Modal Abstractions for Smart Contract Validation

Javier Godoy [1], Margarita Capretto [2], Martín Ceresa [3],
Juan Pablo Galeotti [1], Diego Garbervetsky [1,4], César Sánchez [2], Sebastián Uchitel [1,5]

[1]Universidad de Buenos Aires, UBA, ICC/CONICET, Argentina
{jgodoy, jgaleotti, diegog, suchitel}@dc.uba.ar
[2]IMDEA Software Institute and Universidad Politécnica de Madrid, Spain
{margarita.capretto, cesar.sanchez}@imdea.org
[3]Input Output, Madrid, Spain
martin.ceresa@iohk.io
[4]Guangdong Technion-Israel Institute of Technology, China
[5]Imperial College London, UK

***Abstract*—Smart contracts manage valuable assets, and their immutability hinders bug fixing. Therefore, pre-deployment verification and validation are critical. In fact, auditing has become mandatory in the pipeline of smart contract development. Auditors usually combine manual inspection with automated tools in their auditing work, looking for issues that may be domain dependent (i.e., pertaining to the correct implementation of requirements—which are often informal, partial, and implicit) or independent (e.g., reentrancy, overflow, etc.). To identify domain dependent issues, it is important to understand the non-trivial behavior of the implementation over sequences of calls made by callees playing different roles in the contract. In this paper, we propose a novel approach that combines predicate abstraction with modal transition systems to build abstractions that can help auditors in the smart contract validation process. The required inputs are a set of predicates provided as code and, optionally, constraints over smart contract function parameters. The output is a modal transition system that captures the contract's behavior. We report on a prototype that builds modal abstractions and an evaluation on two established benchmarks where we identified four previously unreported issues.**

***Index Terms*—Smart contract, predicate abstraction, modal transition systems**

I. Introduction

Blockchain technology enables transparency, immutability, and traceability of network transactions [29]. The inherent immutability of blockchains contrasts significantly with traditional software systems. Once a smart contract is deployed on the blockchain, no modification of its source code is possible. While this feature enhances security and transparency, it also presents a challenge by precluding the correction of defects in deployed smart contracts. Additionally, once a smart contract transaction is recorded, it becomes irreversible. Consequently, ensuring the validation and verification of smart contracts before deployment is crucial. A defect that can be easily fixed can lead to serious attacks, potentially resulting in significant financial losses. For instance, a flaw in the DAO smart contract [2] enabled attackers to steal 60 million USD in 2016 [33].

Smart contract errors can be broadly categorized into two types. The first type covers vulnerabilities that are common across different smart contracts. These include well-known issues such as overflow/underflow vulnerabilities. The second type, which this paper focuses on, are business logic errors, i.e., discrepancies between the intended behavior and the actual implementation of the smart contract's business logic.

A widely adopted industrial practice is to have smart contracts audited by specialized third-party security firms, often engaging multiple independent companies for thorough reviews. While these audits leverage automated and semi-automated tools (e.g., [18], [12], [38], [23], [37], [35]), manual tasks such as code inspection and informal documentation review play a significant role. Recent research [24] highlights the potential for developing tools to enhance error detection in smart contracts, but it cautions that many critical bugs may not be addressed *purely* through automation. Furthermore, in [14] authors conclude that business logic related bugs remain inadequately addressed by existing security tools.

In line with these insights, we address the challenge of contract validation by focusing on business logic errors. Our approach, unlike traditional methods requiring formalization of requirements, leverages auditors' expertise. Auditors can compare their understanding of the *intended* smart contract behavior with an automatic and formal abstraction of the implementation-under-analysis to identify inconsistencies.

In [22], the authors use predicate abstraction to construct a finite labeled transition system from the source code of smart contracts, over-approximating the valid sequences of function calls applicable to the contract. In these abstractions, a transition labeled f between two abstract states s_1 and s_2 means that there exists at least one concrete state in s_1 on which f can be called with some suitable actual parameter values such that it terminates successfully in a concrete state in s_2. These abstractions allow auditors (or other stakeholders) to scrutinize the behavior of the smart contract, identify defects, and enhance confidence in its alignment with independent manually crafted requirements. These abstractions can reveal, for example, a defect that prevents non-winning bidders from withdrawing their deposits upon auction finalization by highlighting the absence of a withdraw transition from an abstract

state that represents the auction being closed. Similarly, they may reveal a fault that allows bidding once closed.

In [22] the existence of an *endAuction* transition from a closed auction denotes that it *may be possible* for the beneficiary to receive funds committed by the auction winner, rather than that it *must be possible* for the beneficiary to receive their funds. Indeed, [22] defines what are referred to as *may* abstractions, while an abstraction that captures the guarantee for the beneficiary requires including *must* abstractions.

In this paper we propose the use of modal abstractions [26] to support smart contract validation. Specifically, we introduce *must transitions*, which model situations that a function can *always* be invoked and execute successfully from an abstract state. This contrasts with *may transitions*, which model that a function can be invoked and executed successfully depending on the specific concrete state of the abstract state of the contract. These transitions correspond to those used in [22]. Additionally, we introduce the concept of *constraint transitions*, which allows for constraining the existential quantification of parameters to specific user-defined values. For instance, by restricting withdrawals to a specific user (e.g., *msg.sender* $=$ `0xFA0...`), a resulting may-must abstraction might illustrate that the bidder `0xFA0...` is guaranteed to be able to withdraw their funds if they lose the auction.

In this work, we assume that predicates are provided by auditors, who extract them from the contract code (e.g., `require` clauses) or by leveraging their domain expertise in blockchain. Notably, the extraction of predicates from `require` clauses could be fully automated in future versions. To the best of our knowledge, this is the first approach that uses modal abstractions to validate smart contract implementations. The novel use of modalities and constraints are in line with the Smart Contract Security Verification Standard [6], particularly supporting activities related to two of its key areas: *G4. Business Logic* and *G5. Access Control*.

Summarizing, the contributions of this paper are (i) a novel modal abstraction for smart contract validation, (ii) a prototype for building modal abstractions, and (iii) an evaluation conducted on two established benchmarks, to determine empirically whether the modal and constraint features that we introduce contribute to the validation of smart contracts.

The remainder of this paper is organized as follows. In Section II, we introduce a motivational example showcasing the current limitations. Section III presents the basics on predicate abstraction. Section IV describes the formal model. In Section V, we discuss the evaluation of our approach, followed in Section VI by a discussion of threats to validity. In Section VII, we discuss related work and finally Section VIII concludes and presents future work.

II. Motivating example

In this section, we illustrate the limitations of the may abstractions proposed in [22] for smart contract validation and the advantages that using modal abstractions can provide for the same goal. We use an auction smart contract (Figure 1), inspired by [37] and implemented in Solidity [1].

A. An Auction Smart Contract

The auction smart contract shown in Figure 1 allows users to place bids within a specified bidding period and provides functionality for withdrawing funds and finalizing the auction. The contract maintains several state variables, including a mapping (`pending`) that tracks the bids received that are non-winning and will have to be refunded once the auction ends. The `constructor` function initializes the *auctionStart*, *biddingTime*, *beneficiary*, *owner* and *ownerFee* variables when the contract is deployed. The beneficiary will receive the *highestBid* minus a fee, which is transferred to the owner. The `Withdraw` function allows bidders to withdraw their funds if they were outbid once the bidding period has concluded. The `Bid` function allows users to place bids by sending an amount of Ether. This function checks that the bidding period has not ended, the bid amount is higher than the current highest bid, and the sender is neither the contract owner nor the beneficiary. Finally, `Bid` updates the state variables *highestBidder* and *highestBid* with the new bid value and sender, and the mapping *pending* with the previous highest bid from the previous bidder. The `AuctionEnd` function, enabled only when the bidding period has ended and only callable by the beneficiary, transfers as much Ether as the highest bid to the beneficiary account and the fee (set at deployment) to the owner. In the implementation shown in Figure 1 we introduced a bug deliberately in the `AuctionEnd` function: the instruction in Line 44 transfers the full *highestBid* amount instead of *highestBid - feeAmount*. As a result, the *feeAmount* is effectively stolen from the non-winning bidders.

B. May Abstractions

Figure 2a depicts the Labeled Transition System (LTS) generated by [22] for the auction contract. Each abstract state characterizes the set of concrete smart contract states that satisfy a collection of predicates defined by the *require* clauses of the public methods. The predicates are labeled *bid*, *withdraw*, and *auctionEnd*. The *bid* predicate holds when the bidding period is open and has not ended: $\mathit{bid}(c) \stackrel{\text{def}}{=} \mathit{c.auctionStart} + \mathit{c.biddingTime} > \mathit{block.number} \wedge \neg \mathit{c.ended}$. The *withdraw* predicate holds if the bidding period is closed and at least one user has funds in the *pending* mapping: $\mathit{withdraw(c)} \stackrel{\text{def}}{=} \mathit{c.auctionStart} + \mathit{c.biddingTime} \leq \mathit{block.number} \wedge \exists a : \mathit{Address} \cdot \mathit{c.pending}[a] > 0$.
The *auctionEnd* predicate holds if the bidding period is closed and not finalized: $\mathit{auctionEnd}(c) \stackrel{\text{def}}{=} \mathit{c.auctionStart} + \mathit{c.biddingTime} \leq \mathit{block.number} \wedge \neg \mathit{c.ended}$.
The abstract state $\{\mathit{bid}\}$ indicates that only the *bid* predicate holds. The state $\{\}$ represents an abstract state where none of the predicates hold. In Figure 2a, state $\{\}$ shows a transition labeled with τ, which is always enabled because it represents the passage of time (by modeling the increase of *block.number*). Further details will be provided in the next section.

A transition labeled f between two abstract states s_1 and s_2 means that there exists at least one concrete state c in s_1 and values for the actual parameter values of function f such that, when executing f in state c, the execution ends

```
1  contract Auction {
2    uint auctionStart;
3    uint biddingTime;
4    address payable beneficiary;
5    bool ended = false;  address payable owner;
6    uint highestBid = 0;  uint ownerFee = 0;
7    address payable highestBidder = address(0);
8    mapping(address => uint) pending;
9    constructor(uint _auctionStart, uint _biddingTime,
10     address payable _beneficiary, uint _ownerFee) public {
11     auctionStart = _auctionStart;
12     biddingTime = _biddingTime;
13     beneficiary = _beneficiary;
14     owner = msg.sender;
15     ownerFee = _ownerFee;
16   }
17   function Withdraw() public {
18     require(pending[msg.sender] > 0);
19     uint end = auctionStart + biddingTime;
20     require(block.number >= end);
21     uint pr = pending[msg.sender];
22     pending[msg.sender] = 0;
23     msg.sender.transfer(pr);
24   }
25   function Bid() public payable {
26     uint end = auctionStart + biddingTime;
27     require(end > block.number);
28     require(msg.sender != owner);
29     require(msg.sender != beneficiary);
30     require(!ended);
31     require(msg.value > highestBid);
32     pending[highestBidder] += highestBid;
33     highestBidder = msg.sender;
34     highestBid = msg.value;
35   }
36   function AuctionEnd() public {
37     uint end = auctionStart + biddingTime;
38     require(block.number >= end);
39     require(msg.sender == beneficiary);
40     require(!ended);
41     ended = true;
42     uint feeAmount = highestBid * ownerFee / 100;
43     owner.transfer(feeAmount);
44     beneficiary.transfer(highestBid);
45     //possible fix
46     //beneficiary.transfer(highestBid-feeAmount);
47   }
48 }
```

Fig. 1: Solidity Auction smart contract example

successfully in a concrete state c' in s_2. A transition labeled f gives information about behavior that *may* be possible in the abstract state for some parameters of f.

This abstraction enables an auditor to examine the behavior of the smart contract, thereby revealing crucial details about its underlying protocol. For instance, in state $\{bid\}$, it becomes evident that bidding is allowed until a specific time. The time progression (denoted by the τ) disables the *bid* method. Once the auction concludes, bidding is permanently disabled, as the bid transition is no longer feasible in subsequent states. These insights, derived from the abstraction, align with the mental model an auditor might have of the requirements.

Unfortunately, the information in this may-abstraction is too coarse for human auditors to expose the bug we have introduced in *auctionEnd*. After fixing the bug using Line 46 instead of 44, the abstraction remains identical. Indeed, by observing the abstraction, the auditor may incorrectly increase their confidence that *auctionEnd* is correct and that, combined with *withdraw*, the contact can be taken to state $\{\}$ where non-successful bidders and beneficiary have recovered their funds.

Consider a scenario in which an auction with 10% fee has had only two successful bids: 1ETH by *addressA* and 10ETH from a different address. Once the bidding period is over, the Auction concrete state reached is [*highestBid* $= 10$, *ownerFee* $= 10$, *pending* $= \{addressA = 1\}$, *balance* $= 11$, *ended* $=$ *false*, $\ldots$]. This concrete state corresponds to the abstract state $\{withdraw, auctionEnd\}$ as it satisfies predicates *withdraw* and *auctionEnd*. If the *withdraw* method is executed by *addressA*, the amount of 1ETH will be refunded, resulting in the concrete contract state: [*highestBid* $= 10$, *ownerFee* $= 10$, *pending* $= \{\}$, *balance* $= 10$, *ended* $=$ *false*, $\ldots$]. This state corresponds to the abstract state $\{auctionEnd\}$. In this case, executing the *auctionEnd* method will always fail because the balance is 10ETH but the method needs to transfer 10ETH to the beneficiary and 1ETH (10% of the highestBid) to the contract owner. No signs of this undesirable behavior can be seen in Figure 2a. Indeed, may abstractions can be deceptive. For example, a sequence of may transitions (such as *withdraw* followed by *auctionEnd* from state $\{auctionEnd, withdraw\}$) may not always be possible.

C. Modal Abstractions

We propose using two modalities for validation: may and must abstractions. Must abstractions are depicted in Figure 2b and Figure 2c. Black solid arrows are may transitions and their semantics is exactly the same as the transitions in Figure 2a. Blue dotted arrows correspond to must transitions.

A must transition labeled f between abstract states s_1 and s_2 means that *for all* concrete states c in s_1, there exist actual parameters values for function f such that executing f in state c successfully transitions to a concrete state c' in s_2.

A single must transition can have multiple destination states. For instance, the must transitions labeled *withdraw* from state $\{withdraw, auctionEnd\}$ has two possible destinations: $\{auctionEnd\}$ and $\{withdraw, auctionEnd\}$. This means that for every concrete state in $\{withdraw, auctionEnd\}$, executing *withdraw* will always succeed, leading to either $\{auctionEnd\}$ or back to $\{withdraw, auctionEnd\}$. The specific destination depends on the concrete state: if there is only one entry in *pending*, the transition leads to $\{auctionEnd\}$; otherwise, it remains in $\{withdraw, auctionEnd\}$. The constructor's behavior from the initial state is not represented by a single transition with two destinations but rather by two separate must transitions. This distinction arises because, for any given blockchain state, it is always possible to deploy a contract that reaches $\{bid\}$ by setting an auction end time in the future. Likewise, it is also possible to deploy a contract that reaches *auctionEnd* by setting an auction end time in the past.

In contrast to may abstractions, may/must abstractions are sensitive to the bug in the auction contract. Figure 2b presents

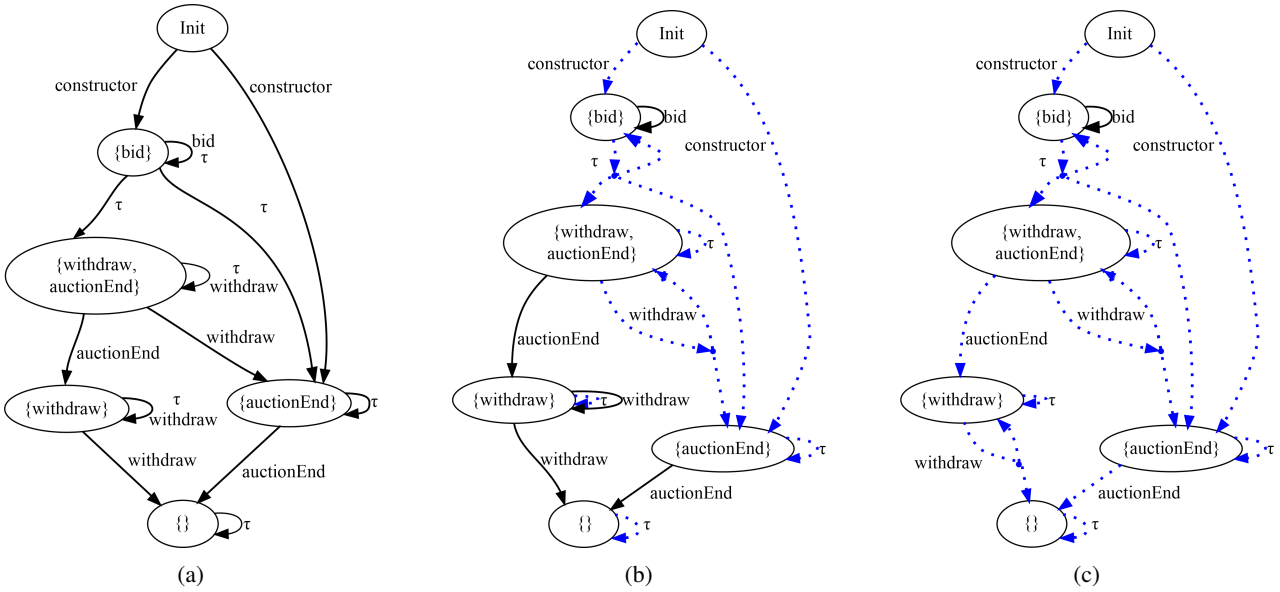

Fig. 2: (a) Abstraction generated by [22] for buggy and fixed Auction implementations. (b) Modal abstraction of the buggy implementation. (c) Modal abstraction for the fixed implementation. May transitions, must transitions, and hyper-must transitions are depicted by solid black, blue dotted, and forked blue dotted arrows, respectively.

a modal abstraction of the buggy contract listed in Figure 1, while Figure 2c shows an abstraction for the fixed contract.

When inspecting the abstraction for the buggy contract, an auditor may ask why there is no must transition from $\{auctionEnd\}$ to $\{\}$. Also, the transition *auctionEnd* from $\{auctionEnd\}$ is a may transition, whereas it is expected to be a must transition. The answer to this question is the scenario discussed previously in which a contract state [$highestBid = 10$, $ownerFee = 10$, $pending = \{\}$, $balance = 10$, $ended = false$, ...] can be reached. Such a scenario can be generated automatically and presented to the auditor, potentially helping them to identify and understand the defect.

It is important to note that the presence of may transitions does not necessarily indicate a bug. In Figure 2c, the *bid* transition is classified as a may transition, which might initially seem suspicious—why is there at least one concrete state in $\{bid\}$ where it is not possible to outbid the highest bidder? However, the bid amount is of type *uint256*, with a maximum value of $2^{32} - 1$. Thus, if a user places a bid at this maximum value, the contract would reach a concrete state in $\{bid\}$ where no further bids can be placed. While this scenario is unlikely in practice, it remains a theoretical possibility.

While abstraction Figure 2c provides useful insights, it cannot answer a key question: *Is it possible for a non-winning bidder to retrieve their funds?* Although *withdraw* transitions are must transitions —meaning that at least one bidder can always withdraw funds— it does not indicate whether the withdrawing bidder is the one who placed the bid.

To support validation of such questions, we introduce a feature that allows auditors to constrain the actual parameter values of specific function calls. In this case, an auditor may apply the constraint $\Phi_{withdraw} \stackrel{\text{def}}{=} msg.sender = A$ to *withdraw* transitions. Constraining transition f with Φ ensures it models the execution of f only when its parameters satisfy Φ.

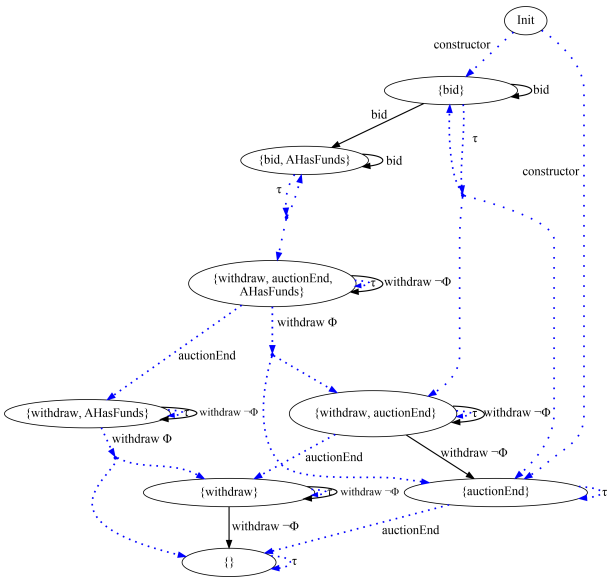

Fig. 3: Refined modal abstraction with the predicate *AHasFunds*, indicating whether user A can withdraw their funds. *withdraw* transitions have constraint $\Phi = \{msg.sender = A\}$.

Figure 3 shows a refined modal abstraction for the fixed auction contract. This refinement incorporates an additional predicate ($AHasFunds(c) \stackrel{\text{def}}{=} c.pending[A] > 0$) that models whether user A has funds available for withdrawal according to the contract's bookkeeping. Additionally, constraints $\Phi = \{msg.sender = A\}$ and $\neg\Phi$ are applied to *withdraw* transitions. In the refined modal abstraction, we can observe

that for abstract states where *AHasFunds* does not hold, there are no outgoing transitions with $\Phi_{withdraw}$, which is correct. Conversely, in abstract states where both predicates *AHasFunds* and *withdraw* hold, we observe that there are $\neg\Phi_{withdraw}$ may transitions (indicating no guarantee that users other than *A* can withdraw) and there are $\Phi_{withdraw}$ must transitions (indicating that *A* is always able to withdraw).

We now return to the question: *Is it possible for a non-winning bidder to retrieve their funds?* The refined abstraction provides evidence that the answer is positive for any bidder *A*. Once user *A* is outbid, the system reaches state $bid, AHasFunds$. The passage of time (via a τ must transition) guarantees that state $\{withdraw, AHasFunds, auctionEnd\}$ will be reached, from which user *A* can withdraw their funds (see the $withdraw_\Phi$ must transition). Even if *auctionEnd* is called before *A* withdraws, the contract still reaches state $\{withdraw, AHasFunds\}$, where $withdraw_\Phi$ remains guaranteed. Thus, user *A* is guaranteed to be able to withdraw their funds, regardless of the actions of other users.

Summarizing, may abstractions have limitations in answering key audit questions. Above, we illustrate how incorporating must transitions and constraints can better support auditors in reasoning about contract behavior.

III. Background

A. Predicate Abstractions

In program verification, abstraction typically refers to a mapping from concrete program states to abstract states. Predicate abstraction denotes a specific kind of abstraction where the mapping is induced by a set of logical predicates. Thus, all concrete states that satisfy the same predicates are mapped to the same abstract state [28].

B. Predicate Abstractions for Smart Contracts

At the function call level, the semantics of a smart contract resembles that of an object protocol. A smart contract state is captured by the values of the internal variables, and transitions are the functions calls that change its state. However, an important difference is that the global state of the blockchain is particularly relevant in a smart contract.

In [22], authors define the semantics of smart contracts, at a black box level of granularity, as a labeled transition system $\langle C, C_0, \Sigma, \longrightarrow\rangle$. The states in C represent the different values that the contract can have plus the state of the blockchain. Subset $C_0 \subseteq C$ contains all possible initial states at contract deployment. Labels in Σ are all of the possible function calls F (names plus actual parameter values) and an additional label τ representing changes in the blockchain that are uncontrollable by the contract (e.g., a transfer made outside the contract, the addition of a new block to the blockchain due to the progress of time, etc.). The set of transitions $\longrightarrow \subseteq C \times \Sigma \times C$ models all possible successful terminations of function calls or changes in the blockchain. The presence of an outgoing transition labeled with a specific function name f and actual parameter values $p_1, \ldots, p_n$ from state c to state c' (i.e., $(c, f(p_1, \ldots, p_n), c') \in \longrightarrow$) indicates the calling $f(p_1, \ldots, p_n)$ on the smart contract when it is in state c, it will successfully execute and transition the contract to state c'. Similarly, an outgoing transition labeled τ from state c to c' (i.e., $(c, \tau, c') \in \longrightarrow$) represents the fact that the blockchain state and, consequently, the contract's state, have changed from c to c'. These transitions are called may transitions and appear in the abstraction proposed by [22]. Moreover, the presence of any transition in the abstraction implies that the corresponding function's precondition must be satisfied. In smart contracts, preconditions are typically expressed using *require* statements (see, for example, Lines 28-32 from Figure 1).

C. Modal Transition System

Modal Transition Systems (MTS) are an extension of Labeled Transition Systems in which transitions between states can exhibit either the *may* or *must* modality [26]. The former indicates a behavior that *may* be allowed from the state, while the latter indicates a behavior that *must* always be allowed. MTS have been shown to be useful for behavioral description of systems in many contexts (e.g., [11]). There are many extensions and variants of MTS such as Disjunctive Modal Transition System [31] and Parametric Modal Transition System [30]. In this paper, we introduce a variation of MTS to express properties of relevance to auditors of smart contracts, but continue to refer to them as MTS.

IV. MTS for Smart Contracts Validation

We now formalize the modal abstraction introduced informally in Section II and briefly outline the prototype design.

A. Formal model

Consider a smart contract $\langle C, C_0, \Sigma, \longrightarrow\rangle$ and F be a family of contract functions. We use $\overrightarrow{P_f}$ for the set of all possible actual parameter values that can be used to invoke f. We consider constraint functions $\Phi_f : C \times \overrightarrow{P_f} \longrightarrow bool$ that limit the arguments that function f can be invoked at a given concrete state. We now introduce the predicate step, which will be used in the following definitions to formalize the concept of one-step reachability within the context of our modal abstraction.

$$\mathsf{step}(c, f, \phi_f, c') \stackrel{\text{def}}{=} \exists \overrightarrow{p} \in \overrightarrow{P_f} \cdot \phi_f(c, \overrightarrow{p}) \wedge c' = f(c, \overrightarrow{p})$$

Informally, predicate $\mathsf{step}(c, f, \phi_f, c')$ holds if there is an actual parameter value $\overrightarrow{p} \in \overrightarrow{P_f}$ that satisfy the constraint $\phi_f(c, \overrightarrow{p})$ and after executing function f on c with actual parameter values $\overrightarrow{p}$, the execution terminates successfully, leaving the smart contract in a concrete state $c' \in C$. Intuitively, the constraint function ϕ_f can disable the outgoing transition $f(c, \overrightarrow{p})$ over the concrete state c.

Following these definitions, we formalize our notion of Modal Transition Systems for smart contracts. Given a set P of predicates over states, a set F of smart contract public function labels (with the additional label τ), and a set of constraints ϕ_f, the model is defined by the tuple

$$\langle S = 2^P, S_0, \Sigma = \bigcup_{f \in F} \bigcup_{\phi \in \Phi_f} \langle f \times \phi\rangle, \xrightarrow{\mathsf{May}}, \xrightarrow{\mathsf{Must}}\rangle$$

1) Set S represents all possible abstract states in the model, defined as the powerset of the set of predicates P. We use s to refer to a particular abstract state, which is a subset of P. We use $c \in s$ to denote that a concrete state c belongs to abstract state s (i.e., for all $p \in s$, $p(c)$ holds).
2) Subset $S_0 \subseteq S$ contains all possible initial abstract states at the moment of contract deployment.
3) Σ is an action alphabet defined with the contract's public function names and an additional label τ, which represents changes in the blockchain that are not controllable by the contract. Each action is paired with constraints ϕ; for τ, we define $\phi_\tau = \mathsf{True}$.
4) Relation $\xrightarrow{\mathsf{May}} \subseteq S \times \Sigma \times S$ is a transition relation that models all possible successful invocations of function f or changes in the blockchain that satisfy the constraints Φ_F. Then, we say that we reach a concrete state c' from concrete state c through f, in symbols: $\langle s, \langle f, \phi_f \rangle, s' \rangle \in \xrightarrow{\mathsf{May}}$ if and only if the following predicate holds: $\mathsf{May} \stackrel{\text{def}}{=} \exists c \in s \cdot \exists c' \in s' \cdot \mathsf{step}(c, f, \phi_f, c')$
5) Relation $\xrightarrow{\mathsf{Must}} \subseteq S \times \Sigma \times 2^S$ is a one-to-n transition relation that models all possible successful terminations of function calls or changes in the blockchain. Note that, unlike in the previous definition, the target of the transition is now a set of abstract states. We denote the set of target abstract states from s as $T_s \subseteq S$. Moreover, an abstract state is included in T_s only if it is reachable from *all* concrete states in s, and T_s must be minimal. We refer to transitions with a single target state (i.e., $|T_s| = 1$) as must transitions, and those with multiple target states as hyper-must transitions. Formally, $\langle s, \langle f, \phi_f \rangle, T_s \rangle \in \xrightarrow{\mathsf{Must}}$ if and only if the following conditions hold:
   - NonEmpty: $s \neq \emptyset$
   - AllMay: $\forall s' \in T_s \cdot \exists c \in s \cdot \exists c' \in s' \cdot \mathsf{step}(c, f, \phi_f, c')$
   - Must: $\forall c \in s \cdot \exists s' \in T_s \cdot \exists c' \in s' \cdot \mathsf{step}(c, f, \phi_f, c')$
   - Minimal: $\nexists T'_s \subseteq T_s \cdot \forall c \in s \cdot \exists s' \in T'_s \cdot \exists c' \in s' \cdot \mathsf{step}(c, f, \phi_f, c')$

   For the sake of clarity, we added the AllMay constraint, although it is implied by Minimal and Must.

B. Prototype design

To evaluate our approach, we extended a *semi-automatic* predicate abstraction tool for smart contracts, taken from [22]. This prototype is designed to accept specifications and predicates written in the Alloy modeling language [27].

Alloy has two main components: the Alloy modeling language and the Alloy Analyzer. The Analyzer employs an exhaustive bounded approach to explore a model's state space up to a specified bound. It can generate counterexamples, allowing auditors to refine abstractions or to shed light on suspicious scenarios. For instance, if auditors encounter unexpected transitions or want to replicate a potential vulnerability, the prototype can produce examples exhibiting the behavior.

This extended prototype requires the following inputs (currently provided manually but potentially automatable): (i) an Alloy specification derived from the Solidity code, including pre- and postconditions for each function and an invariant predicate to exclude invalid instances; (ii) a set of abstraction predicates; and (iii) an optional Φ_f constraint (by default, $\Phi_f \stackrel{\text{def}}{=} \mathsf{True}$). Given these inputs, the prototype automatically builds a predicate abstraction of the smart contract, following a methodology similar to that used for building Enabledness Preserving Abstractions (EPAs) [15]. This methodology emphasizes introducing transitions between abstract states whenever a concrete transition is deemed feasible.

Extending abstractions to MTS involves translating the formulas from Section IV-A into first-order logic using Alloy. In our methodology, transitions labeled as must transitions involve a universal quantifier over the concrete states of an abstract state, which results in a formula that the Analyzer cannot solve directly. To overcome this, we negate the original formula into an existentially quantified expression that Alloy can handle through its exhaustive bounded analysis. The unsatisfiability of this existential formula up to a given bound validates the original universal quantifier. Unfortunately, this approach is unsound: if the bound is too small, we might erroneously identify a transition as must when it is not. However, our methodology targets manual validation by providing a visual representation of the contract's behavior. Therefore, if a suspicious must transition is identified, auditors can adjust the bound and repeat the analysis, which mitigates the unsoundness of the proposed technique.

The naive algorithm for calculating must transitions involves performing a specific reachability query for each potential transition. However, it quickly becomes impractical. Instead, our prototype follows a three-phase process. First, we build a LTS as described in [22]. Second, using this abstraction, we assign a default may modality to all transitions. We then verify whether each transition f satisfies its feasibility constraint ϕ_f. If not, a label $\neg\Phi$ is added to transition f; then we check the feasibility of must transitions for single target states. Third, for transitions that were not marked as must but reach multiple states, we evaluate hyper-must transitions by systematically checking feasibility of all possible combinations of target states, stopping at the smallest satisfying set. Artifacts for this paper are publically available on Zenodo [9].

Since the implementation was developed to empirically evaluate whether the proposed techniques contribute to smart contract validation, performance optimization was not a focus. Nonetheless, Table I reports the generation time in seconds for each phase to offer insight into execution times. Notably, since reachability queries for transitions in each phase are parallelizable, further optimizations in this area could substantially enhance performance.

V. Evaluation

In this section, we evaluate our predicate abstractions with must transitions for validating smart contract implementations. Specifically, we look at the following research questions.

- **RQ#1**: How prevalent are must transitions in modal abstractions of smart contracts?

- **RQ#2**: Does the use of constraint transitions help validate smart contracts when allowed user role is available?
- **RQ#3**: Do auditors exploit the distinction between may and must transitions to reveal issues about behavioral properties that cannot be captured by may abstractions?

To address **RQ#1**-**RQ#3**, we use two benchmarks taken from the existing literature. The first benchmark (denoted as $B\#1$) is the Microsoft Azure Workbench Blockchain [3]. This benchmark has been analyzed in several works [12], [38], [22]. A key characteristic of this benchmark is the inclusion of manually created diagrams representing the expected protocol for each contract. Additionally, each contract comes with an informal description detailing its intended functionality, states and roles involved. Each diagram specifies an "Application Instance Role" (AIR) for each transition, which refers to an instance role data member in the workflow that stores a global role, i.e., executing a function is only allowed if the user address matches the value stored in the instance data variable associated with that role (e.g., buyer, owner, bidder, etc.). For our analysis, we use the same contracts studied in [22], but with the fixed versions from [5].

The second benchmark (denoted as $B\#2$) provides an informal textual description of the requirements for each contract, but lacks details about roles and their executable methods. We used the predicate abstractions built in [22] as our baseline. For some smart contracts, we analyzed multiple subjects, corresponding to different abstractions of the same contract but built with alternative choices of abstraction predicates. The naming convention for each subject follows the structure $\texttt{subjectname} + \texttt{EPA} + \texttt{pred}_1 + \texttt{pred}_2 + \ldots$, where `EPA` indicates that some predicate abstractions are inspired by the `require` clauses [15], and $\texttt{pred}_1$, ... represent additional predicates. For example, *SimpleAuction+Ended+HB* refers to the SimpleAuction contract without EPA predicates but includes the predicates *Ended* and *HB* (related to highestBidder). Table I lists the subjects used for both benchmarks.

A key aspect of the modal abstraction generation process was the introduction of stronger invariant conditions within the Alloy model. In [22], the original invariant used for generating may transitions permitted certain concrete states where no transactions were possible, though this had no effect in the resulting abstraction. However, for must transitions, which require a function to be executable from any concrete state, these stronger invariants proved essential.

To answer **RQ#2**, we restricted the subjects to those of $B\#1$, as they are the only ones for which information about the authorized actions for each role is available. We used the same abstraction predicates used in [22]. Additionally, we added the constraint $\Phi_f \stackrel{\text{def}}{=} \textit{sender} = R$ for each public function f, where R represents the roles defined in the documentation as Application Instance Role (AIR). We then compared the generated modal abstraction with the original documentation diagrams. Our evaluation follows this reasoning: (i) if a transition with an expected role R is not classified as *must*, this indicates a potential issue because there exists a concrete state where a user with role R cannot execute the method; (ii) if a transition with a role different from the expected role R is classified as *must* or *may*, it may point to an access control issue or undocumented behavior.

To address **RQ#3**, one author and an experienced external auditor independently conducted a manual inspection of the modal abstractions for $B\#2$. Their goal was to identify unexpected, anomalous, or relevant behaviors. This inspection started with the abstractions alone, referencing the underlying code only to clarify the protocol's logic or when unusual behavior was detected. The external auditor received instructions in a virtual meeting explaining the may and must abstractions. Additionally, they were provided with a package [8] containing, for each subject, the Solidity code, at least one modal abstraction and a README file with a high level contract description and a brief explanation of the predicates used in each abstraction.

A. Results

*1) **RQ#1**:* Our findings revealed that must transitions were present in *all case studies*. The number of must transitions for each abstraction is shown in column $\#M$ in Table I. We observed that certain transitions are always *must*, regardless of the subject: τ and *constructor* transitions. This aligns with our expectations based on common understanding. Transitions labeled τ are time-related, which are always enabled and inherently `must` (and hyper-must) due to the inevitable progression of time. The constructor is necessary for deploying a contract and is expected to *always* be callable with some actual parameter values. Although rare, certain blockchain states may prevent contract deployment, but such scenarios were not found in our subjects. To refine the analysis, we excluded these always must transitions (i.e., τ and *constructor*) and examined the remaining must transitions. This is depicted in column $\#M{-}\{\tau, C\}$ in Table I. This shows that three abstractions exhibit no remaining must transitions: *Auction+EPA*, *EPX-Crowdsale* and *SimpleAuction+Ended+HB*. Upon inspection, we found that this is due to the low granularity of the predicates. For example, in *SimpleAuction+Ended+HB*, the predicates were based on whether the auction had ended and if a specific user was the highest bidder. This resulted in diverse transitions in each abstract state, not all of which were executable. For instance, in the abstract state $\{\textit{highestBidder} = A, \neg \textit{ended}\}$, the *auctionEnd* method is not classified as *must* because there are concrete states where the auction remains open (i.e., not enough time has elapsed). The *auctionEnd* method requires the auction to be closed (i.e., enough time has elapsed) but not yet ended. In another concrete state, where the auction is closed, executing *auctionEnd* is possible.

To analyze the prevalence of hyper-must transitions (graphically represented as blue forked transitions), we examined column $\#HM$ in Table I. These transitions appeared in 13 out of the 28 cases. When excluding τ and *constructor* transitions (column $\#HM{-}\{\tau, C\}$), the proportion of subjects remains consistent. Notably, no *constructor* transitions were classified as hyper-must transitions. This aligns with expectations, as

TABLE I: B denotes benchmark #1 or #2. $Subject$ refers to the contract name and predicates used for each contract. $|LOC|$ denotes the number of lines of code, $|S|$ abstract states, $|M|$ non-view/non-pure public functions. $\#Trx$ refers to the number transitions, $\#May$ May transitions that are not subsumed by must transitions, $\#M$ must transitions, $\#M - \{\tau, C\}$ must transitions excluding τ and Constructor, $\#HM$ hyper-must transitions, $\#HM - \{\tau, C\}$ hyper-must transitions excluding τ and Constructor. $Time$, $TimeM$ and $TimeHM$ are generation times in seconds for May, Must and HM transitions, respectively.

B	Subject	$\|LOC\|$	$\|S\|$	$\|M\|$	#Trx	#May	#M	#M-$\{\tau, C\}$	#HM	#HM-$\{\tau, C\}$	Time	Time M	Time HM
#1	AssetTransfer	225	10	11	32	0	32	31	0	0	15.291	727.625	0.269
	BasicProvenance	48	3	3	4	0	4	3	0	0	0.361	0.564	0.031
	DefectiveComponentC	33	2	2	2	0	2	1	0	0	0.305	0.227	0.045
	DigitalLocker	148	6	10	12	0	12	11	0	0	4.63	5.89	0.061
	FrequentFlyerRC	50	2	2	3	0	3	2	0	0	0.352	2.747	0.034
	HelloBlockchain	35	2	3	3	0	3	2	0	0	0.192	0.139	0.026
	RefrigeratedTransp	108	4	4	8	0	4	3	2	2	1.563	932.224	250.857
	RoomThermostat	48	2	4	4	0	4	3	0	0	0.387	0.653	0.033
	SimpleMarketplace	66	3	4	4	0	4	3	0	0	0.524	16.509	0.024
#2	Auction+EPA	51	4	4	15	3	4	0	4	2	11.123	320.136	42.398
	AuctionFix+EPA		6	5	21	3	8	2	5	3	10.724	318.497	64.16
	AuctionFix+EPA+Ended		6	5	21	3	8	2	5	3	10.902	314.686	87.5
	Crowdfunding_EPA+Balance	55	7	5	23	0	10	4	6	2	89.091	3932.952	1737.533
	CrowdfundingFix_EPA+Balance		6	5	21	0	11	4	4	2	117.916	4963.163	936.545
	EPXCrowdsale	171	7	6	58	14	8	0	6	6	22.495	182.081	484.761
	EPXCrowdsale+EPA		4	6	17	2	7	4	4	2	19.061	260.106	154.058
	EPXCrowdsale+EPA+isCSClosed		5	6	20	2	9	5	4	2	25.743	336.93	114.529
	EscrowVault	102	4	10	17	1	16	11	0	0	45.078	3745.619	0.144
	EscrowVault+EPA		4	10	17	1	16	11	0	0	136.08	3763.837	0.5
	RefundEscrow	129	3	7	11	2	9	6	0	0	16.842	5200.944	0.062
	RefundEscrow+EPA		4	7	16	1	9	5	3	3	51.049	4437.978	1216.816
	RockPaperScissors+OneWinner	66	6	4	12	0	12	11	0	0	115.825	2583.81	0.031
	RockPaperScissors+EPA		4	4	6	0	6	5	0	0	59.512	1055.916	0.029
	SimpleAuction+Ended+HB	50	6	5	26	19	7	0	0	0	4.107	15.627	7.825
	SimpleAuction+EPA		8	5	33	0	15	6	8	6	13.774	197.587	76.969
	SimpleAuction+EPA+Ended		10	5	38	0	18	7	10	8	16.771	212.258	77.892
	ValidatorAuction	260	5	8	15	7	8	2	0	0	20.69	771.325	71.718
	ValidatorAuction+EPA		6	8	22	2	7	2	5	3	55.282	929.141	926.142

the constructor is responsible for initializing the contract and setting the conditions for subsequent operations. It would be unusual to have a valid concrete state prior deployment in which the constructor lacked valid parameter values to reach all abstract initial states.

An analysis of column *#May*, which reports the number of transitions classified specifically as may (excluding must and hyper-must), shows that only a small subset of subjects exhibit exclusively may transitions. Combined with the earlier analysis, these findings suggest that must and hyper-must transitions are not only common in most contracts but also constitute the majority of transitions. This prevalence may benefit manual audits, as it allows auditors to concentrate on understanding why certain transitions deviate from the expected must or hyper-must classification.

Answer to RQ#1: Must transitions were observed in all analyzed subjects. Even excluding τ and constructor transitions, they appear in all but 3 abstractions. Notably, in 15 subjects, all transitions were classified as must, underscoring the prevalence of must transitions in the studied contracts.

*2) **RQ#2**:* For each AIR R from each subject in $B\#1$, we analyze the number of expected allowed transitions for role R (denoted as *#Exp*) and the number of must transitions (excluding constructor and τ) under the constraint $\phi_f \stackrel{\text{def}}{=} sender = R$ (denoted as $\#M$). When these values differ, we manually inspect the Solidity code to determine the cause.

If *#Exp* is greater than $\#M$, the diagram suggests that a role can execute certain transitions, but the abstraction indicates that it actually cannot. This discrepancy could reveal a bug in the code or an inconsistency in the provided diagram, making it a real issue. Conversely, if *#Exp* is lower than $\#M$, the abstraction allows more transitions than expected according to the diagram. It could indicate an access control issue if the implementation permits an unexpected role to execute restricted functions. Alternatively, this could also suggest that the diagram should explicitly include a role that, according to the code, is always authorized for that function. Among the analyzed subjects, we found that this discrepancy occurred in 3 out of 9 subjects.

Now, we discuss each subject in detail.

a) DigitalLocker: For the roles *BankAgent* and *ThirdPartyRequestor*, we observed that *#Exp* $>$ $\#M$. Specifically, the documentation states that the *BankAgent* role is authorized to call the *Terminate* method. However, this method is not classified as a must transition under the constraint $\phi_{Terminate} \stackrel{\text{def}}{=} sender = BankAgent$. This discrepancy should be reported to the project maintainers to determine whether it originates from an error in the code or in the provided diagram. For the *ThirdPartyRequestor* role, we found that *#Exp* $= 2$ but $\#M = 1$. Upon analyzing the transition not classified as must, we confirmed that there is no constraint in the code enforcing that the transition must be executed by *ThirdPartyRequestor*. This suggests either that a missing `require(ThirdPartyRequestor==msg.sender)` statement should be added to the code, or that the diagram incorrectly marks this transition as a *ThirdPartyRequestor*

AIR. Finally, for the *Owner* role, we observed that $\#Exp < \#M$. Our analysis indicates that some transitions currently assigned to the *BankAgent* role as AIR should instead be associated with the *Owner* role.

b) RefrigeratedTransp: In this subject, we observed that $\#Exp = 1$ but $\#M = 2$ for the roles *InitiatingCounterparty* and *Counterparty*. Upon inspecting the code, we confirmed that both transitions are always executable by both roles. This discrepancy suggests that either the diagram should be updated to reflect that both roles are allowed or the code should be modified to restrict each function to a single role.

c) AssetTransfer: We observed that for two out of the four roles, $\#Exp < \#M$. Upon closer inspection, we found that the *modifyOffer* method was identified as a potential access control issue. Regarding the provided diagram, this method should only be called by *InstanceBuyer*. However, the generated abstraction indicates that this function can be called by other roles (*InstanceInspector* and *InstanceAppraiser*). While the abstraction itself was accurate with the Alloy model, examining the original Solidity code revealed a logical error in the precondition of the Alloy model: it used OR instead of AND [1]. This allowed a scenario where someone could modify the offer even if they were not a buyer, as long as the offer price was non-zero. Interestingly, although this was not a real coding bug, the constraint feature was able to discover a specification error that was not detected in previous work [22].

While our evaluation focused on using constraints to check the *sender* for access control, this methodology can be used to other parameters considered relevant by the auditor.

Answer to RQ#2: The combined application of modal abstraction and the constraint transitions approach enabled the identification of an Alloy specification error in *AssetTransfer* contract that had not been previously detected by [22]. Furthermore, our method uncovered some unreported bugs in the *DigitalLocker* and *RefrigeratedTransp* contracts.

*3) **RQ#3**:* This section presents findings from applying modal abstractions to $B\#2$. We first report the findings of the author, followed by those of the auditor, and conclude with a joint assessment. The author identified unexpected behavior in the following subjects, which we analyze in detail below.

a) EscrowVault: This contract represents a modified Escrow contract akin to the RefundEscrow pattern. Funds are held in escrow until the owner confirms goal achievement by invoking the *setGoalReach* function, enabling subsequent withdrawals. Upon analyzing the abstraction, the *withdraw* method was unexpectedly classified as a may transition (i.e., a non-must transition). Using the counterexample feature described in subsection IV-B, we found that executing the *setGoalReach* method could lead to concrete states with a zero balance, thereby preventing withdrawals. This indicates that *setGoalReach* can be invoked even when the contract balance is empty. Although this is not a vulnerability, the ability to successfully call *setGoalReach* with an empty balance appears to be an unintended or unusual behavior.

b) ValidatorAuction: This contract facilitates an auction mechanism for validator slots. It involves whitelisting participants, allowing them to submit bids within a specified timeframe, and collecting deposits. Once the auction concludes, successful bidders can withdraw their deposits, minus the final slot price. If the auction fails due to insufficient participation, all bidders can withdraw their full deposits. We identified two potential issues related to protocol design and access control. First, we observed that some transitions, such as *bid*, were classified as non-must. Analysis revealed that if no bidders were added to the whitelist, the *bid* method and other transitions would become permanently inaccessible as they require bidders to be in the whitelist. While not a critical vulnerability, this represents an unexpected contract outcome. The issue stems from the *startAuction* method, which can be executed without verifying the presence of bidders. Second, this finding led to a deeper investigation, revealing another issue: the *startAuction* method can be called by any user, which deviates from the expected behavior. Ideally, this method should be restricted to specific roles (e.g., the owner) or subject to specific conditions (e.g., a time-based trigger) to ensure proper auction initiation.

Interestingly, the external auditor also identified issues in the same contracts:

a) EscrowVault: The auditor questioned why the *withdraw* transition was classified as a may-transition and hypothesized that it might be due to the contract balance reaching zero. They suggested that the tool should provide the means to confirm whether this scenario is indeed possible.[2] Upon reviewing the code, they confirmed this behavior and found the fact that the *setGoalReach* function remained enabled at zero balance to be unusual, though not indicative of a vulnerability.

b) ValidatorAuction: The auditor also noted several transitions classified as may-transitions rather than must-transitions, prompting questions such as "What concrete states prevent someone from calling *bid* in the *Started* state?", "Are there states where transitioning to either *Ended* or *Failed* is impossible?", "From *depositPending*, are there concrete states where moving to *Ended* is not possible?". Through this analysis, the auditor identified two key findings: (i) It is possible that no bidders are added to the whitelist, making the *bid* method permanently inaccessible, suggesting a poor protocol design, and (ii) If sufficient time elapses, the contract may enter a state where transitioning from *depositPending* to *Ended* becomes impossible. This occurs because the internal variable *lowestSlotPrice* is reduced to zero, causing funds to become permanently locked. This was considered as a vulnerability, as users would be unable to recover their deposits.

The results from both the author and the external auditor were largely similar but differed in emphasis. For *EscrowVault*,

[1]The Alloy model used *sender* = *InstanceBuyer* OR *InstanceBuyer*>0 instead of the original *sender* = *InstanceBuyer* AND *InstanceBuyer*>0.

[2]Since the auditor did not have access to the prototype, they were unable to refine the abstraction or use the counterexample feature.

the author considered a behavior as a design flaw, while the auditor found it unusual but not necessarily incorrect. For *ValidatorAuction*, both identified the issue preventing bidders to bid, but the auditor also found a problem involving an unrecoverable state in which funds would become permanently locked. In both cases, the key was analyzing non-must transitions. Understanding why certain transitions were classified as may was crucial in determining whether they aligned with the intended contract behavior or revealed underlying issues.

Answer to RQ#3: By leveraging the difference between may and must transitions, the author and external auditor identified unexpected behaviors in two subjects that had gone unnoticed in prior analyses. One case revealed poor design that allows unusual behavior, while the other exposed two issues, a security vulnerability and a design flaw. The integration of may and must transitions within the same abstraction provided novel insights unattainable through may abstractions alone.

VI. Threats to Validity

Internal Validity: The evaluation was conducted using a prototype designed to validate the proposed approach, which involved manually translating Solidity code into the Alloy language. This process introduces the potential risk of modeling errors. Additionally, the use of a lower-bound parameter during analysis may have led to miscategorization of some *must* transitions. To mitigate this risk, we have made all Alloy models and generated abstractions publicly available for independent verification at [7].Moreover, note that the selection predicates greatly impact in the resulting abstractions. A poor selection may lead to an unhelpful model. To mitigate this threat, we selected the same predicates as in [22].

External Validity: To reduce the risk of selection bias, we selected two widely-used benchmarks from the existing literature [38], [12], [37], [22]. While these benchmarks are representative of common patterns in smart contracts, generalizing the conclusions may be limiting. Moreover, the evaluation of **RQ#2** involved only two users, one of whom is an author. This may restrict the generalization of the findings or introduce bias. To address this, one of the users was an external auditor with no involvement in the project, and the evaluation was intended as a complementary assessment alongside our primary contributions. Future work should consider a more comprehensive human study to further validate the approach.

VII. Related Work

This work is inspired by the use of predicate abstraction for smart contract validation approach [22]. The key difference is that our work enhances the expressiveness of these models by incorporating may-must modality and constraint transitions.

Our approach is related to techniques that infer typestates [16], interfaces [25], or automata learning [13], [20], [34], [10] from program code or systems. While commonly used for verification, these methods often impose stricter constraints on program behavior than our more permissive abstractions.

Another related area of research involves generating behavioral models from execution traces [21]. However, these techniques focus on test case generation and are heavily dependent on the quality of the provided traces, differing from our approach, which aims to provide a comprehensive understanding of the system's behavior.

In recent years, several approaches have been proposed for transforming smart contracts into colored Petri Nets [17], [19]. Unlike our approach, which aims for human validation, these methods are designed to verify the smart contract using a model checker. VeriSol [38], for instance, verifies the conformance of smart contracts against state machine based workflows, ensuring that contract behavior aligns with intended logic. Although it provides strong verification capabilities, its focus lies in correctness rather than supporting comprehensive understanding for human auditors. Similarly, other related approaches [37], [36], [32] allow specifying liveness properties such as *Is it possible for a non-winning bidder to retrieve their funds?* However, they need a formal specification language (e.g., LTL). In contrast, our approach obviates the need for formalization and can potentially uncover unexpected properties through auditor inspection of the generated abstractions.

Tools like Slither [18] and Surya [4] provide insights into smart contract structure through visualizations of inheritance, control flow, and call graphs. However, they primarily focus on static analysis and fall short of capturing the dynamic behavior and potential execution sequences offered by our approach.

VIII. Conclusions

This paper introduces a novel approach to constructing modal abstractions for smart contract validation. Our method leverages manually provided predicates to generate abstractions incorporating may, must and constraint transitions. May transitions capture possible executions, while must transitions express always possible execution paths. By introducing constraint transitions, an auditor can focus on particular aspects of contract behavior, such as access control. Our empirical evaluation confirms the prevalence of must transitions in smart contract protocols and highlights how our approach aids in issue detection. Using this method, we uncovered four previously unreported issues in two established benchmarks.

Future work includes full automation of abstraction generation, mainly the manual translation of Solidity to Alloy. We believe that automated verification tools for Solidity can be helpful. We also aim to improve usability by automating predicate extraction and suggestions from code and documentation. As automation increases, a comprehensive human study will be essential to further validate the approach.

Acknowledgements

This work was funded in part by the DECO Project (PID2022-138072OB-I00) funded by MCIN/AEI/10.13039/501100011033 and by the ESF+; as well as by ANPCYT PICT 2019-1442, 2019-1973, 2021-4862, UBACYT 2020-0233BA, 2023-0134BA, CONICET PIP 1220220100470CO, and SUI Research Award.

References

[1] Solidity doc. https://docs.soliditylang.org/en.

[2] The dao smart contract. http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413, 2016.

[3] Azure blockchain workbench. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples, 2017.

[4] Surya's repository. https://github.com/ConsenSys/surya, 2018.

[5] Azure fixed code. https://github.com/blockhousetech/research/tree/master/Solidifier/evaluation/examples/azure_fixed, 2020.

[6] Smart contract security verification standard v2.0. https://github.com/ComposableSecurity/SCSVS, 2023.

[7] Alloy4pa repository. https://github.com/j-godoy/Alloy4PA, 2025.

[8] Package used by external auditor. https://github.com/j-godoy/auditor-package-modal-abstractions, 2025.

[9] Zeonodo link. https://doi.org/10.5281/zenodo.16327482, 2025.

[10] Bernhard K. Aichernig, Sandra König, Cristinel Mateis, Andrea Pferscher, and Martin Tappler. Learning minimal automata with recurrent neural networks. *Softw. Syst. Model.*, 23(3):625–655, March 2024.

[11] Adam Antonik, Michael Huth, Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. 20 years of modal and mixed specifications. *Bull. EATCS*, 95:94–129, 2008.

[12] Pedro Antonino and A. W. Roscoe. Solidifier: bounded model checking solidity using lazy contract deployment and precise memory modelling. In Chih-Cheng Hung, Jiman Hong, Alessio Bechini, and Eunjee Song, editors, *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, pages 1788–1797. ACM, 2021.

[13] Negin Ayoughi, Shiva Nejati, Mehrdad Sabetzadeh, and Patricio Saavedra. Enhancing automata learning with statistical machine learning: A network security case study. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, MODELS '24, page 172–182, New York, NY, USA, 2024. Association for Computing Machinery.

[14] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. Smart contract and defi security tools: Do they meet the needs of practitioners? In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Guido de Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Automated abstractions for contract validation. *IEEE Transactions on Software Engineering*, 38(1):141–162, 2012.

[16] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69, 2001.

[17] Wang Duo, Huang Xin, and Ma Xiaofeng. Formal analysis of smart contract based on colored petri nets. *IEEE Intelligent Systems*, 35(3):19–30, 2020.

[18] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2019.

[19] Ikram Garfatta, Kaïs Klai, Mohamed Graïet, and Walid Gaaloul. Model checking of vulnerabilities in smart contracts: a solidity-to-cpn approach. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 316–325, 2022.

[20] Bharat Garhewal and Carlos Diego N. Damasceno. An experimental evaluation of conformance testing techniques in active automata learning. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 217–227, 2023.

[21] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *2009 IEEE 31st International Conference on Software Engineering*, pages 430–440. IEEE, 2009.

[22] Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, and Sebastian Uchitel. Predicate abstractions for smart contract validation. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, page 289–299, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: Effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 557–560, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. What are the actual flaws in important smart contracts (and how can we find them)? In *Financial Cryptography*, 2020.

[25] Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 31–40, 2005.

[26] Michael Huth, Radha Jagadeesan, and David Schmidt. Modal transition systems: A foundation for three-valued program analysis. In David Sands, editor, *Programming Languages and Systems*, pages 155–169, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[27] Daniel Jackson. Alloy: A language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, aug 2019.

[28] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. *Predicate Abstraction for Program Verification*, pages 447–491. Springer International Publishing, Cham, 2018.

[29] Mahtab Kouhizadeh and Joseph Sarkis. Blockchain practices, potentials, and perspectives in greening supply chains. *Sustainability*, 10(10):3652, 2018.

[30] Jan Křetínskỳ. 30 years of modal transition systems: survey of extensions and analysis. *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, pages 36–74, 2017.

[31] K.G. Larsen and L. Xinxin. Equation solving using modal transition systems. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 108–117, 1990.

[32] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. Verisolid: Correct-by-design smart contracts for ethereum. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers*, page 446–465, Berlin, Heidelberg, 2019. Springer-Verlag.

[33] Muhammad Izhar Mehar, Charles Louis Shier, Alan Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *Banking & Insurance eJournal*, 2017.

[34] Edi Muškardin, Martin Tappler, Bernhard K. Aichernig, and Ingo Pill. Active model learning of stochastic reactive systems (extended version). *Softw. Syst. Model.*, 23(2):503–524, March 2024.

[35] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, 2020.

[36] Jonas Schiffl and Bernhard Beckert. A practical notion of liveness in smart contract applications. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[37] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu K. Lahiri, and Isil Dillig. Smartpulse: Automated checking of temporal properties in smart contracts. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 555–571. IEEE, 2021.

[38] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, volume 12031 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2019.