w22952.

[2] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[3] V. Buterin, *Ethereum: A next-generation smart contract and decentralized application platform*, 2014. [Online]. Available: https://ethereum.org/whitepaper/.

[4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2023. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[5] Wikipedia contributors, *Decentralized application*, Accessed: 2026-07-07, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Decentralized_application.

[6] E. Jiang, B. Qin, Q. Wang, Z. Wang, Q. Wu, J. Weng, X. Li, C. Wang, Y. Ding, and Y. Zhang, "Decentralized finance: A survey," *arXiv preprint arXiv:2308.05282*, 2023. [Online]. Available: https://arxiv.org/abs/2308.05282.

[7] O. Rikken, M. Janssen, and Z. Kwee, "The ins and outs of decentralized autonomous organizations (DAOs) unraveling the definitions, characteristics, and emerging developments of DAOs," *Blockchain: Research*

and Applications, vol. 4, no. 3, p. 100 143, 2023, ISSN: 2096-7209. DOI: 10.1016/j.bcra.2023.100143.

[8] OECD, "Tokenisation of assets and distributed ledger technologies in financial markets," Organisation for Economic Co-operation and Development, Tech. Rep., 2025. DOI: 10.1787/40e7f217-en.

[9] McKinsey & Company, *Tokenized financial assets: From pilot to scale*, 2024. [Online]. Available: https://www.mckinsey.com/industries/financial-services/our-insights/from-ripples-to-waves-the-transformational-power-of-tokenizing-assets.

[10] N. Szabo, "Smart contracts: Building blocks for digital free markets," in *Extropy #16 Q1 '96*, 1. 1996, vol. 8. [Online]. Available: https://github.com/Extropians/Extropy/blob/master/Extropy-16_FirstQuarter-1996.pdf.

[11] S. Tikhomirov, "Ethereum: State of knowledge and research perspectives," in *Foundations and Practice of Security*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:3423063.

[12] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts," in *Principles of Security and Trust*, ser. Lecture Notes in Computer Science, vol. 10204, Springer, 2017, pp. 164–186. DOI: 10.1007/978-3-662-54455-6_8.

[13] Chainalysis Team, *$2.2 billion stolen from crypto platforms in 2024, but hacked volumes stagnate toward year-end as dprk slows activity post-july*, 2024. Accessed: Jun. 27, 2026. [Online]. Available: https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2025/.

[14] Chainalysis Team, *2025 crypto theft reaches $3.4 billion*, 2026. Accessed: Jun. 27, 2026. [Online]. Available: https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2026/.

[15] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[16] T. Jiao, Z. Xu, M. Qi, S. Wen, Y. Xiang, and G. Nan, "A survey of Ethereum smart contract security: Attacks and detection," *Distrib. Ledger Technol.*, vol. 3, no. 3, Sep. 2024. DOI: 10.1145/3643895.

[17] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer Cham, 2021, ISBN: 978-3-030-72272-2. DOI: 10.1007/978-3-030-72274-6.

[18] K.-T. Cheng and A. S. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, no. 1, pp. 57–79, Jan. 1996, ISSN: 1084-4309. DOI: 10.1145/225871.225880.

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

[20] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," in *Analysis and Optimization of Systems*, A. Bensoussan and J. L. Lions, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 475–498, ISBN: 978-3-540-39010-7.

[21] S. Mohajerani, W. Ahrendt, and M. Fabian, "Modeling and security verification of state-based smart contracts," *IFAC-PapersOnLine*, vol. 55, no. 28, pp. 356–362, 2022, 16th IFAC Workshop on Discrete Event Systems WODES 2022, ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2022.10.366.

[22] S. Rouhani and R. Deters, "Blockchain based access control systems: State of the art and challenges," in *IEEE/WIC/ACM International Conference on Web Intelligence*, ser. WI '19, Thessaloniki, Greece: Association for Computing Machinery, 2019, pp. 423–428, ISBN: 9781450369343. DOI: 10.1145/3350546.3352561. [Online]. Available: https://doi.org/10.1145/3350546.3352561.

[23] K. Li, R. Gu, J. Xu, Z. Chen, S. Wu, Y. Zhou, M. Zhang, X. Luo, Y. Tang, Y. Li, X. Zhang, and Y. Wang, "Security analysis and formal verification on blockchain and its applications," *Foundations and Trends in Privacy and Security*, vol. 8, no. 1, pp. 1–121, Jun. 2025, ISSN: 2474-1558. DOI: 10.1561/3300000044. [Online]. Available: https://doi.org/10.1561/3300000044.

[24] H. Guo and X. Yu, "A survey on blockchain technology and its security," *Blockchain: Research and Applications*, vol. 3, no. 2, p. 100 067, 2022, ISSN: 2096-7209. DOI: https://doi.org/10.1016/j.bcra.2022.100067. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2096720922000070.

[25] S.-W. L. Liu Ye Li Yi and Z. Rong, "Towards automated verification of smart contract fairness," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 666–677. DOI: 10.1109/CSCI46756.2018.00112.

[26] Ethereum Foundation, *Solidity documentation*, 2026. Accessed: Jun. 27, 2026. [Online]. Available: https://docs.soliditylang.org/.

[27] Vyper, *Vyper*, 2026. [Online]. Available: https://docs.vyperlang.org/en/stable/.

[28] G. Bansal, *Developing smart contracts using Solidity: Escrow decoded*, https://www.linkedin.com/pulse/developing-smart-contracts-using-solidity-escrow-decoded-bansal/, [Online; accessed 19-November-2024], 2018.

[29] Smart Contract Weakness Classification Registry, *SWC-113: DoS with failed call*, 2020. Accessed: Jun. 27, 2026. [Online]. Available: https://swcregistry.io/docs/SWC-113/.

[30] Wikipedia, *Abstract syntax tree*, https://en.wikipedia.org/wiki/Abstract_syntax_tree, [Online: accessed 23-November-2024], 2024.

[31] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," vol. 77, no. 1, pp. 81–98, Jan. 1989. DOI: 10.1109/5.21072.

[32] S. Mohajerani, R. Malik, and M. Fabian, "A framework for compositional nonblocking verification of extended finite-state machines," *Journal of Discrete Event Dynamic Systems*, vol. 26, no. 1, pp. 33–84, 2016. DOI: 10.1007/s10626-015-0217-y.

[33] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[34] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989. DOI: 10.1109/5.21072.

Part II

Papers

PAPER A

Smart Contract Denial-of-Service Analysis Using Non-Blocking Verification

Nishant Parekh, Wolfgang Ahrendt, Martin Fabian

Published in Discrete Event Dynamic Systems, vol. 35, pp. 355–387, Dec 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.

1 Introduction

Smart contracts are micro-services executing within a blockchain ecosystem. Their main purpose is to enforce agreements among mutually distrusting par-

ties 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 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 [1] has permanently made over $1 million inaccessible to its owners.

Multiple vulnerabilities of smart contracts are known [2, 3], and exploits of these vulnerabilities, called *attacks*, have caused loss of huge funds [4, 5]. 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, [2].

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* [6], 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) [7–9]. These are similar to ordinary finite-state machines, but add bounded discrete variables together with guards and actions defined over these variables. [10] 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.

[10] show how state based smart contracts can be modelled as EFSMs, and how methods from *supervisory control theory* [11] can be applied to verify non-blocking behaviour. [10] 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, [10] 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 [12], presents an approach

to *automatically* convert, from their source code, smart contracts to 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 [10] 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 [2–4].

Related work on modelling smart contracts and their verification is presented by [13] that describe modelling smart contracts as EFSMs and verifying them using the nuXmv [14] model checker. However, in the above-mentioned

work, EFSM models of smart contracts are generated manually. [15] 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 [16] is presented in [17]. [18] and [19] explore strategies aimed at synthesis of secure smart contracts from EFSMs that fulfill requirement specifications. Another approach presented by [20] investigate using interface automata for verification purposes.

Another work, presented by [5], propose a tool, Solvent, which translates a smart contract and its set of user-defined liquidity properties into SMT constraints [21] that can be analyzed by SMT solvers such as Z3 [22] or cvc5 [23].

The work of [5] 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 [5] 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 [2] as SWC-113, *DoS with Failed Call*. This weakness is also recognized as *1.3.2 Improper Exception Handling of External Calls* by [3] who present a hierarchical taxonomy of smart contract vulnerabilities. In an empirical evaluation of smart contract implementations, [24] note that all implementations, in three different smart contract programming languages, of their Contract 3, "King of the currency" [25] were vulnerable to DoS with unexpected revert attacks.

Finally, we do not concur with the view, put forward by [5], 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 [26], 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

[1] called 'state variables' in Ethereum terminology

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.

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, *modi-*

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

fiers. 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:

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

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 [27]. 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 c`.f(...)` (where c is a contract address and `f` is a function of c) reverts if execution in c reverts. In contrast, the low-level call statement c`.call(...)` does not revert if execution in c reverts, but returns `false` in that case. For payments, there is a similar distinction. The statement a`.transfer(`v`)` 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 a`.send(`v`)` 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 c`.f(...)` or a`.transfer(`v`)`,

and *can* lead to 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) [9, 28] 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 *true* (**T**) and *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 $\mathrm{dom}(v)$, and has an initial value $v^\circ \in \mathrm{dom}(v)$. Let $V = \{v_0, \ldots, v_n\}$ be the set of variables with domain $\mathrm{dom}(V) = \mathrm{dom}(v_0) \times \cdots \times \mathrm{dom}(v_n)$. An element of $\mathrm{dom}(V)$ is called a *valuation* and is denoted by $\hat{v} = \langle \hat{v}_0, \ldots, \hat{v}_n \rangle$ with $\hat{v}_i \in \mathrm{dom}(v_i)$, 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 $\mathrm{dom}(V') = \mathrm{dom}(V)$, 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 Π_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., $p\colon \mathrm{dom}(V) \times \mathrm{dom}(V') \to \{\mathbf{T}, \mathbf{F}\}$.

Definition 1: *An* extended finite-state machine (EFSM) *is a tuple* $E = \langle \Sigma, S, S^\circ, \to, S^\omega \rangle$*, where* Σ *is a set of events;* S *is a finite set of* locations*;*

[3] Using the non-reverting statements `call` and `send` 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 `call` and `send` with a `require` on the returned boolean, which brings us back to the problem of a local revert being in the hand of an external party.

$\rightarrow \subseteq S \times \Sigma \times \Pi_V \times S$ *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 $\mathbf{T}$, the system can transit from location q to location q' on the occurrence of the event σ. When the transition occurs the variables in $\text{vars}'(p)$ are updated while the variables not contained in $\text{vars}'(p)$ 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 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`.

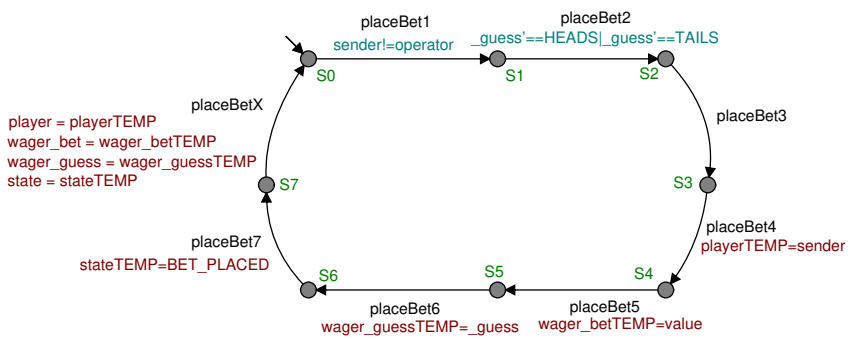


Figure 1: EFSM model of the Casino `placeBet` function (see Fig 2, lines 18–27, and Section 5.2).

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* [29], 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* $E_1 = \langle \Sigma_1, S_1, S_1^\circ, \to_1, S_1^\omega \rangle$ *and* $E_2 = \langle \Sigma_2, S_2, S_2^\circ, \to_2, S_2^\omega \rangle$*, the* synchronous composition *of* E_1 *and* E_2 *is* $E_1 \parallel E_2 = \langle \Sigma_1 \cup \Sigma_2, S_1 \times S_2, \to, S_1^\circ \times S_2^\circ, S_1^\omega \times S_2^\omega \rangle$*, where:*

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

Note that synchronous composition is associative.

For example, the event `placeBet1` of Fig 1 synchronizes with the same label in the EFSMs modeling the `inState` 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 `sender` is different from the `operator` (as required by Fig 1), and `state==GAME_AVAILABLE` (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 $E_1 \parallel \cdots \parallel E_n$.

Non-blocking of an EFSM system, is defined on the *flattened* system [30], 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 [30].

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

- $Q_U = Q \times \mathrm{dom}(V)$*;*
- $(x, \hat{v}) \xrightarrow{\sigma}_U (y, \hat{w})$ *if* E *contains a transition* $x \xrightarrow{\sigma:p} y$ *such that* $p(\hat{v}, \hat{w}) = \mathbf{T}$*;*

- $Q_U^\circ = Q^\circ \times \{v^\circ\}$;
- $Q_U^\omega = Q^\omega \times \mathrm{dom}(V)$.

$U(E)$ 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 $U(\mathcal{E}) = U(E_1 \parallel \ldots \parallel E_n)$. 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 [31].

3.3 Supremica

Supremica [31] 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 [32] 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* [33] 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* [30, 32] 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 [34], 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: `IDLE`, `GAME_AVAILABLE`, and `BET_PLACED`, see line 3, defined by the `enum` type `State`.

[4] A variant of the contract `OneAuction` in [27]

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

```
1  contract Casino {
2
3    enum State {IDLE, GAME_AVAILABLE, BET_PLACED}
4    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 }
```

Figure 2: Solidity code for Casino (some details are omitted).

Based on the modifier `inState(s)`, in the `IDLE` location the operator may create a game by invoking the `createGame` function (line 13). To ensure a fair betting, the Casino must place its bet at the time of game creation. Thus, when calling `createGame`, `hashedNumber` is assigned a value (line 15) to later decide the game outcome. After creating a new game, the state changes to `GAME_AVAILABLE` (line 16) where a game is now available. In this state, the player can call `placeBet` to place a bet, up to the size of the pot, on HEADS or TAILS (lines 20-25). This then changes the state of the contract to `BET_PLACED` (line 27).

Next, the operator may by `decideBet` 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, `addToPot` (line 51). Also, the operator may remove money from the pot, `removeFromPot` (line 55), but only if the player has not placed a bet, that is, if the casino is not in the state `BET_PLACED`. This is ensured by the modifier `noActiveBet`.

4.2 The Auction smart contract

```
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 }
```

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

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 `placeBid` (lines 11–24) and `closeAuction` (lines 26–31) can only be called when the auction is open (see lines 13 and 28, respectively). The `currentBid` is initialized to 0 (line 3), and the `auctionOwner` is set to be the one constructing the auction (line 8). A bidder can place their bid by calling the `placeBid` function (line 11). If the bid placed is higher than the current bid (line 14), `currentBidder` is updated to the caller of `placeBid` and `currentBid` 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 `oldBidder.transfer(oldBid)` (line 22). The auction owner can close the auction by calling function `closeAuction` (line 26), which

sets `auctionOpen` 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 [35]), which is obtained in JSON [36] 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 [10]. One difference is that the automatically converted model describes only the plant behavior, it includes no specification, whereas the model given by [10] has the specification embedded in a rather complicated way. Another difference is that the `State` variable is in the manual model represented by a specific EFSM, which embeds the control of the other functions, thus making the modifiers redundant. In the automatically converted model, `State` is represented by an EFSM variable `state`, much like in the Solidity code, and the modifiers are thus explicitly modeled.

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 `pot` variable cannot be modelled since if `pot` is modelled as an (upper) bounded variable, then a trivial blocking trace calls `addToPot` enough times for `pot` to reach its upper bound, plus once more, and then the system deadlocks. The `pot` variable is automatically ignored by adding it to an

ignore list, so the converter does not generate any `pot` variable in the EFSM model, nor any transitions for statements involving `pot` (lines 21, 44, 48, 53, 57).

Also the Auction contract has a problematic variable, `currentBid`, that causes trivial blocking when the bidding has reached is upper bound, `MAX_BID`. The `require` 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 `MAX_BID`, and then the `placeBid` 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 `currentBid`, as was done with `pot`, so the `require` 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 `placeBet` function is shown in Fig 1; this models lines 18–27 of Fig 2. The initial event `placeBet1` labels the transition from the initial location `S0`, and the `require` clause at line 20, ensuring that `operator` is not the caller of `placeBet` is added as a guard. The handling of the modifiers and the `require` clauses, lines 19–21 are described in Section 5.3, below.

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

The `placeBet` function is called with a parameter `_guess` of type `Coin`, 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 `_guess` is unknown for the `placeBet` call, a non-deterministic assignment is made to the variable `guess`, see the guard on the transition labeled `placeBet2` of Fig 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 labeling a transition from the self-looped location. This is illustrated in Fig 4, where the `operatorWins` and `playerWins` 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 `playerWins` (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 `playerWins`, and then waits for `playerWins` to return.

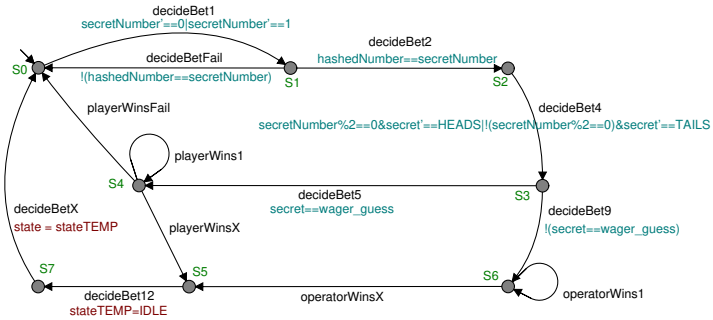


Figure 4: EFSM model of the `decideBet` function, Fig 2, lines 29–39.

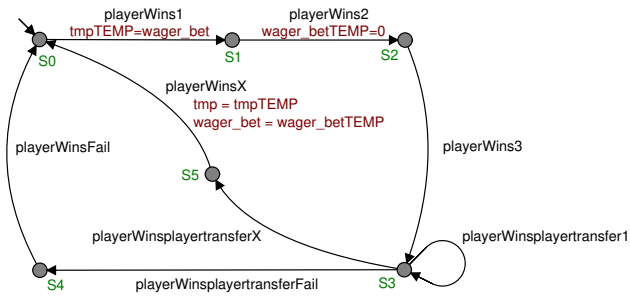


Figure 5: EFSM model of the `playerWins` function, Fig 2, lines 41–45.

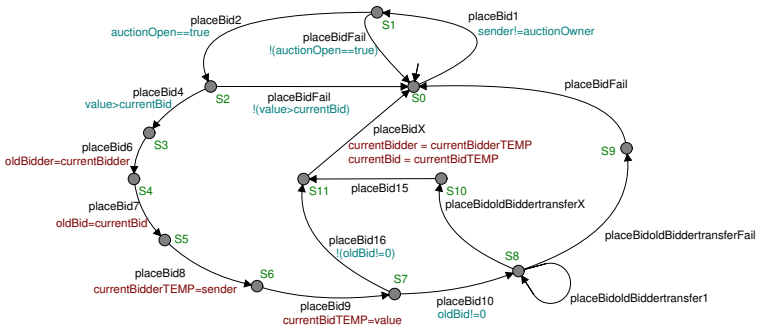


Figure 6: EFSM model of the `placeBid` function, Fig 3, lines 11–24.

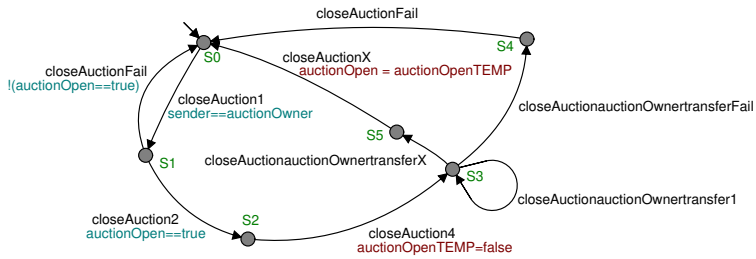


Figure 7: EFSM model of the `closeAuction` function, Fig 3, lines 26–31.

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 `wager` struct variable, is described above. The external hashing function `keccak256` (line 32) is not modelled at all, as its internal workings are not known. This is handled by adding `keccak256` to the abovementioned *ignore list*, so that the call `keccak256(secretNumber)` is automatically converted into simply `secretNumber`.

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 `require` clauses relate to. For instance, the

`byOperator` modifier relates to the `createGame`, `addToPot`, `removeFromPot`, and `decideBet` functions, and so the self-looped transition is labeled by their initial events, see Fig 8, left. Modifier `inState(s)`, 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 `inState(s)` has multiple self-loop transitions instead of one.

Figure 8: EFSM models of (left) the `byOperator` modifier, and (right) the `inState(s)` modifier

Along with modifiers, it is common practice to use `require` statements within the function for additional checks. Modeling of `require` is done by adding the Boolean expression as a guard on the transition. For instance, `require` 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 `require` 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, `require` statement on line 32 containing parameter `secretNumber`, gets converted to the guard on transition from node `S1` to `S2` in Fig 4, after `secretNumber` has been non-deterministically assigned. Case where the condition specified for `require` 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 `msg.sender` variable, and the behavior of `transfer`. As this behavior is not possible to extract from the code, these models are manually predefined, but in a generic way.

Assign Sender and Value

Within the Solidity framework, contracts interact with each other by calling public functions. With each such call follows a data packet `msg`, which includes among other things a reference (address), `msg.sender`, and the amount of Wei sent with the message, `msg.value`, to the contract that called the function. The assignment to `msg.sender` and `msg.value` is handled by the framework, outside of the Solidity code. In the Casino contract there are two participants, `player` and `operator`, and the behavior of the public functions depends on which of the participants that called it, so the EFSM model must include a model for the assignment of `msg.sender` and `msg.value`. This is done by the EFSM of Fig 9.

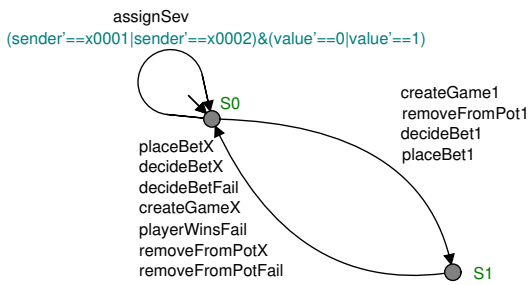


Figure 9: EFSM model of the assignment of `sender` for Casino contract.

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 `createGame` cannot be called by the player, only traces that start with the sender being the operator are of interest for the non-blocking verification.

Out from `S0` is also a transition to location `S1`, labeled with the initial event

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

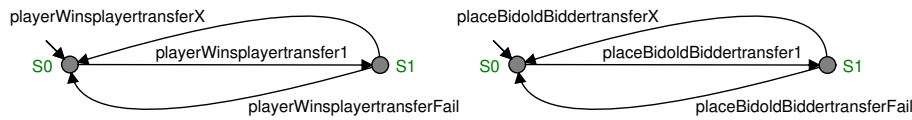


Figure 10: EFSM models of `transfer` to the player (left) and to the bidder (right).

Modelling transfer

When a `transfer` 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 `transfer` to an address is generated by concatenating the name of the function in which the `transfer` 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 `transfer` to `player`. The event name, `playerWinsplayertransfer1`, is generated by concatenating the function name `playerWins` with the address `player` and, `transfer1`, which makes up the initial transition of that particular `transfer` 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 `msg.sender`, 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 `transfer` is modelled in a calling function by a transition from the location where `transfer` 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 `playerWins` EFSM (Fig 5), `transfer` to `player` (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 `playerWins` 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 unsuccessfully, 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 `placeBidX` transition, which represents successful completion of the function.

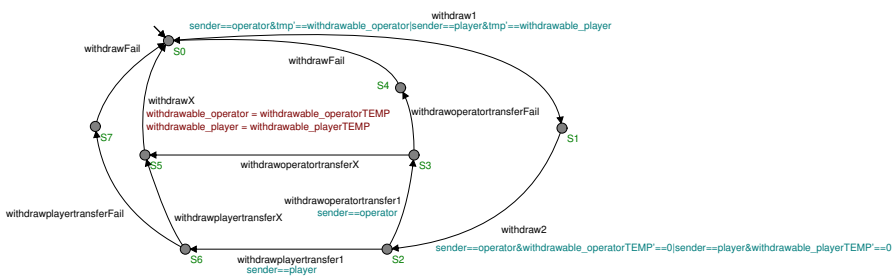


Figure 11: EFSM model of the `withdraw` function in the corrected Casino contract.

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, `decideBet` and `placeBet`, have 8 locations, while the biggest one for Auction is `placeBid` 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 `addToPot` of Casino is a single location with a single self-loop labelled by `addToPot1`. This is because `addToPot` only changes the value of the `pot` 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, `currentBid`, `oldBid`, and their shadow variables, plus `msg.value` have larger domains to make bidding at all possible. The `state` variable of Casino has the same symbolic domain as in the contract `IDLE`, `GAME_AVAILABLE`, and `BET_PLACED`. 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]The models together with the code for the automatic conversion are available from `https://github.com/nishantparekh01/Solidity_to_EFSM/`

6 Non-blocking Verification

Although the generated plant models account for failing transfers, the issue investigated in this paper, DoS by rejecting transfer [2], 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. [11] 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, [6]) 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 [30] implemented by SUPREMICA.

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 it-s/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 [2]), exhibits the malicious behaviour that once rejecting a transfer the attacker always rejects a transfer. Fig 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 `transfer` models of Fig 10. The attacker models stay in their respective initial state until a rejected `transfer` occurs, whereafter only rejected `transfer`s are enabled.

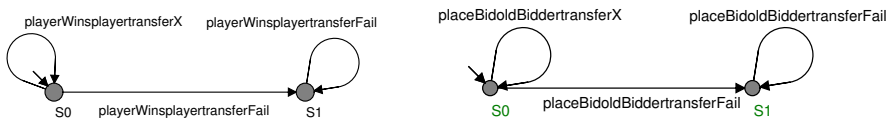


Figure 12: The attacker model for the Casino, left, and the Auction, right.

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.

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 Σ, 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`.

Figure 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.

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 Σ^* 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 Σ, 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 `placeBidX`. 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 `closeAuctionX` to occur. But since after closing the Auction, it is not possible to (meaningfully) interact with it again, it cannot be required that `closeAuctionX` occurs an unbounded number of times, which is why the right progress specification of Fig 13 is used with w equal to `closeAuctionX`.

6.2 Counterexamples

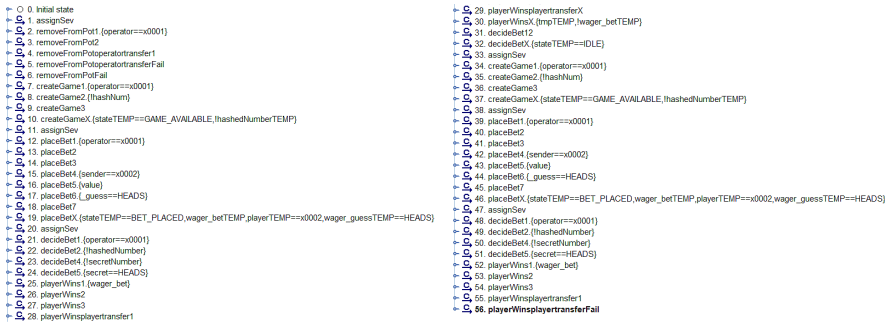


Figure 14: The 56-event long blocking trace of the Casino contract with progress specification for `removeFromPotX`.

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 [30] and [32].

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 [10], among other things due to the more detailed model with the shadow variables that are not present in the model of [10]. 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 `removeFromPot` 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 `removeFromPot` 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 `IDLE` or `GAME_AVAILABLE`. Looking at the counterexample (Fig 14) and relating it to the code (Fig 2) maps the blocking to line 45. If `player.transfer(wager.bet*2)` fails, line 39 of `decideBet` will not be executed and thus `state` will not be assigned the value `IDLE`, which prevents `removeFromPot` to successfully complete. It also prevents `operator` from re-initializing the game (line 14). Though `decideBet` can be called again, when a malicious player refuses the `transfer` on each call, the contract will never progress to reach its `IDLE` state. Thus, the funds of the Casino can never be retrieved, so that its liquidity is compromised.

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 `closeAuctionX` as w is added to the model. This model blocks. Investigating the counterexample (Fig 15) shows that the blocking happens because `closeAuctionX` 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.

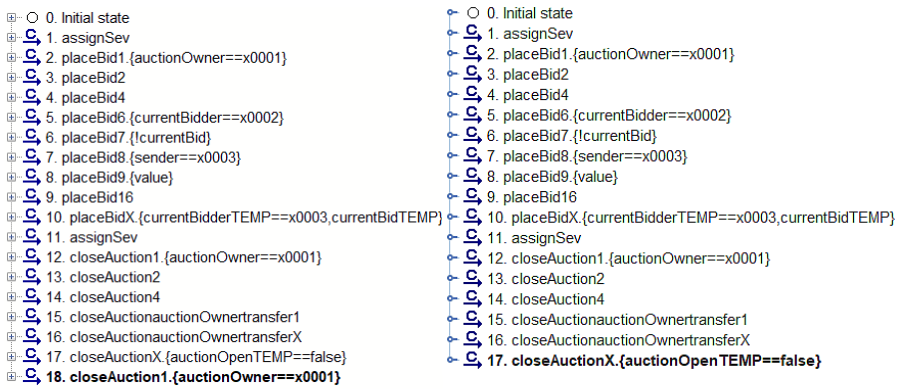


Figure 15: Counterexamples for Auction when verifying `closeAuctionX` (left) and `placeBidX` (right) with the progress specification of Fig 13, left.

The progress specification on the right in Fig 13, with $w =$ `closeAuctionX`, models that whatever happens after `closeAuctionX` 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 `closeAuctionX` can be removed. In fact, the progress specification for `closeAuctionX` *must* be removed to verify other propertises, 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 specification is 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 $w =$ `placeBidX` 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 `auctionOpen == false` reverting `placeBid` 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 `placeBidX`. If this is non-blocking, then it is known that the only issue is with `closeAuctionX` so that the progress specification holds unless `closeAuctionX` is allowed. So we add a `disableCloseAuctionX` specification, which is an automaton with a single marked and initial state, and a blocked events list containing only `closeAuctionX`. Of course, this requires the `closeAuctionX` progress specification to be removed, otherwise it would conflict with the `disableCloseAuctionX` 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 counterexample trace to (Fig 16, left). This trace immediately calls `closeAuction` which then executes until state `S5` (Fig 7) from where only `closeAuctionX` is enabled, but which the `disableCloseAuctionX` specification prevents.

So the `disableCloseAuctionX` specification is replaced by the `disableCloseAuction1` specification, which globally disables `closeAuction1`, the initial event of the `closeAuction` 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 `disableCloseAuction1` 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 `closeAuction`, but due to another issue, here called "the `MAX_BID` issue".

EFSM variables, just as variables in Solidity, are upper bounded, in SUPREMICA by 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 `MAX_BID`. 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 `placeBid` reverts. So once the current bid has reached `MAX_BID`, no further bids can be placed. The counterexample in Fig 16, right, shows

[7]Solidity does not support floating point numbers.

that when the placeBid completes successfully with a MAX_BID, set to 4 in this case (see event 10, currentBidTEMP==4), then on the next call to placeBid, blocking occurs since no bid higher than MAX_BID can be given.

Unfortunately, getting around the MAX_BID issue is not as easy as was the case with closeAuction, 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 placeBid, Fig 6, modelled as guards on the two transitions out from location S2, one guarded by value>currentBid to S3, and the other by !(value>currentBid) to S0.

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 MAX_BID 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 disableCloseAuction1 effectively removes the closeAuction EFSM, and this would allow to verify whether it is actually the case that a bid can always be placed or not.

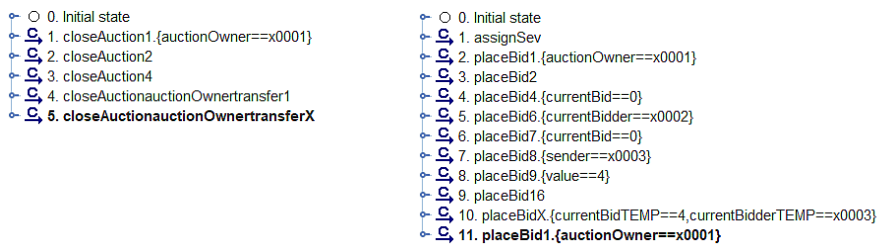


Figure 16: Counterexample for Auction when verifying placeBidX with disableCloseAuctionX (left) and with disableCloseAuction1 (right).

This EFSM system, with the attacker model, the progress specification over placeBid, the disableCloseAuction1, and the Auction plant with the two guards checking currentBid removed, verifies to be blocking. The 30-step counterexample is shown in Fig 17. As can be seen, this blocks when a transfer to the

`oldBidder` occurs at Step 30. From there on, the malicious bidder rejects all transfers, and so the system can only cycle around to try the `transfer` again and again, to always be rejected by the receiver.

Figure 17: 30-step counterexample for Auction when verifying `placeBidX` with `disableCloseAuction1` and the two guards checking `currentBid` removed.

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 `transfer` 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 `transfer` occurs since the `oldBid` is still zero. But the next time a bid is placed, since `oldBid` is different from zero, a `transfer` 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 Auction is not. Still, both contracts suffer from the same attack scenario, and the correction is consequently the same in both contracts.

The corrected Casino smart contract

To prevent the contract from entering a blocking state, a correction described by [10] 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 `player` is updated in the `playerWins` function, line 64, following which `player` 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 `player`.

```
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);}
```

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

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.

The corrected Auction smart contract

The `closeAuction` and `MAX_BID` 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 `MAX_BID`, 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 `withdraw` solution of Casino. Instead of the contract issuing a `transfer` 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 [37], 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.

Funding declaration

This work was funded by Chalmers University of Technology.

Declaration of competing interests

The authors declare that they have no competing financial interests or personal relationships that could have influenced the work reported in this paper.

References

[1] CoinGeek, *Over $1 million permanently locked in DeFi smart contract*, `https://coingeek.com/over-1-million-permanently-locked-in-defi-smart-contract/`, [Online; accessed 19-November-2024], 2020.

[2] SWC Registry, *Smart contract weakness classification (SWC)*, `https://swcregistry.io/docs/SWC-113/`, [Online; accessed 21-November-2024], 2020.

[3] F. Richter Vidal, N. Ivaki, and N. Laranjeiro, "OpenSCV: An open hierarchical taxonomy for smart contract vulnerabilities," *Empirical Software Engineering*, vol. 29, Jun. 2024. DOI: 10.1007/s10664-024-10446-8.

[4] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186, ISBN: 978-3-662-54455-6. DOI: 10.1007/978-3-662-54455-6_8.

[5] M. Bartoletti, A. Ferrando, E. Lipparini, and V. Malvone, "Solvent: Liquidity verification of smart contracts," in *Integrated Formal Methods*, N. Kosmatov and L. Kovács, Eds., Cham: Springer Nature Switzerland, 2024, pp. 256–266, ISBN: 978-3-031-76554-4. DOI: 10.1007/978-3-031-76554-4_14.

[6] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[7] K.-T. Cheng and A. S. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, no. 1, pp. 57–79, Jan. 1996, ISSN: 1084-4309. DOI: 10.1145/225871.225880.

[8] Y.-L. Chen and F. Lin, "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)*, 2000, pp. 941–946. DOI: 10.1109/CCA.2000.897591.

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

[10] S. Mohajerani, W. Ahrendt, and M. Fabian, "Modeling and security verification of state-based smart contracts," *IFAC-PapersOnLine*, vol. 55, no. 28, pp. 356–362, 2022, 16th IFAC Workshop on Discrete Event Systems WODES 2022, ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2022.10.366.

[11] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," vol. 77, no. 1, pp. 81–98, Jan. 1989. DOI: 10.1109/5.21072.

[12] N. Parekh, W. Ahrendt, and M. Fabian, "Automatic conversion of smart contracts for non-blocking verification," *IFAC-PapersOnLine*, vol. 58, no. 1, pp. 282–287, 2024, 17th IFAC Workshop on Discrete Event Systems WODES 2024, ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2024.07.048.

[13] R. Fekih, M. Lahami, M. Jmaiel, A. Ali, and P. Genestier, "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*, Jun. 2022, pp. 83–88. DOI: 10.1109/COMPSAC54236.2022.00020.

[14] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., Cham: Springer International Publishing, 2014, pp. 334–342, ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_22.

[15] J. Godoy, J. P. Galeotti, D. Garbervetsky, and S. Uchitel, "Predicate abstractions for smart contract validation," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '22, Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 289–299, ISBN: 9781450394666. DOI: 10.1145/3550355.3552462.

[16] G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997. DOI: 10.1109/32.588521.

[17] X. Bai, Z. Cheng, Z. Duan, and K. Hu, "Formal modeling and verification of smart contracts," in *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, ser. ICSCA '18, Kuantan, Malaysia: Association for Computing Machinery, 2018, pp. 322–326, ISBN: 9781450354141. DOI: 10.1145/3185089.3185138.

[18] D. Suvorov and V. Ulyantsev, *Smart contract design meets state machine synthesis: Case studies*, 2019. [Online]. Available: https://arxiv.org/abs/1906.02906.

[19] A. Mavridou and A. Laszka, "Designing secure Ethereum smart contracts: A finite state machine based approach," in *Financial Cryptography and Data Security*, S. Meiklejohn and K. Sako, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 523–540, ISBN: 978-3-662-58387-6.

[20] G. Madl, L. Bathen, G. Flores, and D. Jadav, "Formal verification of smart contracts using interface automata," in *2019 IEEE International Conference on Blockchain (Blockchain)*, 2019, pp. 556–563. DOI: 10.1109/Blockchain.2019.00081.

[21] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, vol. 185, IOS Press, 2009, pp. 825–885. DOI: 10.3233/978-1-58603-929-5-825. [Online]. Available: https://ebooks.iospress.nl/publication/5011.

[22] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[23] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "Cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds., Cham: Springer International Publishing, 2022, pp. 415–442, ISBN: 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_24.

[24] R. M. Parizi, Amritraj, and A. Dehghantanha, "Smart contract programming languages on blockchains: An empirical evaluation of usability and security," in *Blockchain – ICBC 2018*, S. Chen, H. Wang, and L.-J. Zhang, Eds., Cham: Springer International Publishing, 2018, pp. 75–91, ISBN: 978-3-319-94478-4. DOI: 10.1007/978-3-319-94478-4_6.

[25] G. Konstantopoulos, *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-

solidity-vulnerabilities-and-how-to-avoid-them-part-2-730db0aa4834, [Online: accessed 24-November-2024], 2018.

[26] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2023. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[27] W. Ahrendt and R. Bubel, "Functional verification of smart contracts via strong data integrity," in *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2020, pp. 9–24, ISBN: 978-3-030-61467-6. DOI: 10.1007/978-3-030-61467-6_2.

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

[29] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[30] S. Mohajerani, R. Malik, and M. Fabian, "A framework for compositional nonblocking verification of extended finite-state machines," *Journal of Discrete Event Dynamic Systems*, vol. 26, no. 1, pp. 33–84, 2016. DOI: 10.1007/s10626-015-0217-y.

[31] K. Akesson, M. Fabian, H. Flordal, and R. Malik, "Supremica – an integrated environment for verification, synthesis and simulation of discrete event systems," in *2006 8th International Workshop on Discrete Event Systems*, IEEE, 2006, pp. 384–385. DOI: 10.1109/WODES.2006.382401.

[32] R. Malik, S. Mohajerani, and M. Fabian, "A survey on compositional algorithms for verification and synthesis in supervisory control," *Journal of Discrete Event Dynamic Systems*, vol. 33, no. 3, pp. 279–340, Sep. 2023, ISSN: 1573-7594. DOI: 10.1007/s10626-023-00378-8.

[33] S. Mohajerani, R. Malik, and M. Fabian, "Partial unfolding for compositional nonblocking verification of extended finite-state machines," Chalmers University of Technology, Göteborg, Sweden; also The University of Waikato, Hamilton, New Zealand, Tech. Rep., 2013. DOI: https://hdl.handle.net/10289/7140. [Online]. Available: https://research.chalmers.se/publication/172205.

[34] VerifyThis, *VerifyThis long-term challenge*, `https://verifythis.github.io/02casino/`, [Online; accessed 21-November-2024], 2021.

[35] Wikipedia, *Abstract syntax tree*, `https://en.wikipedia.org/wiki/Abstract_syntax_tree`, [Online: accessed 23-November-2024], 2024.

[36] ISO/IEC 21778, *Information technology – the JSON data interchange syntax*, `https://www.iso.org/standard/71616.html`, [Online: accessed 23-November-2024], 2017.

[37] S. Matsui and S. Lafortune, "Synthesis of winning attacks on communication protocols using supervisory control theory: Two case studies," *Discrete Event Dynamic Systems*, vol. 32, no. 4, pp. 573–610, Dec. 2022, ISSN: 1573-7594. DOI: `10.1007/s10626-022-00369-1`. [Online]. Available: `https://doi.org/10.1007/s10626-022-00369-1`.

PAPER B

On Bias in User-Centric Discrete-Event Systems

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

Proc. 18th Workshop on Discrete Event Systems, Eindhoven, Netherlands, May, 2026

Abstract

User-centric discrete-event system are discrete-event systems with which multiple users interact, each through their own distinct events. Each user has specific goals that they want to achieve, represented by specific states of the discrete-event systems. In these kinds of systems, *bias* may be present, positive bias that allows some users to always achieve their goals irrespective of the other users, or negative bias that prevents some users from achieving their goals irrespective of what they do. This work shows how the existence of a *strategy*, a special type of supervisor computed according to the Ramadge & Wonham supervisory control theory, can with carefully chosen controllable events and marked states reveal bias, positive and negative, in a user-centric discrete-event system. This can then be a guiding factor in deciding whether to interact or not with a given user-centric discrete-event system.

1 Introduction

Discrete-Event Systems (DES) is a useful formalism for modeling systems that exhibit a discrete state-transition behavior. This includes many man-made systems, such as, infrastructural systems [1], manufacturing systems [2], automotive control systems [3]. Even many types of computer programs can be usefully modeled as DES, like smart contracts [4, 5].

The Supervisory Control Theory (SCT) [6] is a formal approach to generate correct-by-construction controllers for DES. Given a DES modeling a system to be controlled, the *plant*, and a DES modeling the desired behavior, the *specification*, a controller, called a *supervisor*, can be automatically constructed, *synthesized*, that guarantees that the controlled system of plant and supervisor exhibits the realizable parts of the desired behavior. Such a supervisor is required to be *controllable*, meaning that it does not try to affect events that it cannot control, and *non-blocking*, meaning that it guarantees that it is always possible to reach some state of particular interest.

A user-centric discrete-event system (UCDES) is a specific type of DES, where the interaction with multiple *users* is a critical part of the behavior

of the system. Each user has a set of *interaction events* with which that user interacts with the UCDES, and after having issued (by whatever means available) such an interaction event, the UCDES may perform a sequence of *internal events* whereafter new interaction events can be issued. Typical examples of such UCDES are systems where user input directly influences the sequence of events that change the system's state, such as human-machine interaction [7], smart contracts [4], multi-user services and multi-player games.

In many cases, users have specific *goals* when interacting with a UCDES. In online games this is obvious, the user wants to win and collect some reward. In other UCDES, there may be more subtle goals where users just want to interact with the UCDES to achieve some personal benefit, maybe not obvious to others.

However, UCDES may exhibit *bias* towards or against specific users or sets of users. Such bias means that certain users are at an advantage (*positive bias*) with interacting with the UCDES in that there is a guaranteed way for them to always reach their goals, or users are at a disadvantage (*negative bias*) in that there is a way for other users to prevent them from reaching their goals. Bias might manifest due to bugs in improper implementation, or due to deliberate insidious implementation, but in any case it would be useful for any user that interacts with a UCDES to know beforehand whether that UCDES is biased for deciding whether to interact with it or not; especially so for negative bias.

Should bias exist, as shown in this paper, this can berevealed by the existence of a *strategy* to enforce that bias. Such a strategy is a special type of supervisor [6], which in addition to being controllable and non-blocking, also has the property of having no cycles of non-marked states. Due to being controllable, users with controllable interaction events can always steer the system to remain within the strategy, and due to being non-blocking with these user's goal states marked, some goal state is always reachable. The absence of cycles of non-marked states then guarantees that the system can always be controlled to actually reach one of those goal states, which enforces the bias.

Historically, *bias* in DES has concerned discrete event simulation where bias refers to a statistical measure related to stabilizing behavior after atypical startup conditions or perturbations [8]. This is not the type of bias discussed in this work; here is developed a framework for assessing in UCDES whether

there exists behavior that can be exploited by user interactions for the benefit or the detriment of (other) users. To the best of our knowledge, this has not been researched earlier.

This paper is a companion paper to [9], defining the notions of bias and strategy more rigorously than what was possible due to space restrictions in [9]. However, while [9] is specific to Ehereum smart contracts, the work described in this paper is more general, and thereby a contribution of its own.

2 Background

This section gives a terse overview of the background necessary to appreciate the rest of the paper.

2.1 Finite-State Machines

Finite-State Machines (FSM) [10] are a modeling formalism suitable for DES. The main elements are *states*, *events*, and *transitions*, where states represent situations under which certain conditions hold, and events represent incidents that cause the transitioning from one state to another. Formally:

Definition 1: *Let* $G = \langle Q_G, \Sigma_G, \delta_G, i_G \rangle$ *be a (deterministic) FSM where:*

Q_G *is the set of states;*

Σ_G *is the set of event labels, the* alphabet*;*

δ_G *is the transition function; and*

i_G *is the initial state,* $i_G \in Q_G$.

The transition function $\delta_G \subseteq Q_G \times \Sigma_G \times Q_G$, describes the possible evolution of the FSM from state to state on the occurrences of events. The transition function can be extended to finite sequences of events, $s \in \Sigma_G^*$ in the standard way [10], $\delta_G \subseteq Q \times \Sigma_G^* \times Q$. As the FSM transits from state to state in accordance with sequences of events, these sequences make up a *language*, $\mathcal{L}(G) = \{s \in \Sigma_G^* \mid \delta_G(i_G, s) \text{ is defined}\}$.

An FSM $S = \langle Q_S, \Sigma_S, \delta_S, i_S \rangle$ is a *sub-automaton* of G, if it holds that $Q_S \subseteq Q_G$, $\Sigma_S = \Sigma_G$, $\delta_S \subseteq \delta_G$ and, $i_S = i_G$. Note that the alphabet and the initial state of S are equal to those of G.

FSMs can interact through *synchronous composition* [10], where shared

events are transited on simultaneously, or not at all. That is, for two FSMs, G_1 and G_2, their synchronous composition $G_1||\,G_2$ is defined as:

$$G_1||\,G_2 = \langle Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_{G_1||\,G_2}\langle i_1, i_2\rangle\rangle,$$

where for $\langle q_i, q_2\rangle \in Q_1 \times Q_2$ and $\sigma \in \Sigma_1 \cup \Sigma_2$:

$$\delta_{\sigma,G_1||G_2}(\langle q_1, q_2\rangle, \sigma) = \begin{cases} \langle \delta_1(q_1,\sigma), \delta_2(q_2,\sigma)\rangle & \sigma \in \Sigma_1 \cap \Sigma_2 \\ \langle \delta_1(q_1,\sigma), q_2\rangle & \sigma \in \Sigma_1 \setminus \Sigma_2 \\ \langle q_1, \delta_2(q_2,\sigma)\rangle & \sigma \in \Sigma_2 \setminus \Sigma_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that if G_1 or G_2 does not define an event from a certain state but has it in its alphabet, that event and any transition on it is *disabled* at that state.

A *cycle* in an FSM is a sequence of consecutive transitions, $\langle\langle q_1, \sigma_1, q_2\rangle, \langle q_2, \sigma_2, q_3\rangle, \ldots, \langle q_n, \sigma_n, q_1\rangle\rangle$ (for $n \in \mathbb{N}$) that starts and ends at the same state.

2.2 Supervisory Control

The SCT [6] considers a *plant* G and a *specification* K that can both be modeled by FSMs. G describes by its language all possible behavior, while K describes the desired behavior of a controlled system consisting of G and a *supervisor* S to be automatically generated, that is, *synthesized*. The task of S is to observe the behavior of the plant and to influence it to behave within the specification. The supervisor is also an FSM, and then the *controlled system* is modeled by $G||S$, the synchronous composition of the plant and the supervisor.

The supervisor controls the plant by dynamically disabling events that the plant would otherwise have been able to generate. However, the alphabet of the plant is partitioned into the set of *controllable* and *uncontrollable* events, Σ_c and Σ_u, respectively. The supervisor must be such that it never tries to disable any uncontrollable events; this is the notion of *controllability*, formally expressed as:

$$\mathcal{L}(S)\Sigma_u \cap \mathcal{L}(G) \subseteq \mathcal{L}(S) \tag{B.1}$$

Note that full desired behavior of K may not be possible to achieve, due to parts of the behavior not being controllable in the above sense.

In addition, the supervisor is required to be *non-blocking*. Some states, $Q_m \subseteq Q_G$ (in this setting defined not as a part of G itself) are considered to

be *marked*, meaning that they are of special interest. The supervisor being non-blocking then means that from any state that the supervisor allows the controlled system to reach, it must be possible to reach some marked state. This is a liveness property of the system, formally expressed as:

$$\mathcal{L}(G||S) \subseteq \overline{\mathcal{L}_m(G||S)}, \tag{B.2}$$

where $\mathcal{L}_m(G||S)$ is the set of event sequences reaching marked states, that is, $\mathcal{L}_m(G||S) = \{s \in \mathcal{L}(G||S) \mid \delta_{G||S}(\langle i_G, i_S \rangle, s) \in Q_m\}$, and $\overline{\mathcal{L}_m(G||S)}$ denotes the *prefix-closure*, that is all shortenings of the event sequences of $\mathcal{L}_m(G||S)$, including the entire sequence and the empty sequence ε.

In addition, the supervisor is required to be *minimally restrictive*, meaning that when effecting its control of G, it disables as few transitions as absolutely necessary, giving the plant the maximum possible freedom.

The standard synthesis algorithm [6] starts from $G||K$ and removes states and transitions so as to fulfill (B.1) and (B.2), above. To also guarantee that S is minimally restrictive, the algorithm is careful to remove as little as absolutely necessary. Let $\sup \mathcal{CNB}(\cdot)$ denote this synthesis algorithm, then for $S = \sup \mathcal{CNB}(G||K)$, it holds that $G||S = S$.

Sometimes, as in the following, the specification K has no internal dynamics of its own, but is simply expressed by desired (or forbidden) states already in G. This can be fit into the above framework by making K a copy of G, with the desired states marked (and the forbidden states removed).

In either case, S is a sub-automaton of G.

3 User-Centric Discrete-Event System

A user-centric discrete-event system is a DES with which multiple users interact, each through their own distinct events. Each user has specific goals that they want to achieve. These goals correspond to specific states of the DES, so achieving some goal means interacting with the DES so as to make the system eventually reach one of those states.

Definition 2: *An FSM G is a* user-centric discrete-event system *if:*

- Σ_G *can be partitioned into* $n+1$ *subsets* Σ_0, Σ_1, ..., Σ_n, *for* $n \in \mathbb{N}$, *such that each* $\emptyset \neq \Sigma_i$, *for* $i \in \{1..n\}$, *is the set of* interaction events *used by user* U_i *to interact with* G*;*

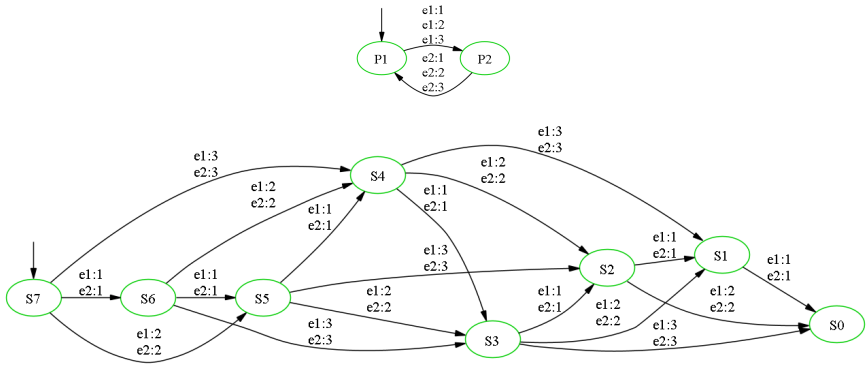


Figure 1: Tho FSM modeling the Stick-Picking user-centric discrete-event system game. The top FSM models the player's turn taking, with Player 1 taking the first turn. The bottom FSM models the decrease of the number of sticks as each player takes their turn, starting at 7 sticks.

- *For each user U_i there exists a set of states $\emptyset \neq Q_{U_i} \subseteq Q_G$ designating the* goals *of U_i. Goals may overlap between users, that is, it may hold that $\emptyset \neq Q_{U_i} \cap Q_{U_j}$ for $i, j \in \{1..n\}, i \neq j$.*

The events of Σ_0 (which might be empty) are referred to as the *internal events* of G. As users interact with G through their interaction events, one or more internal events may be triggered in sequence.

Note that the interaction events are not considered to be forcible, they are just ordinary events through which a user may choose to interact with G. If in a certain state there are only interaction events enabled, one of those will eventually be chosen by one of the users.

As an example, consider a simple two-player game, called the *Stick-Picking Game*, where users take turns to pick a number of sticks, 1, 2, or 3, to decrease the total number of sticks starting at 7. The player to pick the last stick loses the game. A DES model of this game could consist of one FSM modeling the player's turn taking and one FSM modeling the decrease of the number of sticks, see Fig 1.

In Fig 1, the top FSM models the turn picking, showing that Player 1 takes the first turn. The `e1:1`, `e2:1`, etc., events denote that Player 1 or Player 2 pick a number of sticks, respectively, with Player 1 starting to pick. The

bottom FSM models how the number of sticks decrease as each player takes their turn. For instance, the transition from `S5` to `S2` is taken when either player picks three sticks. The model of the sticks is agnostic to which player starts the picking.

In this particular case, the model has no internal events, but this is generally not the case. Also, this particular model has a finite language, which again is generally not the case.

4 Bias in UCDES

Bias exists in a user-centric discrete-event system when a user or users[1] can by carefully selecting their interaction events steer the DES to their benefit, or detriment of other users. The sequences of internal and interaction events representing such a steering is called a *strategy*.

There are two types of bias. *Positive bias* towards a user exists when that user can by choosing their interaction events steer the DES to the benefit of themselves, that is, eventually reaching some of their goals. *Negative bias* against a user exists when other users can by choosing their interaction events steer the DES to the detriment of that user, meaning always avoiding the that user's goals.

Note that the existence of positive bias towards a user, does not necessarily mean that there exists negative bias against other users, or the other way around. This entirely depends on the goals of the respective users. In some cases, users may have complementary goals so that positive bias exists towards all users simultaneously. It may even be the case that some user-centric discrete-event system embed negative bias against all users.

The choice of which interaction events to consider controllable decides which user's perspective the bias is regarded from, while the choice of goals to consider, the marked states, relates to the type of bias, positive or negative, that is considered.

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

5 Detecting Bias in UCDES

Bias in a user-centric discrete-event system can be detected by computing a supervisor with appropriately chosen controllable events and marked states.

Let a *configuration* be a tuple $\langle \Sigma_c, Q_m \rangle$, where Σ_c is the set of events to consider controllable, and Q_m is the set of states to consider marked. Let G be a UCDES and let $S = \sup \mathcal{CNB}(G, \langle \Sigma_c, Q_m \rangle)$ denote the synthesis from G of a controllable and non-blocking supervisor, S, for the configuration $\langle \Sigma_c, Q_m \rangle$, that is, with Σ_c the controllable events and Q_m the marked states.

If no supervisor exists, then it is clear that no marked states are reachable, hence no user with Σ_c as interaction events can steer G to reach the goals represented by Q_m; that is, no strategy for any bias, whether positive or negative, exists.

If a supervisor does exist, it is controllable (B.1) and non-blocking (B.2). Being controllable, means that with Σ_c as interaction events a user can make sure that whatever other users do, this always remains within the behavior of the supervisor. Non-blocking, on the other hand, meaning that some marked state is always reachable, is a weak property in that it only guarantees the *possibility* for some marked state to be reached, it does not guarantee that any marked state will ever actually be reached. The essence of that weakness is that the supervisor can include cycles wherein the system just runs around without ever reaching any marked state. A strategy, though, is meant to *guarantee* that a user can always steer the system to a marked state or away from it. Thus, a supervisor is not always a strategy.

Definition 3: *Given a UCDES G, and considering a configuration $\langle \Sigma_c, Q_m \rangle$, a* strategy *$T$ is a sub-automaton of G such that:*

$$\mathcal{L}_m(T) \subseteq \{s \in \mathcal{L}(G) \mid \delta_G(i_G, s) \in Q_m\} \tag{B.3}$$

$$\mathcal{L}(T)(\Sigma_G \setminus \Sigma_c) \cap \mathcal{L}(G) \subseteq \mathcal{L}(T), \tag{B.4}$$

$$\mathcal{L}(T) = \overline{\mathcal{L}_m(T)}, \tag{B.5}$$

$$\forall s \in \mathcal{L}(T), \forall t \in \Sigma_G^* : st \in \mathcal{L}(T) \setminus \mathcal{L}_m(T), \quad \exists n \in \mathbb{N} : |t| < n. \tag{B.6}$$

In Definition 3, expression (B.3) defines the marked language of T to be a subset of the sequences of events that reach the marked states Q_m. Expression (B.4) is the standard controllability with respect to the uncontrollable events $\Sigma_G \setminus \Sigma_c$. Expression (B.5) is the standard non-blocking, which as

discussed above does not guarantee that some marked state will actually be reached. However, expression (B.6) guarantees the absence of cycles with non-marked states. Together, expressions (B.4)–(B.6) mean that irrespective of what other users do, the user whose interaction events are considered controllable can always steer the system to reach some (marked) state favorable to that user (positive bias), or detrimental to some other user (negative bias).

Thus, given a UCDES G, a strategy to reveal bias towards or against a user U_i can be computed by:

(i) Setting U_i's interaction events, Σ_i, to be controllable.

(ii) Marking the states favorable or detrimental to U_i.

(iii) Synthesing a supervisor $S = \sup \mathcal{CNB}(G, \langle \Sigma_c, Q_m \rangle)$.

(iv) If S contains cycles of non-marked states, break those cycles, regard S as a new specification and go to (iii)).

Detecting positive bias towards a user U_i is achieved by the configuration $\langle \Sigma_i, Q_m \rangle$, where U_i's external actions Σ_i are considered controllable, and $Q_m = Q_{U_i} \subseteq Q_G$ is the set of U_i's favorable states.

Detecting negative bias against a user U_i is achieved by the configuration $\langle \bigcup_{j \neq i} \Sigma_j, Q_m \rangle$, with $Q_m \subseteq Q_G \setminus Q_{U_i}$ a set of states detrimental to U_i; here the other user's interaction events are considered controllable, and the strategy, if it exists, allows U_i to do whatever they want (but to no avail).

In both cases, the internal events are considered uncontrollable, capturing that no user can affect the internal behavior of the UCDES.

When a supervisor has a finite language, no cycles exist, and hence no cycles with non-marked states, thus such a supervisor is directly a strategy. When the supervisor language is non-finite, things are not as easy; it needs to be checked for cycles of non-marked states, which can be done in time complexity $\mathcal{O}((|Q_S| + |\delta_S|)(c + 1))$, with c the number of cycles, and space complexity $\mathcal{O}(|Q_S| + |\delta_S|)$, by the extended Johnson's algorithm [11]. And if such cycles are found, they need to be removed, which can be done by removing a transition of the cycle that is not involved in any other cycle, and then the synthesis has to be performed again with the altered supervisor as specification. This iteration has to be done until it converges, which in many cases means when the *null* supervisor is computed. At that point it is clear that no strategy exists.

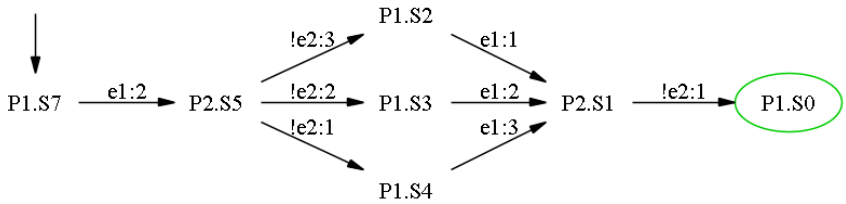


Figure 2: Supervisor for the two-player stick-picking game showing positive bias towards Player 1.

6 Examples

This section presents some examples to illustrate the notion of bias and how to detect it. The models are available at `https://tinyurl.com/BiasUCDES`.

6.1 Two-Player Stick-Picking

For the two-player stick-picking game of Fig 1, when examining positive bias towards Player 1, Player 1's interaction events are set controllable, and the states `P1` and `S0`, which together mean that Player 2 took the last stick, are marked. For this model, a supervisor can indeed be synthesized, see Fig 2, and since this is a finite-language supervisor it shows that there does exist positive bias towards Player 1.

Examining the strategy exhibited by the supervisor of Fig 2, it is seen that Player 1 should start by picking two sticks, then whatever Player 2 does, Player 1 can always steer the game into the state `P2.S1` where there is only one stick left for Player 2 to pick, and thus Player 2 has to pick that stick and loses the game.

In this particular case, the two-player stick-picking game, positive bias towards Player 1 means also negative bias against Player 2. And indeed, examining negative bias against Player 2, by setting Player 2's interaction events controllable and and all other events uncontrollable, and marking Player 1's winning state `P1.S0` (with Player 1 still taking the first turn), no supervisor exists. This seems typical of two-player games; if one wins the other loses. However, for multi-player games this is not necessarily the case.

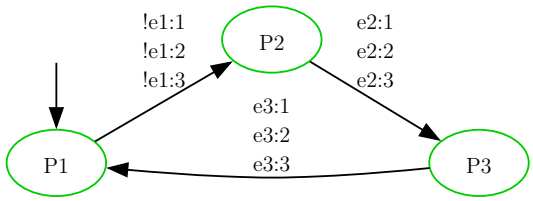


Figure 3: Turn taking model of the 3-player, 7-sticks Stick-Picking Game.

6.2 Multi-Player Games

For a multi-player game, consider the 3-player, 7-sticks stick-picking game. In that case the turn-taking has three states, as seen in Fig 3, where all states are marked as yet no detection of bias is considered. The model of the actual stick-picking is shown at the bottom of Fig 1. For the configuration $\langle \Sigma_1, \{\texttt{P1.S0}, \texttt{P3.S0}\}\rangle$, which defines Player 1's events to be controllable, and Player 1's not-losing states to be marked, no supervisor and hence no strategy exists, showing that there does not exist positive bias towards Player 1. In fact, there does not exist positive bias towards any individual player.

However, there does exist negative bias against individual players, meaning that two players can collude against the third. For instance, with the configuration $\langle (\Sigma_2 \cup \Sigma_3), \{\texttt{P2.S0}\}\rangle$, Player 1's events are uncontrollable, and the state where Player 1 takes the last stick and loses the game is marked, a supervisor does exist, see Fig 4. Since this supervisor has a finite language, it is a strategy that shows negative bias against Player 1. The other two players can collude to guarantee that Player 1 always loses the game.

6.3 Non-finite Languages

The above examples are all special cases where the supervisor language is finite, hence a supervisor is directly a strategy. As an illustration of the complexity that arises with a non-finite supervisor language, this section uses the supervisor synthesized for a variant of the `Two Machines and a Buffer` model of [12]. This system can be viewed as a user-centric discrete-event system with two users, a `Loader` and an `Unloader`, with their respective interaction events {`load1`, `load2`}, and {`unload1`, `unload2`}, and the set of internal events empty. Several supervisors are shown with the same structure, only

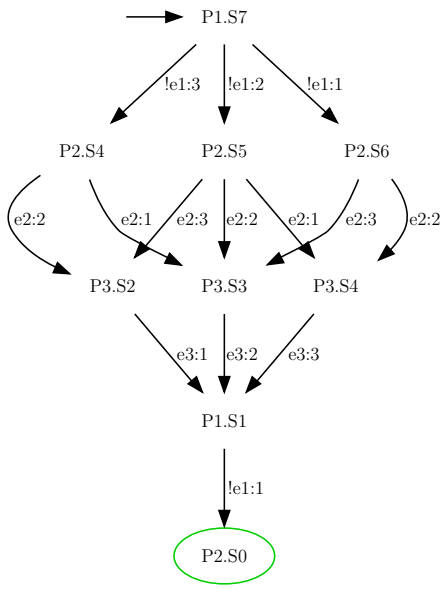


Figure 4: Negative bias against Player 1 in the 3-player, 7-sticks Stick-Picking Game.

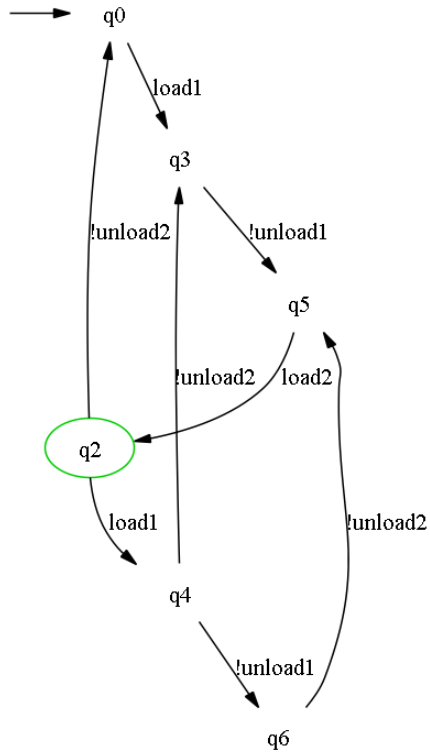


Figure 5: Supervisor for `Two Machines and a Buffer` model, this supervisor is a strategy revealing positive bias towards `Loader`.

differing in what state is marked. Of course, all supervisors are controllable and non-blocking w.r.t. to the plant and the uncontrollable events.

Fig 5 shows the supervisor computed for the `Two Machines and a Buffer` model with the events of `Loader` controllable, and the events of `Unloader` uncontrollable. The state `q2` is marked, as it represents a favorable state for `Loader`, whose perspective bias is investigated from. By inspection, it can be assessed that there is no cycle of non-marked states, thus the supervisor is a strategy that reveals positive bias towards `Loader`; whatever `Unloader` does, `Loader` can always steer the system to reach `q2`.

Fig 6 shows two supervisors for the `Two Machines and a Buffer` model,

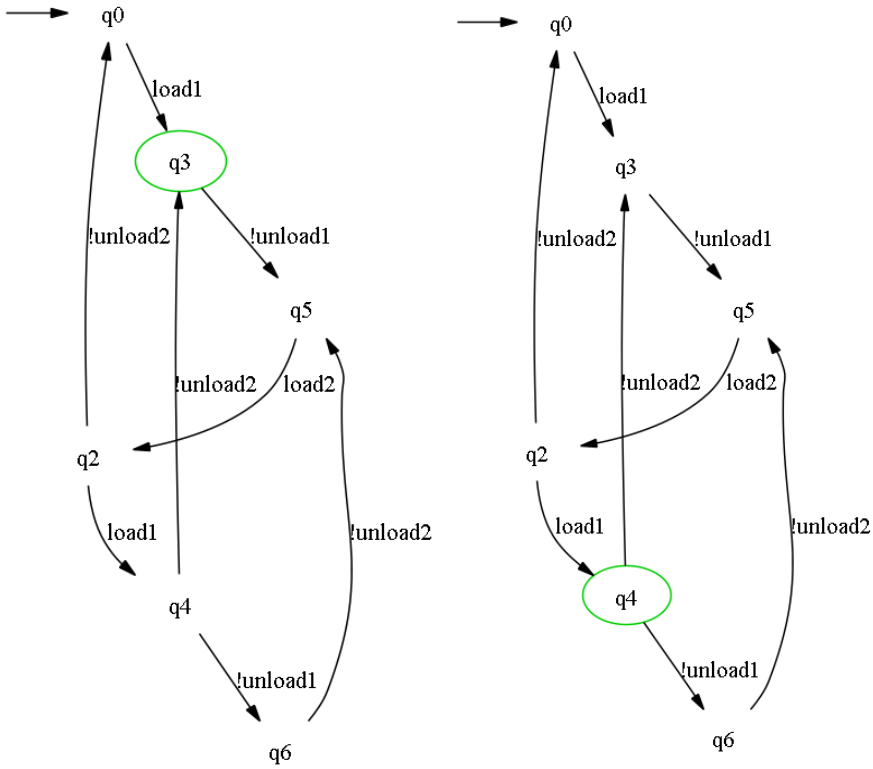


Figure 6: Two variants of a supervisor for `Two Machines and a Buffer` model. In either case, the supervisor is not a strategy. With `q3` marked (left), a strategy does exist; with `q4` marked (right), no strategy exists.

computed with different marked states, that is, states considered favorable to `Loader`. In either case, the supervisor is not a strategy because of cycles of non-marked states.

For the supervisor of Fig 6, left, with `q3` marked, the cycle of non-marked states is `q5 q2 q4 q6 q5`. Thus, since at `q4`, `Unloader` can always choose the `unload1` event to remain with in the cycle of non-marked states, `Loader` is not guaranteed to be able to steer the system to always reach the marked state `q3`.

However, from the supervisor of Fig 6, left, it is possible to break the cycle of non-marked states and find a strategy for `Loader`. Except for the $\langle$`q5`, `load2`, `q2`$\rangle$ transition, removing any of the transitions in the cycle and re-synthesizing a supervisor togehter with the original plant and specification, will generate a strategy that guarantees that `Loader` can always steer the system to reach `q3`. The reason that `load2` cannot be removed is that it is also part of a cycle with some marked state. The resulting strategy is a simple cycle over the states `q0 q3 q5 q2 q0`.

For the right supervisor of Fig 6, things are different; the cycle of non-marked states `q0 q3 q5 q2 q0`, now includes the initial state `q0`. Thus, removing this cycle will result in an empty strategy, meaning that no strategy exists, hence there is no bias towards `Loader` in this case.

To summarize,

- With `q2` (or `q5`) marked, the supervisor is a strategy;
- with `q3` marked, the supervisor is not a strategy, but a strategy exists;
- with `q4` (or `q6`) marked, a supervisor exists, but no strategy exists.

Finally it can be noted that for the configuration $\langle\{$`unload1`, `unload2`$\}, Q_m\rangle$, that is, from the perspective of `Unloader` with whatever state marked, no supervisor exists, hence no strategy and therefore no bias.

7 Conclusions

This paper defines the notion of *user-centric discrete-event system*, which are DES with which users interact, each with their own distinct events. Each user has specific goals they want to achieve with their interaction, and these goals can be represented by marking specific states of the DES. In such systems, *bias*

may be embedded, which manifests itself through the existence of a *strategy* to enforce that bias; positive bias towards a user or users that allows these users to steer the DES to always achieve their goals, or negative bias against these users that prevents them from achieving their goals. The existence or absence of such strategies can be revealed by supervisor synthesis. If no supervisor exists, then no strategy and hence no bias exists.

However, if a supervisor exists, then this may or may not be a strategy, as a supervisor, though being non-blocking, does not guarantee the ability to actually reach some marked state. Thus, a strategy is a supervisor with added constraints on the cycles of non-marked states.

Future work includes investigating the exact properties of cycles of non-marked states that guarantee that a strategy exists given a supervisor with non-finite language. Also, an efficient algorithm, preferably modular or compositional, for directly computing strategies without first computing supervisors would be interesting.

References

[1] F. Reijnen, J. van de Mortel-Fronczak, M. Reniers, and J. Rooda, "Design of a supervisor platform for movable bridges," in *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, 2020, pp. 1300–1306. DOI: 10.1109/CASE48305.2020.9216894.

[2] B. van der Sanden, M. Reniers, M. Geilen, T. Basten, J. Jacobs, J. Voeten, and R. Schiffelers, "Modular model-based supervisory controller design for wafer logistics in lithography machines," in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2015, pp. 416–425. DOI: 10.1109/MODELS.2015.7338273.

[3] T. Korssen, V. Dolk, J. van de Mortel-Fronczak, M. Reniers, and M. Heemels, "Systematic model-based design and implementation of supervisors for advanced driver assistance systems," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 2, pp. 533–544, 2018. DOI: 10.1109/TITS.2017.2776354.

[4] N. Parekh, W. Ahrendt, and M. Fabian, "Automatic conversion of smart contracts for non-blocking verification," *IFAC-PapersOnLine*, vol. 58,

no. 1, pp. 282–287, 2024, 17th IFAC Workshop on Discrete Event Systems WODES 2024, ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2024.07.048.

[5] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2023. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[6] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989. DOI: 10.1109/5.21072.

[7] B. Bonafilia, P. Carlsson, S. Nilsson, and M. Fabian, "Robust manual control of a manufacturing system using supervisory control theory," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 748–753, 2014, 19th IFAC World Congress, ISSN: 1474-6670. DOI: 10.3182/20140824-6-ZA-1003.01812.

[8] W. Grassmann, "Rethinking the initialization bias problem in steady-state discrete event simulation," in *Proceedings of the 2011 Winter Simulation Conference (WSC)*, 2011, pp. 593–599. DOI: 10.1109/WSC.2011.6147788.

[9] N. Parekh, W. Ahrendt, and M. Fabian, "On detecting bias in smart contracts using supervisor synthesis," submitted to WODES 2026.

[10] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer Cham, 2021, ISBN: 978-3-030-72272-2. DOI: 10.1007/978-3-030-72274-6.

[11] K. A. Hawick and H. A. James, "Enumerating circuits and loops in graphs with self-arcs and multiple-arcs," in *Proc. 2008 Int. Conf. on Foundations of Computer Science (FCS'08)*, Las Vegas, USA: CSREA, 14-17 July 2008, pp. 14–20.

[12] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," in *Analysis and Optimization of Systems*, A. Bensoussan and J. L. Lions, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 475–498, ISBN: 978-3-540-39010-7.

PAPER C

On Detecting Bias in Smart Contracts Using Supervisor Synthesis

Nishant Parekh, Wolfgang Ahrendt, Martin Fabian

Proc. 18th Workshop on Discrete Event Systems, Eindhoven, Netherlands, May, 2026

Abstract

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

1 INTRODUCTION

Smart contracts [1] are programs that execute within a blockchain environment, enforcing the terms defined by the contracts without any third party involvement. The trustless nature of the smart contract ecosystem enables interaction between multiple non-trusting parties, allowing them to participate in a given business (or game) scheme by placing their trust in the contract's implementation. However, absence of a third party assurance presents risks of its own, as interacting with poorly or maliciously designed smart contracts can harm some users. It could even be that a smart contract is intentionally designed to be advantageous to one side, while the opposing party remains

exposed to the biases coded in the contract. Even if the implementation of smart contracts is openly available [2], it can be difficult to see potential biases embedded in the implementation.

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

Previous work [3] outlines principles for automatically converting smart contracts from their source code to *Extended Finite State Machines* (EFSMs) [4, 5], and shows how the *Supervisory Control Theory* [6] can be applied to verify *non-blocking* behavior. Non-blocking is a progress property that when fulfilled guarantees that some states of particular interest, *marked* states, can always be reached, and in [7] this is used to verify whether a contract is resilient to a certain kind of denial-of-service attacks by a malicious user. If the contract is not resilient to such attacks, counter-examples can guide the developer in how to make the contract resilient.

Using the automatic conversion from smart contract source code to EFSMs of [3], this paper investigates how to detect whether a smart contract embeds biased behavior that is beneficial or detrimental to one or more users. For this, the distinction between *controllable* and *uncontrollable* events (which was irrelevant for the non-blocking verification of [7]) is introduced into the EFSM model of the smart contract. With careful selection of which events are controllable, and appropriate marking of states, computing a supervisor (which was not done in [7]) can reveal whether there exists a *strategy* that allows a particular user (or colluding users) to steer the contract towards beneficial outcomes regardless of what other users do. Revealing the presence of such a strategy reveals ways in which the contract's logic can be leveraged by a user (or users) to always reach their desired outcome. Similarly, the existence of a strategy can reveal if a (coalition of) user(s) can prevent a user (or users) from reaching their goals.

Current research on smart contract verification mainly focuses on security and safety using techniques such as model-checking and symbolic execution. Existing work around detecting bias and collusion is limited and primarily focuses on preventing collusion in auction and voting smart contracts. Related

work by [8] presents a framework, FairCon, for auction and voting smart contracts, to verify truthfulness, efficiency, optimality and collusion-freeness by converting smart contracts to mechanism-design models and verifying fairness properties using SMT. Another related work, CReam [9] presents a smart contract based e-auction system that achieves collusion-freeness by building in incentive mechanism directly in the contract. Our work on the other hand, focuses on whether the contract's behavior enables enforcing strategies for some users, and provides a generalized framework for detecting bias in smart contracts. To the extent of our knowledge, little to no work addresses detecting biases embedded in smart contracts.

The use case of this paper, the NineGame smart contract, is a three-player variant of the Game of 21 [10, 11], a full information multi-player game where players take turns to choose a number within a given range to increment a running total towards a target value, making it a suitable setting to investigate if a winning or losing strategy exists for the players. Such a strategy then defines at each state the actions that guarantee to reach the goal, irrespective of the actions of the other players. The results show that though there is no winning strategy for any single player, making the game appear fair, there do exist joint strategies in which two players can collude and force the third player to lose.

2 Background

This section presents a terse overview of the background necessary for the rest of the paper, the reader is referred to [1, 3, 5–7] for more comprehensive explanations.

2.1 Smart Contracts, Ethereum, and Solidity

The first, and still major, blockchain framework supporting smart contracts is Ethereum [1], with its built-in cryptocurrency Ether. In Ethereum not only users, but also contracts can receive, own, and send Ether. Ethereum miners look for transaction requests on the network. A transaction request contains the address of a contract to be called, the call data, and the amount of Ether to be sent (which can be 0). Miners are paid for their efforts with units of (Ether priced) *gas*, to be paid by the address that requested the transaction.

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

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

2.2 Extended Finite State Machines

Extended Finite State Machines [4, 5] are finite-state machines extended with bounded, discrete variables and associating with the transitions *guards* and *actions* over those variables. A guard is a condition that when satisfied enables the transition, and when the transition occurs the variables associated with the actions are updated.

Define an EFSM as $E = \langle \Sigma_E, L_E, L_E^0, L_E^\omega, \delta_E \rangle$, with Σ_E the finite set of event labels, the *alphabet*; L_E the finite set of *locations*, with $L_E^0 \subseteq L_E$ the set of initial, and $L_E^\omega \subseteq L_E$ the set of marked locations; δ_E the *transition relation*, where a transition $\langle l_m, \sigma, g, a, l_n \rangle$ denotes that the EFSM can transit from location $l_m \in L_E$ to location $l_n \in L_E$ when the guard g evaluates to *true* and the event $\sigma \in \Sigma_E$ occurs, and then the action a is performed.

The *state* of an EFSM is given by the location together with the values of the variables. Just as locations can be marked, so can variable values be marked. Thus, a *marked state* is one where the location and all variable values are marked. The (prefix-closed) *language* of an EFSM E, $\mathcal{L}(E) \subseteq \Sigma_E^*$, is the set of all possible finite sequences of events. The *marked language*,

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

$\mathcal{L}_m(E) \subseteq \mathcal{L}(E)$, is the set of all finite sequences of events reaching marked states. For a language $\mathcal{L}$, its *prefix-closure*, $\overline{\mathcal{L}}$, is the set of all prefixes of all traces of $\mathcal{L}$, that is, $\overline{\mathcal{L}} = \{s \in \Sigma^* \mid \exists t \in \Sigma^*, st \in \mathcal{L}\}$.

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

EFSMs interact with each other via *synchronous composition* [12] (denoted by $||$) of *shared events*, that is, events common between the EFSMs. In the composition, a shared event is enabled only if it is enabled in all EFSMs containing that event in their alphabets. For instance, the event `move01` in Fig 2 synchronizes with the same labeled event of the EFSM in Fig 3 that models the `require` clause on line 31 of Fig 1, so that the transition labeled by `move01` in Fig 2 cannot occur unless the guard `state_total + num'<=9` of Fig 3 evaluates to true.

2.3 Supervisory Control Theory

A *supervisor* is a control function whose purpose is to regulate the behavior of a *plant* ensuring that the systems jointly operate within a given *specification*. The Supervisory Control Theory (SCT) [6] provides a formal framework for *synthesis* of a supervisor S such that, given a plant G and a specification K, the supervisor dynamically restricts the behavior of G, so as to always fulfill K.

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

3 On Bias and Strategies

A smart contract's behavior is typically triggered by users initiating *external actions*, as the result of calling, through the blockchain framework, the con-

tract's public functions with certain parameter values. Upon such a call, the contract executes a sequence of *internal actions*, after which eventually a new external action may be triggered by the users.

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

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

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

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

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

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

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

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

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

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

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

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

Both (C.1) and (C.2) express variants of *controllability* [6]. What differs is the set of events considered uncontrollable. In both cases, the contract's internal events are uncontrollable, but for positive bias towards U, Def 1, the other user's events are considered uncontrollable, while for negative bias against U, Def 2, U's own events are considered uncontrollable.

As defined, a strategy allows a user to steer the contract by selecting external actions so as to reach some marked state. For this, the strategy has to be controllable wrt G and the uncontrollable events, so that uncontrollable events cannot divert from the strategy, and it has to be non-blocking so as to always be able to reach some marked state. This is much like a supervisor. However, non-blocking only guarantees the *possibility* for some marked state to be reached, it does not guarantee that any marked state will ever actually be reached; a supervisor can include cycles of non-marked states, from which no controllable user action can escape. Thus, to be a strategy, a supervisor must be free from such cycles of non-marked states. This means that if no supervisor exists, then no strategy and hence no bias exists. However, that a supervisor does exist is no guarantee that a strategy exists. Finding cycles of non-marked states can be done by Johnson's algorithm [13] in time complexity $\mathcal{O}((|Q_S| + |\delta_S|)(c + 1))$, with $|Q_S|$ the number of states, $|\delta_S|$ the number of transitions, and c the number of cycles of the strategy.

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

4 EXAMPLE

The Solidity code implementing the 3-player NineGame contract is shown in Fig 1. Players take turns choosing a number (1, 2 or 3) to increase a running total starting at 0. The player to first reach 9 wins the game and receives the funds held in the contract. The NineGame is a variant of the Game of 21 [10, 11], where instead of two, the game has three players. Addition of the third player allows us to analyze scenarios where users collude. The games are structurally similar, but lowering the target total from 21 to 9 significantly reduces the state space and allows to more comprehensibly illustrate the results.

The game begins with a user launching the contract and placing an initial bet set in the `constructor` at line 15. The contract creator's address is stored in the variable `player1`. Other user can participate in the game, if the game is available, by calling the function `join`, line 17, and placing an identical bet, line 19, following which the second and third user's addresses are stored as `player2` and `player3` respectively, lines 21 and 23. The variable `state.whoseTurn` at line 9 keeps track of whose turn it is next in the game, and is set at line 24 once all three players have joined. The game play takes place in the `move` function, where players take turns calling `move`, passing an integer indicating their choice. The value of the parameter is either 1, 2 or 3, ensured by the `require` at line 30, increasing a global running total `state.total`, line 32, initialized to 0 at line 11.

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

5 METHODOLOGY

Smart contracts are automatically converted from their source code to EFSMs by the framework detailed in [3]. The generated EFSM models jointly represent all possible behavior of the code, above denoted by G. For illustration,

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

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

a reduced version of the EFSM model [3] generated for function `move` (lines 27–45 of Fig. 1) is shown in Fig 2. While the full model is too large to be presented here, the reduced model preserves enough detail to follow the execution behavior and to investigate bias.

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

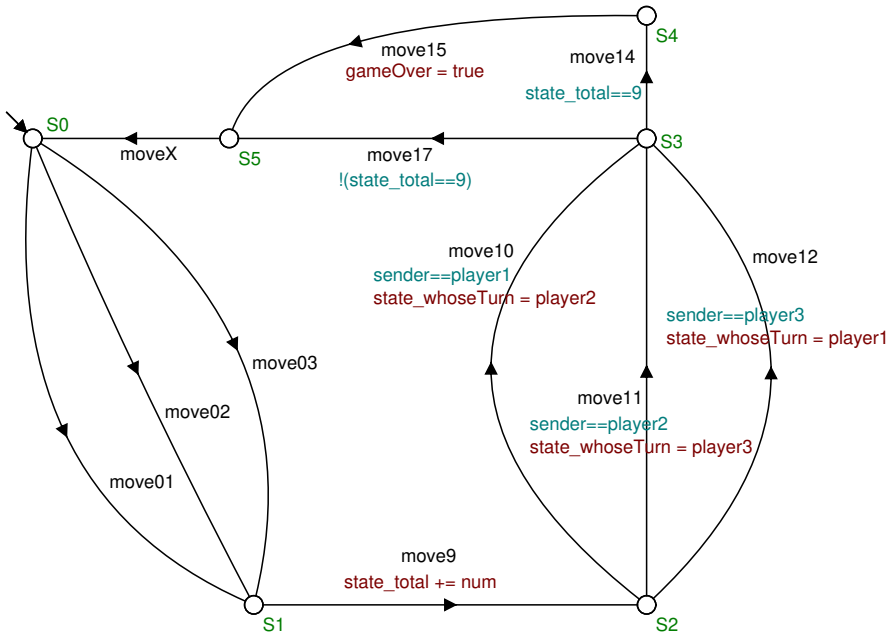


Figure 2: EFSM model of the `move` function.

5.1 Function Invocation

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

In addition, Fig. 4 models the non-deterministic assignment of the parameter `num` passed with the call of `move`. When `move01` occurs, the EFSM of Fig. 4

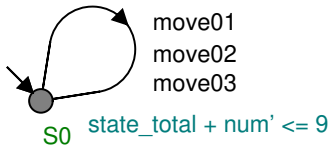


Figure 3: EFSM model of the require statement at line 31. The three events represent invocation of `move` by `player1`, `player2` and `player3`, respectively.

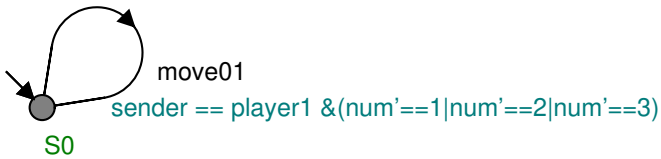


Figure 4: EFSM model tying `move01` to `player1`, and modeling the non-deterministic assignment of the `num` variable by `player1`.

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

5.2 Configuration

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

Configuration for positive bias

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

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

Configuration for negative bias

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

In the NineGame, when analyzing negative bias from `player1`'s perspective, `player1`'s events are considered uncontrollable, while the other player's events are controllable. As above, the variable value representing the end of game, `state_total==9`, is marked, but now the the variable values of `whoseTurn` representing that the *current* and *previous* player in sequence (`player1` and `player2` in this case) has the turn, is marked. Thus, when reaching a marked state, the game is over and it is either `player1`'s turn or `player2`'s turn, representing that

either `player3` or `player2` won the game. Concretely, this is done by marking the values `state_whoseTurn != x0002`, indicating that it was *not* `player1`'s turn at which `state_total` reached 9.

6 RESULTS AND ANALYSIS

This section presents the results of analyzing both positive and negative bias of the NineGame smart contract. As the example is rather small, the monolithic synthesis of SUPREMICA [14] is used, which synchronizes all model components and generates a single supervisor. For larger contracts, more efficient algorithms, like abstraction-based compositional synthesis [15], would have to be used. In any case, if no supervisor exists, this means that there is no strategy towards or against the player(s) from whose perspective bias is analyzed.

6.1 Analyzing positive bias

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

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

6.2 Analyzing negative bias

Analyzing negative bias for the three-player NineGame involves marking for each player the winning states of the other players, and considering the other player's external actions controllable. Synthesizing a supervisor now gives a

result, showing that negative bias exists for each player; that is, for each player the other players can force that player to lose no matter what the player does. Analysis of the synthesized supervisor further reveals that the strategy relies on coordinated behavior by the remaining two players. In other words, if two players collude and work jointly, they can enforce a losing outcome for the third player.

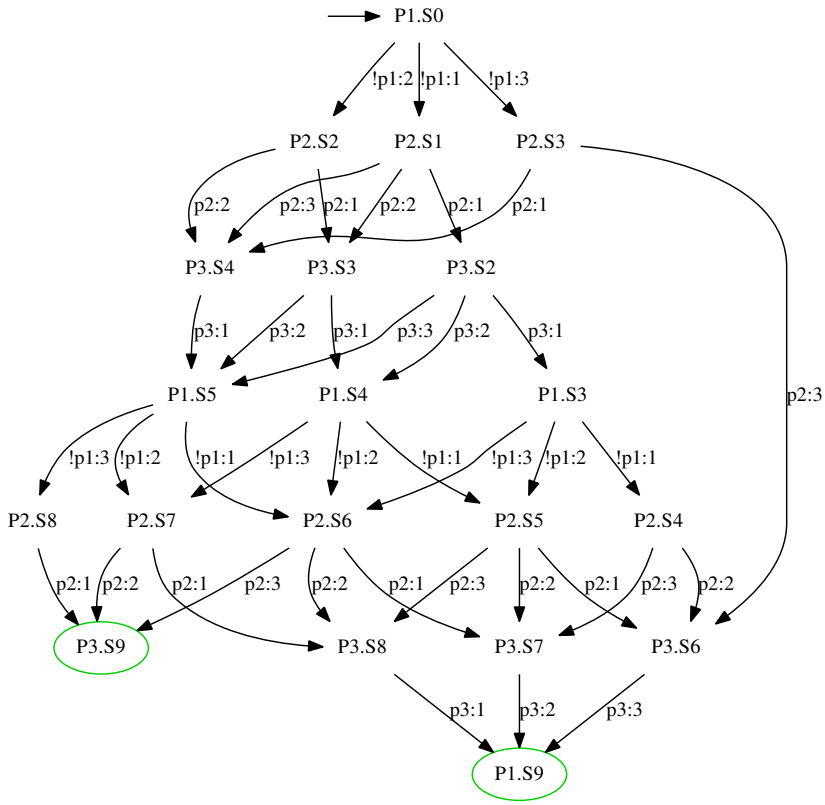


Figure 5: Abstract form supervisor revealing negative bias against `player1`. Events prefixed with an exclamation mark ! are uncontrollable.

An abstraction of the supervisor synthesized when analyzing negative bias against `player1` is shown in Fig 5. This supervisor describes how `player2` and

player3 can jointly play the game so that one of them wins (the marked states P3:S9 and P1:S9), hence player1 is guaranteed to lose. Here, an event pi:v represents that player i chooses value v. As shown in Fig 5, player1 can make any move during its turn. For instance at the initial state P1:S0, all events !p1:1, !p1:2, and !p1:3 are enabled. On the other hand, at location P3:S4 only p3:1 is enabled indicating that player3 can continue to steer the game to player1's loss only by selecting 1.

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

7 CONCLUSIONS AND FUTURE WORK

This paper shows how supervisor synthesis can be used to detect bias that might be embedded in, seemingly fair, smart contracts. Though the source code for deployed smart contracts is publicly available on platforms like Etherscan [2], detecting bias is hard even for simple contracts, like the NineGame. The analysis presented in this work can reveal such crucial information for deciding which contracts a user should (or should not) interact with. This is a subtle yet significant problem that traditional verification methods typically do not deal with.

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

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

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2023. [Online]. Available: `https://ethereum.github.io/yellowpaper/paper.pdf`.

[2] *The Ethereum blockchain explorer*, `https://etherscan.io/`, [Online; accessed 27-March-2025].

[3] N. Parekh, W. Ahrendt, and M. Fabian, "Automatic conversion of smart contracts for non-blocking verification," *IFAC-PapersOnLine*, vol. 58, no. 1, pp. 282–287, 2024, 17th IFAC Workshop on Discrete Event Systems WODES 2024, ISSN: 2405-8963. DOI: `10.1016/j.ifacol.2024.07.048`.

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

[5] M. Sköldstam, K. Åkesson, and M. Fabian, "Modeling of discrete event systems using finite automata with variables," in *2007 46th IEEE Conference on Decision and Control*, 2007, pp. 3387–3392. DOI: `10.1109/CDC.2007.4434894`.

[6] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," vol. 77, no. 1, pp. 81–98, Jan. 1989. DOI: `10.1109/5.21072`.

[7] N. Parekh, W. Ahrendt, and M. Fabian, "Smart contract denial-of-service analysis using non-blocking verification," *Discrete Event Dynamic Systems*, Dec. 2025, ISSN: 1573-7594. DOI: `10.1007/s10626-025-00418-5`.

[8] S.-W. L. Liu Ye Li Yi and Z. Rong, "Towards automated verification of smart contract fairness," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 666–677. DOI: `10.1109/CSCI46756.2018.00112`.

[9] S. Wu, Y. Chen, Q. Wang, M. Li, C. Wang, and X. Luo, "Cream: A smart contract enabled collusion-resistant e-auction," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 7, pp. 1687–1701, 2019. DOI: `10.1109/TIFS.2018.2883275`.

[10] D. J. B. Martin Dufwenberg Ramya Sundaram, "Epiphany in the game of 21," *Journal of Economic Behavior & Organization*, pp. 132–143, 2010, ISSN: 0167-2681. DOI: 10.1016/j.jebo.2010.03.025.

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

[12] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978, ISSN: 0001-0782. DOI: 10.1145/359576.359585.

[13] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975. DOI: 10.1137/0204007.

[14] S. .-. P. Repo, https://tinyurl.com/Supremica, [Online; Accessed: 2025-03-31].

[15] R. Malik, S. Mohajerani, and M. Fabian, "A survey on compositional algorithms for verification and synthesis in supervisory control," *Discrete Event Dynamic Systems*, vol. 33, no. 3, pp. 279–340, Sep. 2023, ISSN: 1573-7594. DOI: 10.1007/s10626-023-00378-8.