Model to mitigate: Using DCR graphs to prevent vulnerabilities in smart contracts

Downloaded from: https://research.chalmers.se, 2026-06-19 05:34 UTC

Citation for the original published paper (version of record):
Eshghie, M., Ahrendt, W., Artho, C. et al (2026). Model to mitigate: Using DCR graphs to prevent vulnerabilities in smart contracts. Journal of Logical and Algebraic Methods in Programming, 151.
http://dx.doi.org/10.1016/j.jlamp.2026.101131

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

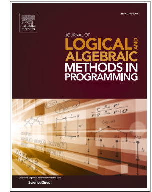
(article starts on next page)

J. Log. Algebraic Methods Program. 151 (2026) 101131

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



Model to mitigate: Using DCR graphs to prevent vulnerabilities in smart contracts

Mojtaba Eshghie [a,*], Wolfgang Ahrendt [b], Cyrille Artho [a], Thomas Troels Hildebrandt [c], Gerardo Schneider [b]

[a] KTH Royal Institute of Technology, Stockholm, Sweden
[b] Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden
[c] University of Copenhagen, Copenhagen, Denmark

ARTICLE INFO

Keywords:
Smart contract security
Dynamic condition response graphs (DCR graphs)
Vulnerability mitigation
Blockchain
Declarative business logic modeling
Decentralized applications (dApps)

ABSTRACT

We propose a 'Model to Mitigate' methodology: designing a platform-agnostic model of smart contract business logic and analyzing it before implementation. Using Dynamic Condition Response (DCR) graphs, originally developed for modeling business processes, we formally specify smart contracts and introduce a trace-conformance notion that links DCR-level guarantees to Solidity execution traces. Our method captures high-level properties such as event ordering, role-based access control, and time constraints, enabling the identification of design-rooted vulnerabilities through the discipline of explicit modeling. The DCR formalism requires developers to make concrete decisions about access control, preconditions, initial states, and event ordering-decisions that, when left implicit until implementation, are a documented source of vulnerabilities. Our analysis of real-world exploited and audited smart contracts yields six key insights, demonstrating how DCR-based modeling can enhance smart contract security by surfacing design flaws before they reach deployment. While we validate the approach on existing smart contracts with known flaws (i. e., post-implementation scenarios), the proposed methodology is applicable during design time (pre-development) .

1. Introduction

Smart contracts are self-executing programs that run on decentralized blockchain platforms like Ethereum, enabling autonomous management of digital assets according to predefined rules [1,2]. While smart contracts offer significant potential in automating complex business processes, programming languages used to write them, such as Solidity, lack explicit constructs for representing essential process-oriented concepts like roles, action dependencies, and temporal constraints. This limitation complicates the design, analysis, and verification of smart contracts, often leading to vulnerabilities and inefficiencies.

Several studies have highlighted the prevalence of vulnerabilities in smart contracts due to flaws in their design and implementation [3]. High-profile incidents including the Parity multi-signature wallet freeze ($280 million, 2017), the Nomad bridge exploit ($190 million, 2022), and the Ronin Network bridge hack ($12 million, 2024) demonstrate that many vulnerabilities originate not from low-level coding errors but from ambiguities in design decisions: who can execute which function, what preconditions must

* Corresponding author.
E-mail addresses: eshghie@kth.se (M. Eshghie), ahrendt@chalmers.se (W. Ahrendt), artho@kth.se (C. Artho), hilde@di.ku.dk (T.T. Hildebrandt), gerardo.schneider@gu.se (G. Schneider).

https://doi.org/10.1016/j.jlamp.2026.101131
Received 7 February 2025; Received in revised form 1 April 2026; Accepted 8 May 2026
Available online 22 May 2026
2352-2208/© 2026 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

hold, and what the contract's initial state should be. There is therefore a pressing need for methodologies that can specify contracts rigorously, detect design flaws early, and improve security.

Existing approaches to modeling smart contracts often focus on low-level code analysis and rarely describe the business logic of the contract at a level that reveals these design-level issues. Declarative modeling languages like Dynamic Condition Response (DCR) graphs have been successfully used in business process modeling to capture complex workflows and event dependencies [4,5]. Recently, DCR graphs have been shown to be a suitable formalism for capturing design patterns in smart contracts [6]. However, their application in mitigating vulnerabilities and design flaws is under-explored.

In this paper, we propose a *Model to Mitigate* methodology that leverages DCR graphs to model the intended business logic of smart contracts before implementation. Previous works in this field [6–8] provide evidence for DCR graphs being effective for capturing the design of the contract. By providing a formal and platform-agnostic specification, we capture and analyze high-level properties such as partial ordering of events, role-based access control, and temporal constraints that facilitate the identification of design flaws and potential vulnerabilities at an early stage, reducing the risk of costly errors post-development.

Our primary contribution is methodological rather than tool-based: the act of constructing a DCR model forces developers to make explicit decisions about access control, event ordering, preconditions, and initial states-decisions that, when left implicit until implemented in code, are a documented source of vulnerabilities [3,9]. The DCR formalism requires answering questions such as “Who can execute this event?” and “What must have happened before this event is enabled?” that might otherwise surface only after deployment. To ground these claims formally, we introduce a trace conformance notion that connects DCR-level guarantees to Solidity execution traces: if a property holds for all valid DCR execution sequences and the deployed contract conforms to the model, then no feasible execution trace violates the property. For each case study, we identify formally specifiable properties-ranging from enabledness invariants and initial-marking consistency checks to safety conditions and temporal trace properties-and discuss the practical scope of automated analysis over DCR models.

While we validate the approach on contracts with known flaws (a necessary first step to demonstrate its relevance), the methodology's value lies in the *discipline of modeling* itself. We model and analyze a variety of real-world smart contracts, including successfully exploited contracts and extensively audited ones such as the Sygma contract [10], Nomad bridge [11], Deus DAO [12], and IOU [13]. Moreover, as we discuss in Section 7, the formal semantics of DCR graphs enable lightweight property checking that can complement the manual analysis presented here, and we provide an honest assessment of both capabilities and current limitations. We verify the ‘model to mitigate’ based on case studies of already-deployed smart contracts; however, the methodology is not restricted to post-development analysis. By extracting the business logic from specifications such as decentralized protocol white papers, developers can build and refine a DCR model before writing any contract code. We suggest modeling the contract's functional and security requirements at the design stage, prior to or in parallel with implementation. This approach mitigates design flaws earlier and can inform a secure-by-construction implementation.

The contributions of this paper are as follows:

1. We demonstrate how DCR graphs can be used to model complex smart contract logic, revealing design flaws in existing implementations, and we introduce a correspondence to formally link DCR-level guarantees to Solidity execution traces.
2. We illustrate how DCR models can mitigate known vulnerabilities by enforcing correct sequences of actions and access controls, and for each case study we identify formally specifiable properties and discuss the scope and limitations of automated analysis.

The rest of the paper is organized as follows: Section 2 provides background on smart contracts and DCR graphs. Section 3 presents the correspondence between DCR models and Solidity smart contracts, including the trace conformance formalization. Section 4 reviews related work in smart contract modeling and vulnerability analysis. In Section 5, we outline our proposed methodology. Section 6 discusses the results of our analysis. Finally, Section 7 provides a discussion of the findings, including threats to validity, tool support, and practical limitations, and Section 8 concludes the paper.

2. Background

Smart contracts automate complex transactions on platforms like Ethereum but often conceal vulnerabilities within their intricate business logic [7,8,14]. Dynamic Condition Response (DCR) graphs provide a clear, declarative framework that models process dependencies, role permissions, and temporal constraints. This high-level approach exposes subtle design flaws and guides secure contract development by abstracting behavior into well-defined events and conditions. We provide a summarized background on both smart contracts specifically for Ethereum blockchain and DCR graphs as the formalism for business logic modeling [2].

2.1. Smart contracts: ethereum and solidity

While our proposed methodology and results are platform independent, we use Ethereum [2] and Solidity [15] smart contracts as an example platform for our study. Ethereum, with its built-in cryptocurrency Ether, remains the leading blockchain framework supporting smart contracts. In Ethereum, both users and contracts can receive, own, and send Ether.

Ethereum operates using transactional semantics, where a transaction represents a state-changing operation in response to a function call from an account. Transactions can be reverted due to reasons such as running out of gas (a fee paid to execute smart contract code), sending unbacked funds, or failing runtime assertions. When a transaction reverts, all its effects are undone (except for the gas paid), as if the call never occurred [2].

```
contract IOU {
  mapping(address => uint256) balances;
  mapping(address => mapping(address => uint256)) allowed;
  function approve(address _spender, uint256 _val) {
    allowed[msg.sender][_spender] = _val;
    return true; }
  function transferFrom(address _from, address _to, uint256 _val) {
    require(allowed[_from][msg.sender] >= _val && balances[_from] >= _val && _val > 0);
    balances[_from] -= _val;
    balances[_to] += _val;
    allowed[_from][msg.sender] -= _val;
    return true; }
}
```

Fig. 1. The IOU contract [13] with possible deviant behavior.

Solidity [15], the most popular programming language for Ethereum smart contracts, follows an object-oriented paradigm with state variables and functions. Each user and contract instance has a unique `address`. Addresses own Ether, can receive Ether, and can send Ether to other addresses.

Mappings in Solidity are used to create key-value associations. Fig. 1 illustrates the IOU contract that manages token balances and allowances using mappings. The `balances` mapping tracks the token balance of each address, while the `allowed` mapping records how much an owner has approved a spender to transfer on their behalf.

The `approve` function allows a user (`msg.sender`) to authorize another address (`_spender`) to spend a certain amount of tokens (`_val`) on their behalf by updating the `allowed` mapping. The `transferFrom` function enables a spender to transfer tokens from an approved address (`_from`) to a recipient (`_to`), provided that the spender has sufficient allowance and the owner has enough balance.

The `require` statement in `transferFrom` ensures that necessary conditions are met before proceeding with the transfer: the spender's allowance is sufficient (`allowed[_from][msg.sender] >= _val`), the owner's balance is adequate (`balances[_from] >= _val`), and the transfer amount is positive (`_val > 0`). If any of these conditions fail, the transaction reverts.

Core Language Features. In Solidity, functions (e. g., `public`, `external`, `internal`, `private`) and mappings (e. g., `balances` and `allowed` in Fig. 1) help express core behavior of the contract. A `require`/`assert` statement functions like a pre-/post-condition: if it evaluates to `false`, the transaction reverts immediately, preventing execution of the logic or undoing any state changes. Hence, `require`/`assert` effectively guards the function body from progressing under invalid conditions. Smart contracts also support `modifier`s to factor out recurring preconditions, and `events` to log execution for off-chain listeners. These language features (functions, mappings, condition checks, events) map naturally to events and guards in DCR graphs.

2.2. Dynamic condition response graphs

The core DCR Graph model [16] defines a process as a graph where nodes represent events, and edges represent constraints and effects between events. Let E be a finite set of events. The state of the process is given by a marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$, consisting of three sets of events:

- Ex: the set of events that have been *executed*,
- Re: the set of events that are *required* to be executed in the future or excluded,
- In: the set of events that are currently *included*.

We consider DCR Graphs with six relations that define constraints and effects between events:

1. *Condition* ($e \rightarrow\!\bullet\, e'$): For e' to be enabled, e must be excluded or have been executed at least once, i. e., $e \notin \mathsf{In}$ or $e \in \mathsf{Ex}$.
2. *Response* ($e \bullet\!\!\rightarrow e'$): If e is executed, e' is required to be eventually executed; i. e., execution of e adds e' to Re.
3. *Cancel* ($e \bullet\!\!\rightarrow\!\times\, e'$): If e is executed, e' is removed from Re, canceling any pending obligation to execute e'.
4. *Milestone* ($e \rightarrow\!\diamond\, e'$): For e' to be enabled, e must be excluded or not currently required to be executed, i. e., $e \notin \mathsf{In}$ or $e \notin \mathsf{Re}$.
5. *Include* ($e \rightarrow\!+\, e'$): If e is executed, e' is included, i. e., e' is added to In.
6. *Exclude* ($e \rightarrow\!\%\, e'$): If e is executed, e' is excluded, i. e., e' is removed from In.

To model time-sensitive processes, we extend the DCR Graphs to *timed DCR Graphs*. In timed DCR Graphs, the marking is generalized: Ex and Re become partial functions mapping events to time values. Specifically:

- $\mathsf{Ex}(e)$ denotes the time since e was last executed,
- $\mathsf{Re}(e)$ denotes the remaining time within which e must be executed again.

Time steps are natural numbers; we let ω denote infinity (i. e., an indefinite future deadline). The condition and response relations are extended with delays and deadlines, respectively. For example:

- For $e \rightarrow\!\bullet_k e'$, e' is enabled if e is excluded or was executed at least k time steps ago, i. e., $e \notin \mathsf{In}$ or $\mathsf{Ex}(e) \geq k$.

- For $e \bullet\!\!\rightarrow_d e'$, if e is executed, then $\mathsf{Re}(e') = d$, meaning e' must be executed within d time steps, be excluded permanently, or have the requirement removed by a cancel relation.

To model hierarchical processes, we make use of *DCR Graphs with sub-processes*, a formalism that extends traditional DCR graphs by incorporating sub-process containment. We define a partial sub-process containment function $sp : E \rightharpoonup E$, where $sp(e) = e'$ indicates that e' is a sub-process event containing e. This relation is constrained to be acyclic to prevent infinite containment.

In DCR Graphs with data, events are associated with values upon execution, either from environmental input or as results of assigned expressions. We introduce a function Va added to the marking to record these values. Each relation is extended with a Boolean guard expression; if the guard evaluates to false in the current marking, the relation is disregarded. Let $\mathsf{Exp_E}$ be a set of expressions, with $\mathsf{BExp_E} \subseteq \mathsf{Exp_E}$ being the Boolean expressions. Each event $e \in E$ can be associated with an expression in $\mathsf{Exp_E}$. The data-assignment function $D : E \to \mathsf{Exp_E} \uplus \{?\}$ defines events as:

- *Computation events* if $D(e) \in \mathsf{Exp_E}$,
- *Input events* if $D(e) = ?$.

For $d \in \mathsf{Exp_E}$, let $[[d]]_M$ denote the evaluation of expression d in marking M. Event identifiers can be used as basic expressions, evaluating to their current assigned value: $[[e]]_M = \mathsf{Va}(e)$.

Notation and preliminaries. We write $f : A \rightharpoonup B$ to denote a *partial function* from A to B, i.e., f may be undefined for some elements of A. Let $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$, where ω represents an indefinitely large value (used to indicate that an event has never been executed or that a delay constraint is effectively unbounded). Similarly, let $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$, where ∞ represents an indefinite future deadline (the event must eventually occur, but with no fixed time bound). Let V be a domain of data values (e.g., integers, addresses, byte arrays, or structured records) that may be associated with events upon execution. Let R be a finite set of *roles* representing participants who may execute events (e.g., `admin`, `user`, `relayer`, or a wildcard "any"), and let A be a finite set of *action names* corresponding to identifiable operations (e.g., Solidity function names such as `approve` or `transferFrom`).

We now formally define the timed DCR Graphs with sub-processes, data, and roles.

Definition 1. A *timed DCR Graph with sub-processes, data, and roles* is a tuple $(E, sp, D, M, \rightarrow\!\bullet, \bullet\!\!\rightarrow, \bullet\!\!\rightarrow\!\!\times, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\%, L, l)$ where:

1. E is a finite set of *events*,
2. $sp : E \rightharpoonup E$ is the *sub-process* containment function,
3. $D : E \to \mathsf{Exp_E} \uplus \{?\}$ is the *data-assignment* function,
4. $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$ is the *timed marking with data*, where:
   - $\mathsf{Ex} : E \rightharpoonup \mathbb{N}_\omega$ records, for each executed event, the time (in discrete steps) since its last execution;
   - $\mathsf{Re} : E \rightharpoonup \mathbb{N}_\infty$ records, for each pending event, the remaining time within which it must be executed (or ∞ if no deadline applies);
   - $\mathsf{In} \subseteq E$ is the set of currently included events;
   - $\mathsf{Va} : E \rightharpoonup V$ records the data value most recently associated with each event (undefined for events that have not yet been executed with a value).
5. $\rightarrow\!\bullet \subseteq E \times \mathbb{N}_\omega \times \mathsf{BExp_E} \times E$ is the *guarded timed condition relation*,
6. $\bullet\!\!\rightarrow \subseteq E \times \mathbb{N}_\infty \times \mathsf{BExp_E} \times E$ is the *guarded timed response relation*,
7. $\bullet\!\!\rightarrow\!\!\times, \rightarrow\!\diamond, \rightarrow\!+, \rightarrow\!\% \subseteq E \times \mathsf{BExp_E} \times E$ are the *guarded cancel, milestone, include, and exclude relations*, respectively,
8. $L = \mathcal{P}(R) \times A$ is the *set of labels*, with R and A being sets of roles and actions,
9. $l : E \to L$ is a *labeling function* mapping events to labels.

Notation remark. When we write $e \rightarrow\!\bullet_k e'$ with a subscript k, this is shorthand for the tuple $(e, k, true, e') \in \rightarrow\!\bullet$, emphasizing the delay parameter. The guard component defaults to $true$ when omitted. Similarly, $e \bullet\!\!\rightarrow_d e'$ denotes $(e, d, true, e') \in \bullet\!\!\rightarrow$ with deadline d. For the unguarded, untimed relations ($\rightarrow\!+$, $\rightarrow\!\%$, $\rightarrow\!\diamond$, $\bullet\!\!\rightarrow\!\!\times$), we write $e \rightarrow\!+ e'$ as shorthand for $(e, true, e') \in \rightarrow\!+$, etc.

Enabledness. The condition ($\rightarrow\!\bullet$) and milestone ($\rightarrow\!\diamond$) relations constrain when events can be executed. Informally: Formally, an event e is enabled in marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$ and can be executed by role $r \in R$ if:

1. $l(e) = (R', a)$ with $r \in R'$,
2. $e \in \mathsf{In}$,
3. All conditions are met: For all $(e', k, g, e) \in \rightarrow\!\bullet$, if $e' \in \mathsf{In}$ and $[[g]]_M$ is true, then $\mathsf{Ex}(e') \geq k$,
4. All milestones are met: For all $(e', g, e) \in \rightarrow\!\diamond$, if $e' \in \mathsf{In}$ and $[[g]]_M$ is true, then $\mathsf{Re}(e')$ is undefined,
5. Either e is not contained in a sub-process, or $sp(e)$ is enabled and can be executed by role r.

Note that here, e is in the target position of each relation tuple.

Roles, Actions, and Labels. In our DCR specification, each event e is labeled by (R', a) where $R' \subseteq R$ is the set of roles authorized to execute e, and a is the named action (mapping to e. g., `approve` and `transferFrom` in Fig. 1). These labels help track which participant(s) can fire an event and how that event is referenced in the system design. In smart contracts, common roles might be `admin`, `owner`, `relayer`, or even "any user," while actions map directly to function calls (or sub-calls in a nested call stack).

Execution. When an event e is executed in marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$, the marking updates to $M' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}', \mathsf{Va}')$. The update depends on the response, include, and exclude relations, and the computation expressions assigned to events. In DCR Graphs with sub-processes, executing an event e can trigger the execution of its containing sub-process event $e' = sp(e)$, which may in turn trigger its own containing sub-process, and so on. This cascade is finite due to the finite number of events. Informally, a sub-process event e' is executed if one of its immediate sub-events e (i. e., $sp(e) = e'$) is executed, and after e's execution, all immediate sub-events of e' are either not included or not pending; in this case, the sub-process is said to be accepting [4,5].

An event e is unique in the set E, so In, Ex, and Re refer to different properties of the same underlying event. e cannot be 'included' and 'excluded' at the same time. If a particular activity must occur multiple times (i. e., multiple instances), one can use the timed repetition constructs outlined in our prior works that reset $\mathsf{Ex}(e)$ after each execution [6] or may model each instance as a separate event (e_1, e_2, etc.).

3. Correspondence between DCR models to solidity smart contracts

We present the correspondence between the elements of a DCR graph model and the constructs found in typical smart contracts, particularly those written in Solidity for the Ethereum Virtual Machine (EVM) [17]. This mapping provides the foundation for using DCR graphs as formal specifications for smart contract behavior, bridging the gap between the abstract process model and the concrete implementation. A DCR graph comprises both *static structure* (the set of events E, the relations, and the role/action labels) and *dynamic state* (the marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$). The figures in this paper primarily show a brief visual view of the models, where events are represented as boxes, relations as directed edges, and roles as annotations. This view corresponds to the Solidity code's functions and modifiers. The marking evolves during execution: each event execution updates Ex, Re, In, and Va according to the effect relations. When we speak of "the DCR model" of a contract, we mean the structure together with an *initial marking* M_0 that captures the contract's state at deployment. In M_0, typically $\mathsf{Ex}(e)$ is undefined for all events (none have been executed), Re may contain events with initial obligations (if any), In_0 specifies which events are initially available, and Va is undefined or initialized to default values matching Solidity's default state variable initialization.

The DCR graphs in this section use the standard visual notation from DCRGraphs.net [18]. Each *box* represents an event, with the event name displayed inside. The *role(s)* authorized to execute an event appear above the box (e.g., "admin" in Fig. 4). An event with no role annotation is executable by any participant. In the marking visualization (if shown), a *green border* or checkmark indicates the event is included ($\in$ In), a *blue checkmark* indicates it has been executed ($\in$ Ex), and an *exclamation mark* (!) indicates it is pending ($\in$ Re). Guards and time delays, when present, appear as annotations on the arrows. For brevity, not all marking details are shown in every figure; the structural aspects (events, roles, relations) are the focus.

DCR Event. A DCR event typically corresponds to a `public` or `external` function call in a Solidity contract. Executing the event in the DCR model mirrors invoking the corresponding function via a blockchain transaction initiated by an Externally Owned Account (EOA), a non-contract account on Ethereum blockchain. For instance, in a casino contract, a `createGame` event in the DCR model would map to the corresponding `createGame(bytes32 hash)` function in the Solidity code.

DCR Role. Roles assigned to DCR events map directly to access control mechanisms within the smart contract. This is commonly implemented using Solidity `modifier`s or `require` statements that check `msg.sender` against authorized addresses (e.g., a `byOp` modifier checking against an `operator` address, corresponding to an *Operator* role assigned to the relevant events in the DCR model). Events without explicit roles in the DCR model are generally callable by any authorized user, potentially subject to other state-based conditions.

DCR Marking (Contract State). The marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$ collectively represents the current state of the smart contract stored on the blockchain's distributed ledger.

- *Values* (Va): The data values associated with events map directly to contract state variables (e.g., `uint`, `address`, `bytes32`, `enum` types representing contract state, operator addresses, or stored hashes). DCR input/computation events model the update of these variables (e.g., storing a hash upon game creation). DCR input/computation events model the update of these variables (e.g., `hashedNumber = hash` in `createGame`).
- *Included* (In): The set of included events represents which contract functions are logically permitted to execute according to the contract's current business process state. In Solidity, this is often implicitly managed by control flow logic or state variables checked by modifiers. The DCR model makes this explicit via the In set and include/exclude relations.
- *Required/Pending* (Re): Pending DCR events represent obligations or actions that must (or should) eventually occur according to the protocol. This often maps to state variables or flags indicating required next steps, potentially coupled with deadlines stored in state variables and checked against `block.timestamp`. The response relation •→ frequently establishes such pending states (e.g., the obligation to call a *decideBet* function after *placeBet* is executed, modeled by a •→ relation in the corresponding DCR graph). Time incentivization patterns [6] heavily rely on this concept.
- *Executed Time* (Ex): The time since an event's execution, or simply the fact that it has been executed, maps to stored timestamps (`block.timestamp`), block numbers (`block.number`), or boolean flags in the contract state. This information is used when DCR conditions →• depend on whether or when a previous action occurred.

DCR Enabledness (Conditions). The enabledness criteria for a DCR event, determined by relations like condition →• and milestone →◇, correspond directly to the pre-condition checks performed at the entry points of a Solidity function, typically implemented using `require(condition, ‘‘error message’’);` statements or within `modifiers`. A function call triggers a transaction revert if these conditions are not met at runtime, mirroring the inability to execute a non-enabled DCR event.

DCR Relations (State Effects). The effect relations (→+, →%, •→, •→×) model the state changes and side effects resulting from the successful execution of a smart contract function's body.

- Include →+ / Exclude →% map to changes in state variables that logically enable or disable the preconditions for future function calls (e.g., updating a `State` enum variable effectively includes/excludes functions guarded by a state-checking modifier; explicit state transitions can be modeled using include/exclude relations as described in previous work [6]).
- Response •→ maps to setting state that signifies an obligation or expected follow-up action is now active.
- Cancel •→× maps to clearing flags or state variables indicating a previous obligation has been fulfilled or voided.

DCR Execution Step vs. Transaction Success. A successful DCR execution step $M \xrightarrow{e,[v]} M'$ corresponds to a successful (non-reverted) blockchain transaction invoking function e. This transaction updates the contract's storage on the blockchain, transitioning its logical state from M to M'.

DCR Non-Enabled Execution Attempt vs. Transaction Revert. Attempting to execute a DCR event e when it is not enabled in the current marking M corresponds directly to a blockchain transaction calling the associated function e that fails its `require` checks (or modifier conditions). This results in the transaction being reverted, leaving the blockchain state unchanged (as in M), except for the consumption of gas paid by the initiator.

DCR Time Constraints. Timed DCR relations, such as delays specified on condition relations →• or deadlines on response relations •→, map to Solidity logic that explicitly checks `block.timestamp` or `block.number` against stored time values within `require` statements. This enforces timing constraints crucial for patterns such as automatic deprecation or time-based incentivization [6].

This explicit correspondence allows the DCR graph to serve as a precise, high-level behavioral specification for a smart contract. It abstracts away low-level EVM details while formally capturing the essential control flow, data dependencies, access control rules, and temporal constraints governing the contract's interactions. Understanding this mapping is key to interpreting the contract models presented in Section 6, and it forms the basis for formal analysis and verification techniques, such as the runtime monitoring approach using HighGuard mentioned in Section 4.

3.1. Trace conformance

Here we formalize what it means for a deployed contract to *conform* to its DCR model. We introduce three lightweight definitions that rely only on the DCR semantics already given in Section 2.2 and on standard notions of blockchain transaction traces.

Contract–Model Mapping. Let $G = (E, sp, D, M_0, \rightarrow\bullet, \bullet\rightarrow, \bullet\!\rightarrow\!\times, \rightarrow\diamond, \rightarrow+, \rightarrow\%, L, l)$ be a timed DCR Graph with sub-processes, data, and roles (Definition 1), intended as the specification of a Solidity contract C. We assume a *contract–model mapping* $\mu = (\mu_f, \mu_r)$ where:

- $\mu_f : Func(C) \rightharpoonup E$ is a partial function from the public/external functions of C to DCR events, and
- $\mu_r : Addr \rightarrow R$ maps an Ethereum address (specifically, the `msg.sender` of a transaction) to a role in R.

Functions of C not in the domain of μ_f (e.g., pure view functions or internal helpers with no corresponding DCR event) are outside the scope of conformance checking. The mapping μ formalizes the informal correspondences described in the preceding paragraphs of this section (DCR Event ↔ Solidity function, DCR Role ↔ `msg.sender`).

Definition 2 (Solidity Execution Trace). A *Solidity execution trace* of contract C is a finite sequence

$$\sigma \;=\; \langle\, tx_1,\ tx_2,\ \dots,\ tx_n \,\rangle$$

where each $tx_i = (f_i, s_i, v_i, t_i)$ records: a successfully executed (non-reverted) call to function $f_i \in Func(C)$ by sender $s_i \in Addr$, with input/output values v_i and block timestamp t_i (with $t_i \le t_{i+1}$). Reverted transactions are excluded, mirroring the DCR semantics in which a non-enabled event simply cannot execute (cf. the "Transaction Revert" paragraph above).

Definition 3 (Trace Projection). Given a contract–model mapping $\mu = (\mu_f, \mu_r)$, the *projection* of a Solidity trace σ onto G is the subsequence

$$\pi_\mu(\sigma) \;=\; \big\langle\, (e_j,\ r_j,\ v_j,\ t_j) \,\big\rangle_{j \in J}$$

obtained by retaining only those transactions tx_i for which $\mu_f(f_i)$ is defined, and setting $e_j = \mu_f(f_i)$, $r_j = \mu_r(s_i)$, $v_j = v_i$, and $t_j = t_i$.

Definition 4 (Trace Conformance). A Solidity execution trace σ of contract C *conforms* to DCR model G under mapping μ if, for the projected trace $\pi_\mu(\sigma) = \langle (e_1, r_1, v_1, t_1), \dots, (e_m, r_m, v_m, t_m) \rangle$, there exists a sequence of markings $M_0, M_1, \dots, M_m$ such that for every $1 \le j \le m$:

1. event e_j is *enabled* in marking M_{j-1} and can be executed by role r_j (per the enabledness conditions in Section 2.2), and
2. $M_{j-1} \xrightarrow{e_j,[v_j]} M_j$ according to the execution semantics of G.

We say that contract C *conforms* to G (under μ) if every feasible Solidity execution trace of C conforms to G.

This definition yields a conditional guarantee that connects the DCR-level analysis to implementation-level security:

> *If (i) a property φ holds for all valid execution sequences of the DCR model G (e.g., "event* process *is never enabled without prior execution of* initialize*"), and (ii) the deployed contract C conforms to G under μ, then no feasible execution trace of C violates φ (projected through μ).*

The practical implication is that our "model to mitigate" claims are *conditional* on conformance: the DCR model rules out undesirable behaviors *at the design level*, and these guarantees transfer to the implementation to the extent that the implementation faithfully realizes the model. Establishing conformance itself can be achieved through manual code review guided by the mapping μ, runtime monitoring (e.g., HighGuard [7] checks each transaction against the DCR specification at runtime), or certified code generation from DCR models.

Scope and limitations. This formalization captures *inter-transaction* ordering and access control, which is the level at which our case studies identify vulnerabilities. It does not model *intra-transaction* behavior such as reentrancy within a single call stack; such low-level properties are better addressed by complementary static analyzers [19,20]. The mapping μ is currently constructed manually as part of the modeling methodology (Section 5); automating its extraction from Solidity ASTs is future work. While this formalization facilitates the future work on automated contract generation from DCR models, we sometimes break this down to model-specific intra-transaction properties as well, which is discussed in the upcoming subsection.

3.2. Challenges in model correspondence

While the correspondence described above covers the common case where each Solidity function maps to a single DCR event, several recurring situations require additional modeling workarounds. We discuss three such challenges and the solutions we have adopted.

3.2.1. Keeping the effects of excluded events

A challenge arises when an event has outgoing effect relations (e.g., responses, includes) but may be dynamically excluded during execution. If the event is excluded before it fires, its effects never occur, but if it was previously executed, some of those effects (e.g., pending responses it created) may still be active in the marking.

Solution: Wrapper events. We address this by introducing *wrapper events* that encapsulate the effect-producing logic. The wrapper event remains included and is responsible for managing the effects, while the original event can be excluded without orphaning its prior effects. This pattern separates the "can this action occur?" question (controlled by inclusion of the main event) from the "what are the consequences of past occurrences?" question (managed by the wrapper).

3.2.2. Modeling post-conditions

Solidity functions may have post-conditions (invariants that must hold after execution) that are distinct from pre-conditions (checked via `require`). DCR's native semantics focus on enabledness (pre-conditions) and effects, but do not directly express post-conditions.

Solution 1: Strengthened guards. We can encode the post-condition into the guard of the event itself, using the conjunction of the pre-condition and the weakest predicate that ensures both the function body and post-condition are satisfiable. This approach is elegant but can be difficult to compute, especially when external calls are involved.

Solution 2: Helper events via sub-processes. A more practical approach models the function as a *sub-process* containing two helper events: one for the main body and one for the exit point. A condition relation links the body to the exit-point event, and the exit-point event has a response relation (liveness obligation) from the outset. The sub-process is in an accepting state only when the exit-point event has been executed, signaling successful completion including post-condition satisfaction. This approach uses multiple DCR events for a single Solidity function but maintains a clear correspondence.

3.2.3. Dynamic paths inside functions

Some Solidity functions contain conditional logic (e.g., `if`/`else`, `try`/`catch`) that leads to different state effects depending on runtime conditions. Modeling such functions as a single DCR event loses this branching structure.

Solution: Branch decomposition. We decompose the function into multiple DCR events, one per significant execution path, grouped within a sub-process. Mutual exclusion between paths is modeled via exclude relations, and guards on the events capture the branching conditions. This is illustrated in the Sygma `executeProposal` case (Section 6.2.1), where the try/catch branches correspond to distinct sub-events with different effects on `proposal._status`.

The trade-off is increased model complexity, but this complexity reflects genuine behavioral complexity in the code. Hiding it would obscure exactly the kind of design flaw (e.g., redundant state writes on failing paths) that the modeling process is meant to surface.

4. Related work

We categorize related works into three areas: modeling formalisms for smart contracts, correct-by-design approaches, and bug hunting and auditing for smart contracts.

Modeling Formalisms for Smart Contracts. Modeling formalisms are either business process-level or implementation-level [6,21]. Business process-level formalisms focus on high-level contract behavior, ignoring technical details; DCR graphs are an example. State-transition systems compatible with Solidity are used in languages like Obsidian and Bamboo [6,22,23], with validation via model checkers and temporal logic properties [24]. Petri Nets provide intuitive graphical models [25,26]; the BIP framework models interactions between contracts and users [27]. Timed automata capture temporal behaviors and time constraints [6,28,29]. Probabilistic transition systems, like Markov decision processes, represent nondeterministic execution and user behaviors [30–32].

A prior study explored the expressiveness of DCR graphs for modeling certain smart contract interactions [6], but it did not address how this modeling approach can be integrated into a pre-development security-focused and vulnerability-mitigation workflow. Our contribution goes beyond that initial demonstration by proposing specific methodological steps to (a) detect design-level flaws (e. g., entangled access control, ambiguous initial states) and (b) mitigate known vulnerability patterns (e. g., misuse of allowances, bridging misconfigurations). We also emphasize how DCR modeling can be integrated pre-implementation for a correct-by-design paradigm and used post-implementation for thorough audits.

Correct-by-Design. Suvorov and Ulyantsev [33] propose a correct-by-construction method using Finite State Machines (FSMs) and LTL specifications, transforming models into Solidity code. VeriSolid [24] offers a framework for designing and verifying Ethereum contracts as transition systems, with automated verification using CTL and automatic Solidity code generation. FLAMES [34] uses domain-adapted LLMs to synthesize executable `require` statements for contract hardening without needing vulnerability labels, addressing the gap between design-level specifications and deployable defenses. While these approaches contribute to modeling and verification, DCR graphs provide declarative modeling with adjustable detail levels and bridge business processes with implementation.

Unlike automated verification frameworks (e. g., Coq, TLA^+, and Move Prover [35–37]) or low-level static analyzers [19,20], our approach targets the high-level business logic where a certain class of functional vulnerabilities often originate [7,9,14]. This complements formal verification, as DCR-based models can highlight workflows, roles, and design patterns that implementation-level verifiers might overlook. Consequently, we see our method as bridging the gap between high-level process modeling and formal correctness checks in the lower-level code.

Bug Hunting and Auditing for Smart Contracts. [38] propose an on-chain bug bounty system using multi-version programming to form a *Hydra* contract with multiple versions managed by a proxy. However, it has limitations like increased gas consumption, lack of opcode support, and no code delegation management. Our approach uses tools like HighGuard [7] and XploGen [8] to model contracts and formally verify correctness, offering a more comprehensive solution. Zhou et al. [39] classify exploits and defense strategies, highlighting that incorporating defenses is insufficient without holistic system design. Raven [40] complements offensive analysis by mining defensive invariant categories from reverted Ethereum transactions, discovering six previously unreported categories of working on-chain defenses. Our current work argues that modeling using DCR graphs pre-development can mitigate flaws and vulnerabilities. We verify our ‘model to mitigate’ method using already developed or exploited contracts. Whereas analysis and auditing tools work post-development on source code or transaction data.

5. Proposed method

Our methodology involves iterative modeling and manual expert analysis. The modeling process entails understanding contract logic and interactions, then visualizing this logic using DCR graphs (Section 2.2) with the DCR visual modeling tool [18]. Modeling can be challenging for complex, interdependent multi-contract systems. We gained significant insights into their operational and security aspects, presented in the results section.

5.1. Modeling methodology

To construct DCR graphs for each contract, a seven-step methodology (visualized in Fig. 2):

1. **Contract Logic Identification:** Analyze the smart contract code, documentation, and inheritance hierarchies to identify functions, modifiers, state variables, and events.

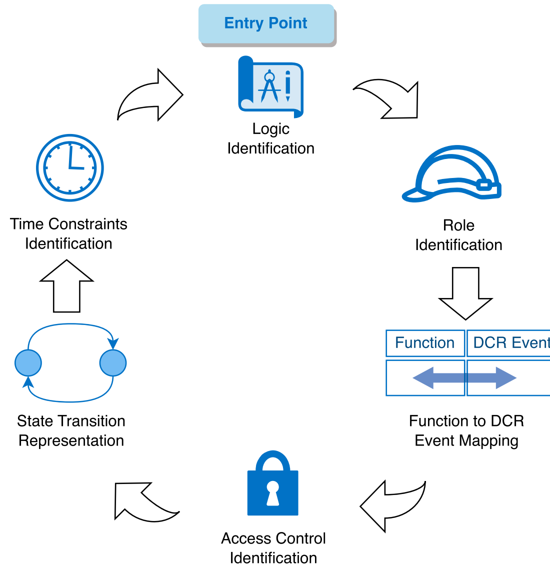


Fig. 2. Modeling methodology used in "model to mitigate".

2. **Identifying Roles:** Extract explicit roles (e. g., `admin`, `operator`, `player`) that execute contract functions.
3. **Mapping Functions to DCR Events:** Map `public` and `external` functions to events. Consider `internal` and `private` functions as sub-activities or include them via inclusion →+ / exclusion →% and response •→ relations if they affect multiple `public`/`external` functions. When a single function contains significant branching logic or multiple distinct state-changing paths (e.g., `try`/`catch` blocks), it may be decomposed into multiple DCR events grouped within a sub-process, as discussed in Section 3.2.3.
4. **Modeling Access Control:** Assign roles to events to reflect role-based access control, ensuring only permitted roles can execute specific activities. This step consists of putting (2) and (3) together. Furthermore, we also model implicit access control mechanism of the contract.
5. **Representing State Transitions:** Use relations between activities to model function availability. Represent modifiers and `require` statements using condition relations →• and guards in the DCR graph to capture conditional logic and dependencies.
6. **Incorporating Time Constraints:** For time-based features, use delays and deadlines in the DCR graph according to design patterns from previous work [6].
7. **Iterative Refinement:** Refine models by adding inferred covert roles from `require` statements and validate against intended behavior. Use DCRGraphs.net [18] to visually create and simulate models, aiding in logic validation and issue identification.

5.2. Pre-Development vs. Post-development workflows

Our methodology supports two complementary workflows, depending on when the DCR model is constructed:

Pre-development (modeling intended behavior). In this workflow, the modeler constructs a DCR graph from requirements documents, protocol whitepapers, or design discussions *before* the Solidity implementation exists. The modeling process itself surfaces design flaws. The DCR formalism requires explicit decisions about which roles can execute each event, what preconditions must hold (condition relations), what obligations are created (response relations), and what the initial marking M_0 is. Ambiguities or unresolved questions in the informal specification become apparent when they cannot be or are not expressed in the model. The resulting DCR graph then serves as a *specification* that guides implementation and against which the code can be validated.

Post-development (modeling actual code). In this workflow, the modeler reverse-engineers a DCR graph from an existing smart contract implementation. Flaws are identified in two ways: *(i) Model–intent mismatch:* The extracted model is compared against the intended behavior (from documentation, audits, or domain knowledge). Discrepancies indicate potential vulnerabilities. *(ii) Simulation anomalies:* Simulating the model may reveal unexpected enabled events, unreachable states, or missing preconditions that were not apparent from code inspection alone.

In our case studies (Section 6), we apply the post-development workflow to contracts with known vulnerabilities. This retrospective analysis validates that the modeling process *would have* surfaced the flaw, even though the vulnerability was historically discovered post-mortem. The value of the methodology lies in the discipline of modeling, where explicit decisions required by DCR force consideration of exactly the questions (access control, ordering, initialization) whose implicit or incorrect handling caused the vulnerabilities.

```
function adminAddRelayer(address relayerAddress) external {
  require(
    !hasRole(RELAYER_ROLE, relayerAddress),
    "addr already has relayer role!"
  );
  require(_totalRelayers() < MAX_RELAYERS, "relayer limit reached");
  grantRole(RELAYER_ROLE, relayerAddress);
  emit RelayerAdded(relayerAddress);
}
```

Fig. 3. *adminAddRelayer* function from Sygma contract [10].

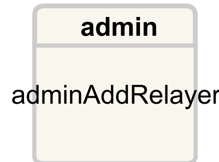


Fig. 4. A straightforward role-based access control in the DCR model for `adminAddRelayer`, requiring the `admin` role.

6. Results

In our case studies, we model and analyze DEUS DAO [12], Sygma [10], Nomad Bridge [11], and IOU [13]. This combination was carefully chosen to cover both cases of complex real-world contracts that are extensively audited, such as Sygma (audits available in [41,42]) and DEUS (audits available in [43]) as well as successfully exploited contracts (Nomad, DEUS DAO, and IOU).

6.1. Discovery of design flaws

As a result of the modeling process involving the *Sygma* contract [10], we found two flaws in the design of the contract: entangled access control and unclear initial state. These flaws were previously unnoticed by the audits [41,42]. Furthermore, beyond manual analysis, for each case study we outline here what classes of properties are relevant for each case.

6.1.1. Entangled access control

Flaw. In the Sygma contract [10], the function `adminAddRelayer` (Fig. 3) is intended to be callable only by an administrator. However, the code does not include an explicit permission check within `adminAddRelayer` itself. Instead, the check is deferred to a call inside its body, requiring the developer to maintain a second conditional revert to enforce the same access policy. Although the net effect is eventually the same-reverting if the caller is not the admin-this design suffers from two drawbacks:

- Access control checks become *entangled* in multiple function calls, increasing the likelihood of missing or misplacing them during maintenance or updates.
- Reverting a transaction later in the call chain wastes gas for the caller, who pays for execution up to the point of failure. An earlier, unified check would revert sooner and save gas.

Modeling and Mitigation. The flaw was surfaced during model construction. Creating the DCR event for `adminAddRelayer` required assigning a role-the formalism demands this explicitly. We assigned `admin`, reflecting the function's intended access control (Fig. 4). This immediately raised the question: "Where in the Solidity code is this role actually enforced?" Examining the code revealed that `adminAddRelayer` itself contains no `require(msg.sender == admin)` check; the authorization is deferred to `grantRole`, called internally. The DCR model thus made the *intended* access control explicit, exposing the *actual* implementation's entangled structure. Translating the model's requirement back into Solidity suggests placing the role check directly in the `adminAddRelayer` function or in a dedicated access-control modifier. Consolidating role verification into a single place avoids the maintenance overhead of duplicate checks and allows earlier reversal of transactions (thus reducing gas costs on failing calls).

Formally Specifiable Properties. The intended property is an *enabledness invariant* where for every reachable marking M, the event *adminAddRelayer* is enabled only if the executing role is `admin`. In the DCR model (Fig. 4), this holds by construction because the event's label restricts execution to the `admin` role. A model checker can verify that no alternative execution path enables the event for an unauthorized role.

> **Insight:** Consolidating authorization into a single, direct check (as naturally represented in the DCR model) makes both the implementation and future updates clearer and more efficient. Distributing permission checks across multiple functions (i.,e., deferring role verification to deeper calls) creates an *entangled access control* structure which complicates debugging and results in unnecessary gas consumption due to redundant revert checks.

```
enum ProposalStatus { Inactive, Active, Passed, Executed, Canceled }
```

Fig. 5. *ProposalStatus* enum from Sygma contract.

6.1.2. Unclear initial state

Flaw. The Sygma contract contains a part for governance protocol that uses decentralized governance proposals [44]. Decentralized governance proposals have emerged as a mechanism for enabling community-driven decision making in blockchain systems. In these schemes, stakeholders submit proposals that undergo clearly defined state transitions, typically moving from an initial inactive state, to active discussion, and eventually to a conclusive decision (e.g., passed or canceled) [45]. The implementation of Sygma, uses a Solidity `enum` to show the state the proposal is at (Fig. 5).

Modeling and Mitigation. Constructing the DCR model required specifying the initial marking M_0, including which events are initially included (In_0) and what values state variables hold at deployment. For a governance system with proposal states, this means deciding which transitions are available from the outset. The Solidity code, however, provided no explicit initialization-it relied on the default value of a Solidity `enum` (the first item, `Inactive`). The modeling process forced us to make this decision explicit, revealing that the intended initial state was implicit and potentially ambiguous. As presented in previous work [6], modeling the order of actions with built-in DCR semantics is more straightforward than modeling the order of actions through finite state machines, but capturing the correct event ordering demands starting from the correct available-for-execution set of activities. Although relying on the default enum value is not a vulnerability per se, it makes the implementation less understandable and harder to review. Explicit initialization of the abstract state `enum`-mirroring the explicit M_0 required by the DCR model-would enhance clarity and intent.

While unclear initial state may appear to be a purely syntactic coding issue detectable by automated syntactic analysis tools, real-world incidents demonstrate that the problem is fundamentally one of design-level ambiguities about a contract's intended starting configuration. In the 2017 Parity multi-signature wallet incident, the shared library contract was deployed with a valid initialization function (`initWallet`) already defined in the code, yet this function was never called during deployment, leaving ownership unset at its default value [46,47]. An anonymous user exploited this by calling the unguarded initialization function, claiming ownership of the library, and subsequently invoking the self-destruct operation leading to permanently freezing approximately $280 million in Ether across 587 wallets [46]. More recently, in August 2024, the Ronin Network bridge was exploited for $12 million after a contract upgrade defined two initialization functions (`initializeV3` and `initializeV4`), but only the latter was invoked during deployment [48]. The omitted `initializeV3` was responsible for setting the `_totalOperatorWeight` variable, which determined the minimum vote weight required to authorize cross-chain withdrawals. Because this variable defaulted to zero, the bridge's multi-signature verification was effectively disabled, allowing any single arbitrary signature to approve a withdrawal [48]. In both cases a syntactic rule such as "all variables must be explicitly initialized" would not have mitigated the incident: the initialization logic *existed* in the source code but was omitted from the deployment workflow. The underlying problem was the absence of an explicit specification of what the system's initial state *should* be and which functions should be enabled at startup. A DCR model's initial marking $M_0 = (\mathsf{Ex}_0, \mathsf{Re}_0, \mathsf{In}_0, \mathsf{Va}_0)$ addresses this by requiring the modeler to declare which events are initially included, which are pending, and what values state variables hold before any interaction occurs, the modeling process surfaces mismatches between the intended and actual initial configuration as a first-class design concern. These incidents underscore that unclear initial state is not merely a matter of coding style but a documented root cause of vulnerabilities that the discipline of explicit DCR modeling can mitigate.

Formally Specifiable Properties. The property is an *initial-marking consistency check*: the set of initially included events In_0 and initially pending events Re_0 must match the intended starting configuration (e.g., only proposal-creation events are enabled, not execution or cancellation events). This is trivially inspectable via the DCR model's initial marking and can be checked by enumerating the enabled events at M_0.

> **Insight:** Relying on implicit initializations—such as the default value of an enum—to set a contract's starting state can obscure the intended behavior. An explicit initialization improves clarity for developers and auditors, ensuring that the contract's initial conditions are unambiguous and aligned with the business logic presented by the DCR model of the contract.

6.2. Efficiency improvement

As a result of modeling and analyzing the DCR model of the Sygma [10] contract, we identified a part of the implementation where the logic of the contract could be implemented in a different way to reduce the gas consumption of the transactions.

6.2.1. Reducible gas consumption

Flaw. In the `executeProposal` function of the Sygma contract, part of the logic (Fig. 6) unconditionally sets `proposal._status` to `ProposalStatus.Executed` and then relies on additional condition checks and event emissions. This sequence introduces unnecessary writes to storage and extra conditional jumps. Moreover, the `catch` branch always emits an event and updates the proposal status, even in cases where a full revert would be cheaper and clearer (e.g., if `revertOnFail` is `true`). Consequently, transactions incur higher gas costs than necessary.

```
proposal._status = ProposalStatus.Executed;
IDepositExecute depositHandler = IDepositExecute(handler);
if (revertOnFail) {
    depositHandler.executeProposal(resourceID, data);
} else {
    try depositHandler.executeProposal(resourceID, data) {} catch (
        bytes memory lowLevelData
    ) {
        proposal._status = ProposalStatus.Passed;
        emit FailedHandlerExecution(lowLevelData); return;
    }
}
```

Fig. 6. Original (reducible) part of `executeProposal` function from Sygma contract.

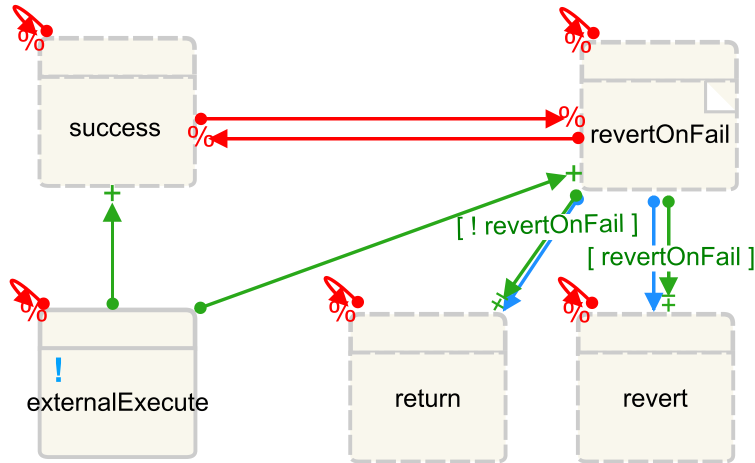


Fig. 7. Inferred semantics of `executeProposal` function, revealing redundant writes and conditionals.

```
IDepositExecute depositHandler = IDepositExecute(handler);
try depositHandler.executeProposal(resourceID, data) {
    proposal._status = ProposalStatus.Executed;
} catch (
    bytes memory lowLevelData
) {
    if (revertOnFail) {
        revert();
    } else {
        proposal._status = ProposalStatus.Passed;
        emit FailedHandlerExecution(lowLevelData);
        return;
    }
}
```

Fig. 8. Modified (reduced) part of `executeProposal` function from Sygma contract.

Modeling and Mitigation. Fig. 7 shows a DCR model of `executeProposal`, highlighting its control-flow branches and the potential for reverts or event emissions after an external call. We re-implemented this DCR model in Fig. 8. By contrast, the revised code in Fig. 8 corresponds more directly to the DCR structure, setting `proposal._status` *only if* the call succeeds, while reverting (or emitting an event) if the call fails. This avoids a needless state write or conditional check. The resulting logic is both more intuitive and more gas-efficient, as it moves certain write operations and event emissions inside the `try` block, preventing them from being triggered unless necessary.

This is an example of intuitive action dependencies in the model leading to a version of the contract that consumes less gas. Code snippets in Fig. 6 (original version) and Fig. 8 (reduced) have the following differences:

- In the original code (Fig. 6), the `proposal._status` is always set to `ProposalStatus.Executed`, which costs a high gas fee for an `SSTORE` operation in Ethereum Virtual Machine (EVM) when changing an storage variable. Then, there is an `if` condition that checks `revertOnFail`, which also costs gas for a conditional `JUMPI`. In the reduced code (Fig. 8), `proposal._status` is only set if the `try` statement succeeds, potentially saving gas if the function call within the try statement fails.
- The `try`/`catch` blocks in the two snippets handle exceptions differently, which has an impact on gas consumption. In the original snippet, the catch block always changes the `proposal._status` and emits an event, leading to potentially higher gas usage,

```
function burnFrom(address account, uint256 amount) public virtual {
    uint256 currentAllowance = _allowances[_msgSender()][account];
    _approve(account, _msgSender(), currentAllowance - amount);
    _burn(account, amount);
}
```

Fig. 9. Buggy Deus DAO contract code (custom burn functionality).

while in the reduced snippet, the catch block conditionally reverts or emits an event based on `revertOnFail`. This means less gas consumption if the contract reverts since no event is emitted and no state change occurs.

Formally Specifiable Properties. While gas optimization is not a safety property per se, the underlying concern of whether a state write is reachable on a path that later reverts can be expressed as a reachability question: "Does there exist an execution trace in which *setStatus(Executed)* occurs and is later rolled back?" Dead-transition analysis (identifying transitions that can never fire or always lead to rollback) can flag such redundancies.

> **Insight:** Pre-development DCR modeling allows developers to visualize and streamline control-flow paths, making it easier to spot unnecessary or redundant state changes. In the Sygma executeProposal case, modeling the branching logic clarified when state writes were truly required, resulting in a more gas-efficient design.

6.3. Vulnerability mitigation through design constraints

6.3.1. Mitigating the deus DAO vulnerability

Incident. On May 6, 2023, the Deus DAO Protocol was exploited on the Arbitrum, Ethereum, and BNB chains due to a bug in its token burn function. The hack resulted in loss of 6.41 million USD accross multiple chains [49–51].

The vulnerability stems from a bug in public token `burn` function (Fig. 9). Deus DAO contract implements a custom version of ERC20 standard [52] where it defines its `burnFrom` function (Fig. 9). The bug enabling the vulnerability is caused by misplacing `account` and `_msgSender()` as the indices of `_allowances` mapping (line 2 in Fig. 9). This mistake leads to assuming the wrong current amount of allowance for the attacker enabling him to burn tokens of another owner more than he is allowed. This function does not seem to provide a clear access control as it is meant to be called by any address on the chain. If written correctly (`_msgSender()`, `account` not being misplaced), the attacker could only burn what he was allowed by the owner account. This is despite the fact that in other more carefully implemented ERC20 contracts, there often is an extra access control mechanism to revert the transaction in corner cases (not just rely on zero burning as an extreme).

Modeling and Mitigation. The ERC-20 standard [52] does not indicate any sort of access control other than the general instruction for the spender (or burner in this context) to be authorized. Even `safeTransferFrom` function in ERC-721 [53] does not imply having dedicated access control in place. These leave the developers deciding about the security violation checks based on their understanding of the contract interactions, specially when not having a full view of security properties their implementation should satisfy. An example of a more secure implementation is OpenZeppelin ERC-20 contract `_update` functionality, also used as a burn functionality (Fig. 11). Line 6 and 7 in this implementation revert the transaction in case the attacker's balance is less than the value he is asking for. With the same access control mechanism in place for DEUS DAO, the bug would not be as dangerous as the attacker's transactions would be rejected before approving the wrong action. Particular conditions of accessing `burnFrom` function such as balance or allowance checking could be modeled using condition →• or inclusion →+ / exclusion →% relation. One such model is depicted in Fig. 10. Here, the function itself comprises two significant state-changing activities: invoking `_approve` and `_burn`. This model captures the access control properties of the secure alternative implementation by: first, an unassigned role on top of each activity means it is executable by all roles in the model, hence, capturing public `burnFrom` function characteristic; second, availability of `_approve` is conditioned on other state-changing actions in the contract (that are not part of `_burnFrom` function) which is captured by the guarded inclusion relation from *previous activity* to `_approve`.

Formally Specifiable Properties. The property is a *guarded enabledness condition*. For every reachable marking M, the event *_burn(account, amount)* is enabled only if $\mathsf{Va}(allowances[msgSender][account]) \geq amount$. This is a safety property over the data-aware DCR model. Violation corresponds to a reachable marking where the burn event is enabled despite insufficient allowance, precisely the exploited scenario in the actual attack.

> **Insight:** By modeling a custom burn function (or any token operation) in DCR, developers must explicitly include the authorization checks and permissible state transitions (e.g., verifying balance or allowance). This forces clarity on who can invoke sensitive operations under which conditions, reducing the risk of the overlooked mismatch of parameters that caused the DEUS DAO exploit.

6.3.2. Mitigating the nomad bridge vulnerability

Incidnet. Bridging is the practice of connecting two blockchain networks to enable the transfer of tokens, information, and functionality between them. Nomad [11] is a bridging protocol that supports multiple chains. A successful exploit happened on its contracts

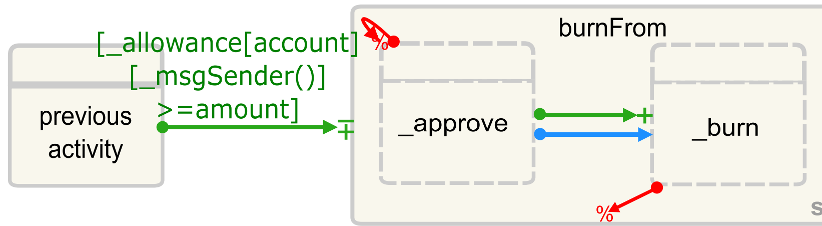


Fig. 10. Modeling access control mechanism for *burnFrom* function from Fig. 9.

```
function _update(address from, address to, uint256 value) internal virtual {
    if (from == address(0)) {
        _totalSupply += value;
    } else {
        uint256 fromBalance = _balances[from];
        if (fromBalance < value) {
          revert ERC20InsufficientBalance(from, fromBalance, value);
        }
        unchecked { _balances[from] = fromBalance - value; }
    }
}
```

Fig. 11. Access control mechanism in ERC20 standard (OpenZeppelin [54]) burn functionality.

on August 1st, 2022 that resulted in a failure of authentication in the contract. Based on the inner workings of bridge smart contract, three specific functions were involved in the flaw that resulted in their vulnerability: `initialize`, `acceptableRoot`, `process`. Without going into too much detail, `initialize` function is supposed to correctly initialize a variable (`confirmAt`) that will later be checked inside `acceptableRoot` to allow or disallow processing of a message (using the `process` function). The problem arises in the `initialize` function by not checking any condition to correctly initialize `confirmAt`, a variable that is later used to authorize the processing of messages. Therefore, the incident, in essence, is the result of missing a condition before making an event plausible. The condition is added to their implementation by their developers for the updated contract after being exploited.[1]

Modeling and Mitigation. Modeling the `process` event required specifying its enabling conditions in the DCR graph. The intended behavior is that `process` should only be enabled after `initialize` has correctly set `confirmAt`; expressing this as a condition relation (*initialize*→• *process* with an appropriate guard) made the dependency explicit. Comparing this model to the actual code revealed that the `initialize` function did not properly constrain `confirmAt`, meaning the condition was not enforced-precisely the vulnerability exploited in the incident. As it is possible to build a partial model of the contract that captures only certain high-level security properties, we present such a partial model in Fig. 12. This model uses response •→ and (guarded) inclusion →+ to show that by executing `acceptableRoot`, processing of the message is allowed under certain conditions (guarded inclusion →+).

While the presented partial model in Fig. 12 may seem very trivial, in reality implementing the condition requires functions in code to work together (`initialize`[2] and `acceptableRoot`[3]) to perform a correct authorization. The fact that the correct condition can only be checked if the setting of the values is distributed temporally in two code locations adds to the complexity of implementing a correct version.

The compelling aspect of modeling the Nomad bridge using DCR graphs is that although it captures the fact that two conditions should align (Fig. 12) to enable the `process` function, it is not concerned with the code location where this is done. This independence from implementation details not only hides the complexity and lets the modeler visualize the actual intended behavior, but also provides developers with a mind map to follow and check after implementing the contract.

Independently, RAVEN [40] identifies a business process-based defensive invariant category mined from on-chain data and demonstrates that it can serve as a fuzzing oracle to detect and prevent Nomad vulnerability. This is consistent with our DCR-based modeling and analysis.

> **Insight:** Capturing the bridging flow in a DCR model forces developers to define the precise prerequisites (e.g., correct initialization) before enabling critical functions. By rendering cross-function dependencies explicit, potential oversights are exposed. This general principle of 'model the multi-step logic before coding' can help avoid similar bridging or cross-chain permission failures.

[1] https://github.com/nomad-xyz/monorepo/commit/0e02cc1f09d16f809f5d2d8f05abbeea6d1af04e
[2] https://github.com/nomad-xyz/monorepo/blob/0e02cc1f09d16f809f5d2d8f05abbeea6d1af04e/packages/contracts-core/contracts/Replica.sol#L262C5-L274C6
[3] https://github.com/nomad-xyz/monorepo/blob/0e02cc1f09d16f809f5d2d8f05abbeea6d1af04e/packages/contracts-core/contracts/Replica.sol#L103C5-L117C6

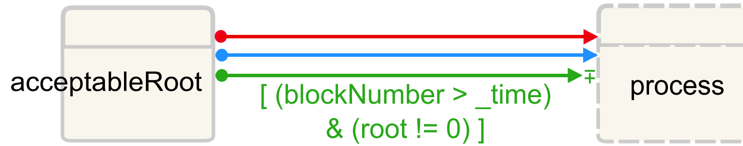


Fig. 12. Execution condition that was missing in the Nomad bridge hack.

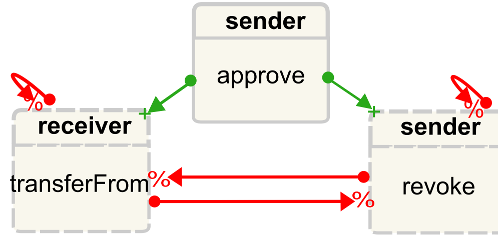


Fig. 13. IOU contract DCR model representing the vulnerability reflected in the high-level business process.

Formally Specifiable Properties. The property is a safety/reachability condition. The event *process* is never enabled in any reachable marking where *initialize* has not been executed with a valid `confirmAt` value. Formally: for all reachable markings M, if $process \in \mathrm{In}$ and *process* is enabled, then $\mathrm{Ex}(initialize)$ is defined and the guard on the condition relation from *acceptableRoot* to *process* evaluates to true. This can be verified via reachability analysis on the composed DCR model.

6.3.3. Mitigating order-of-events vulnerability

Incident. Kolluri et al. [13] analyze a critical race condition in the `IOU` contract, as illustrated in Fig. 1. The contract's intended behavior allows a token sender to approve a receiver to spend up to a specified amount of tokens on their behalf. The `approve` function in Fig. 1 also enables the sender to revoke this approval by setting the allowance to zero. However, a vulnerability arises due to the non-deterministic interleaving of transactions within the same block.

Specifically, if a sender initially approves a spender and later calls `approve(0)` to revoke this approval, there is no guarantee that the revocation transaction will execute before a concurrent `transferFrom` call by the spender. In cases where `transferFrom` is executed before the revocation, the spender can still withdraw funds, violating the sender's intent. This behavior exposes a race condition on the `allowed` state variable (Fig. 1), fundamentally breaking the expected security guarantees.

The Ethereum developer community has long acknowledged this issue [55]. The vulnerability originates from ERC20's approval mechanism [52], which lacks atomicity in modifying allowances. The community's primary recommendation has been for clients to follow a defensive pattern: first setting the allowance to zero before assigning a new value. However, this approach shifts responsibility onto users and front-end implementations, rather than enforcing safety at the contract level.

Modeling and Mitigation. We modeled the existing contract code in Fig. 13, capturing the contention between `transferFrom` and `revoke`. Each function's execution inherently excludes the other, yet both can be included within the same block due to Ethereum's transaction processing model. Crucially, constructing this model required deciding the relative ordering of `approve` and `transferFrom`. The DCR formalism does not permit "undefined" ordering: events are either independent (can execute in any order), ordered (one conditions the other), or mutually exclusive (one excludes the other). This forced decision directly confronted the race condition. If we model them as independent, the race exists; if we add a timed condition, the race is eliminated. The original code implicitly allowed independence, but the *intended* behavior-the sender can revoke before the spender acts-requires ordering, a mismatch the modeling process made explicit.

The fundamental issue visible in Fig. 13 is that it does not accurately reflect the intended business process. The expected behavior should not allow disapproving and spending to occur simultaneously, creating a race condition where execution order determines security outcomes. Instead, the correct semantics should prioritize the sender's authority in revoking permissions before they are exercised. Our proposed way to address this at the smart contract level is to enforce a strict ordering of events via a delayed condition relation ($\rightarrow\bullet$), linking `approve` to `transferFrom`. By introducing a required delay or dependency, the spender cannot execute `transferFrom` immediately after `approve` without first verifying whether the sender has revoked permission. This mechanism gives the sender a deterministic time window to ensure the revocation takes effect before any withdrawal occurs. Fig. 14 presents the model that captures the intended behavior of this class of smart contracts (ERC20) in a way that enforces design constraints ruling out this vulnerability. The guard written above the condition relation ($\rightarrow\bullet$) is the time according to ISO 8601 [56], meaning that there should be at least a delay of one day before `transferFrom` is available for execution to the receiver.

Formally Specifiable Properties. The manual modeling of the contract and inspection of it prior to development has two benefits. First, the *modeling process* itself forces the developer to decide whether `approve` and `transferFrom` should be mutually exclusive, ordered, or concurrent. This decision point directly confronts the race condition, because one cannot construct a well-formed DCR model without

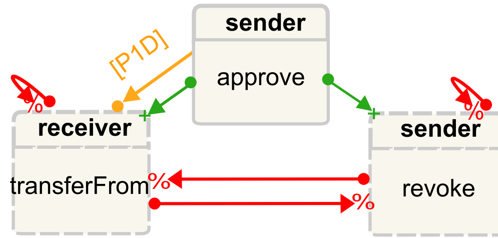


Fig. 14. Alternative model of the IOU contract that resolves the order-of-events vulnerability.

Table 1
Key insights from contract modeling.

Contract	Sec.	Insight	Design Patterns
Sygma [10]	6.1.1	Consolidate authorization into a single, direct check (as naturally represented in the DCR model relations) to make implementation clearer and more gas efficient.	Access control
	6.1.2	Explicitly initialize the contract's initial state to enhance clarity and auditability to check the contract's business logic against its DCR specifications.	—
	6.2.1	Visualize control-flow using DCR graphs to mitigate unnecessary or redundant state changes.	Action dependencies
Deus DAO [12]	6.3.1	Implement dedicated access control and guard checks for sensitive operations according to the DCR model relations even when logic embeds authorization.	Guard checks, Access control
Nomad bridge [11]	6.3.2	DCR model captures multi-step business logic distributed in multiple places in source code in a logically unified way which helps developers align conditions correctly to mitigate unwanted behavior.	Access control, Action dependencies, Guard checks
IOU [13]	6.3.3	DCR model allows specifying time constraints over sensitive actions that address race conditions, especially between users with varying permissions.	Time constraints

resolving the relative ordering of these events. Second, the formal semantics enable a precise trace property: "there exists no execution trace in which `transferFrom` executes between `approve(n)` and `approve(0)` by the same sender without respecting the timed delay k." This property is expressible in LTL and can be model-checked against the timed DCR model. In the corrected model (Fig. 14), the timed condition relation $\rightarrow\!\bullet_k$ enforces this ordering; a model checker would confirm that no violating trace exists, providing formal assurance that the race condition is eliminated. Without such a formal check, the model alone may not suffice, which underscores the importance of combining modeling with lightweight formal analysis.

> **Insight:** Race conditions often arise from ambiguous order requirements. In DCR, specifying conditional time-delayed relations between functions reveals unintended orders. Developers can infer the expected behavior from the model to prohibit the 'whoever gets in first' races common in token approvals. This approach generalizes to any scenario where partial ordering or temporal constraints on critical functions must be strictly enforced.

We present the summary of our results in Table 1. The analysis of the Sygma contract's `adminAddRelayer` and `revokeRole` functions reveals an entangled access control system where permission checks are deferred and scattered across the code, increasing complexity, potentially raising gas costs, and making maintenance more challenging. An explicit permissions check within these functions could mitigate this issue (Section 6.1.1). This study also highlights an unclear initial state (in a system that uses abstract contract states design pattern [6]) for the Sygma contract. This weakness may lead to confusion and negatively impact code understandability. Explicit initialization of abstract state `enum` could enhance clarity and intent in such instances (Section 6.1.2). On the gas consumption front, the `executeProposal` function presents opportunities for reduction. A manual implementation based on the DCR graph can improve the code's efficiency without changing its core functionality (Section 6.2.1). Reviewing the exploits in Deus DAO and Nomad Bridge reveals how meticulous modeling of access control mechanisms and guard conditions can mitigate similar occurrences, thereby emphasizing the role of DCR modeling in mitigating contract exploits (Section 6.3). The examination of the IOU contract demonstrates the importance of correctly modeling the order of events. A deviation from the intended sequence can lead to exploitations, especially in functions like `approve` (Fig. 1) where the sequence of actions directly impacts the output.

7. Discussion

In this section, we reflect on the validity of our findings, position DCR-based modeling within the spectrum of formal methods available to smart contract developers, and discuss the current tool support and practical limitations for automated analysis over DCR models.

7.1. Threats to validity

All our case studies involve contracts with known vulnerabilities or flaws. We do not claim that a developer using DCR modeling *before* knowing about the vulnerability would have caught all issues. However, we observe that the modeling process imposes a systematic discipline: constructing a DCR model requires explicitly specifying who can execute each event, what preconditions must hold, and what the initial state is. These are precisely the design decisions whose omission caused the vulnerabilities in our case studies (e.g., the missing initialization in Nomad, the entangled access control in Sygma, the unresolved event ordering in IOU). While the evidence is necessarily retrospective, we argue that a formalism which *requires* answering these questions may reduce the likelihood of silently leaving them unresolved. A prospective user study with developers where one group uses DCR modeling pre-development and a control group does not would provide stronger evidence and is an important direction for future work.

7.2. Lightweight formal modeling for smart contracts

A developer aiming to ensure the security and correctness of a smart contract can adopt either a purely formal approach, using, for example, Coq [35] or TLA$^+$ [36], or a purely informal approach, such as brainstorming on a whiteboard. However, each extreme has disadvantages: fully formal methods demand specialized expertise and significant effort, while unstructured discussions risk overlooking design flaws because they lack a mechanism to ensure completeness [57–59]. Our proposed DCR-based methodology sits between these poles. It offers enough formal rigor to specify dependencies, constraints, and roles, yet remains lightweight for rapid iteration and broader participation. Unlike a detailed formal proof, a DCR model does not require theorem-proving or exhaustive coverage of low-level code details. At the same time, it goes beyond a purely text-based whitepaper format by forcing one to articulate who can execute each event, under which conditions, and in what order. This is particularly valuable in decentralized application (dApp) development, where hidden corner cases or ambiguities in role permissions often lead to vulnerabilities [22,24]. This is reminiscent of other model-based approaches such as SOFL (Structured-Object-based-Formal Language) [60,61], which also span from informal to formal specification. However, compared to general-purpose frameworks like SOFL, our approach combines DCR graphs with domain-specific design patterns tailored to smart contracts (Table 1). In contexts where these patterns apply, that combination reduces the effort of formal modeling, since developers do not have to start from scratch but can reuse established best practices. Complementary approaches such as FLAMES [34], which synthesizes `require` statements via security-aligned LLMs, could automate the translation of DCR-level constraints into deployable Solidity guards. This further helps with concretization of the design decisions which are made explicit by our 'model to mitigate' methodology by the developers or designers.

7.3. Tool support for automated analysis

Existing DCR tools already provide partial support for the analyses mentioned in this paper. The DCR Solutions portal [18] supports interactive simulation that allows step-by-step exploration of enabled events and marking evolution, which is useful for validating initial-marking properties and access-control invariants (Sections 6.1.2 and 6.1.1). Static analysis modules in the portal can check for deadlock freedom and liveness [18], addressing whether pending obligations can always be fulfilled. Furthermore, the formal semantics of DCR graphs (including timed and data-aware extensions) have been shown to be amenable to encoding in existing model-checking frameworks. Prior work has established decidability results for safety and liveness properties of DCR graphs [4,5], and translations to Büchi automata or Petri-net-based representations enable the use of standard temporal-logic model checkers.

For the data-aware properties (Sections 6.3.1 and 6.3.3), verification is more challenging because guard expressions introduce data dependencies that can make the state space infinite or very large. Abstraction techniques such as predicate abstraction over the value domain or bounding the number of participants is needed to make model checking tractable in these cases. Lightweight approaches, such as bounded model checking (exploring traces up to a fixed depth), offer a pragmatic middle ground that can detect violations without full state-space exploration.

7.4. Practical limitations for formal analysis

We identify two practical limitations of formal analysis over DCR models of smart contracts: state explosion with data and modeling fidelity.

State explosion with data. Smart contracts manipulate rich data types (balances, mappings, timestamps), and the data-aware DCR semantics may reflect this. However, full model checking over unbounded data domains is undecidable in general [5]. Practical approaches require either abstraction (e.g., treating balances as "sufficient" vs. "insufficient") or bounded analysis.

Modeling fidelity. DCR models abstract away low-level implementation details (Section 3). A property verified on the DCR model holds for the *design* but does not automatically transfer to the Solidity implementation unless the implementation faithfully refines the model. Bridging this gap requires either certified code generation from DCR models or runtime monitoring tools such as HIGHGUARD [7] that check implementation behavior against the DCR specification.

Despite these limitations, even lightweight formal checks such as verifying enabledness invariants and performing bounded trace analysis add significant rigor beyond purely manual inspection. We view the integration of such checks into the DCR modeling workflow as an important direction for future work that bridges process modeling and formal correctness guarantees.

Our pre-development modeling ('model to mitigate') raises essential questions, such as 'Who can call this function?' and 'What must happen before or after it?' that might otherwise surface too late in the smart contract's life cycle, leading to vulnerabilities. Although we do not present a head-to-head comparison of DCR against other formalisms, our experience (Section 6.3) shows that even a short, high-level DCR model can detect design-level vulnerabilities (e.g., Nomad exploit) and efficiency improvements (e.g., Sygma). This aligns with existing work in model-based systems engineering: graphical, iterative modeling can reduce error rates and clarify complex interactions early on [58].

8. Conclusion and future work

We have presented the 'Model to Mitigate' methodology, demonstrating how Dynamic Condition Response (DCR) graphs can serve as an effective tool for analyzing and mitigating vulnerabilities in smart contract design. The core insight is that the discipline of constructing a DCR model forces developers to make explicit decisions about access control, event ordering, preconditions, and initial states-precisely the design decisions whose omission is a documented root cause of smart contract vulnerabilities.

Our analysis of four real-world contracts-Sygma, Deus DAO, Nomad bridge, and IOU-yielded six key insights spanning entangled access control, unclear initial states, reducible gas consumption, and mitigatable vulnerability patterns. In each case, we identified how the modeling process itself surfaces the flaw by requiring explicit answers to questions that the original implementations left implicit or ambiguous. To substantiate these findings formally, we introduced a trace conformance notion linking DCR-level guarantees to Solidity execution traces, providing a conditional guarantee: properties verified on the DCR model transfer to the implementation to the extent that the implementation faithfully conforms to the model. For each case study, we identified the formally specifiable property at stake and discussed the practical scope of automated verification.

Several avenues for future work emerge from this study. First, automating the extraction of DCR models from Solidity code would reduce the manual effort required and address concerns about inter-modeler variability. Second, integrating lightweight model checking-such as bounded trace analysis and enabledness invariant verification-directly into the DCR modeling workflow would strengthen the formal guarantees. Third, leveraging runtime monitoring tools such as HighGuard [7] and exploit-generation tools such as XploGen [8], in conjunction with DCR specifications, would enable real-time detection of specification violations. These oracles can further be used with fuzzing techniques [40,62,63] to detect undiscovered vulnerability patterns. Finally, certified code generation from DCR models would bridge the modeling fidelity gap, ensuring that the implementation provably conforms to its specification. Data-driven invariant catalogs such as those produced by Raven [40] could guide which properties to prioritize when analyzing DCR models.

Despite the challenges that persist in smart contract security due to the complexity and interrelationships of decentralized applications [64–67], our results suggest that even lightweight, high-level DCR modeling positioned between unstructured design discussions and heavyweight formal verification can meaningfully reduce the risk of design-level vulnerabilities by imposing a systematic discipline.

CRediT authorship contribution statement

Mojtaba Eshghie: Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Formal analysis, Conceptualization; **Wolfgang Ahrendt:** Writing - review & editing, Writing - original draft, Formal analysis; **Cyrille Artho:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis; **Thomas Troels Hildebrandt:** Writing – review & editing, Writing – original draft, Formal analysis; **Gerardo Schneider:** Writing – review & editing, Writing – original draft, Formal analysis.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests. Mojtaba Eshghie reports financial support was provided by KTH Royal Institute of Technology. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

Part of this work has been funded by grant 2019-0458 ("TEMOS: Temporal monitoring of smart contracts") by the Swedish Science Council (Vetenskapsrådet).

References

[1] N. Szabo, Smart contracts: building blocks for digital markets, EXTROPY J. Transhumanist Thought 18 (2) (1996) 28.
[2] G. Wood, Ethereum yellow paper, 2022. https://github.com/ethereum/yellowpaper.
[3] H. Rezaei, M. Eshghie, K. Anderesson, F. Palmieri, SoK: root cause of $1 billion loss in smart contract real-world attacks via a systematic literature review of vulnerabilities, 2025. arXiv:2507.20175
[4] H. Normann, S. Debois, T. Slaats, T.T. Hildebrandt, Zoom and enhance: action refinement via subprocesses in timed declarative processes, in: BPM 2021, Springer, Cham, 2021, pp. 161–178.
[5] T.T. Hildebrandt, H. Normann, M. Marquard, S. Debois, T. Slaats, Decision modelling in timed dynamic condition response graphs with data, in: Business Process Management Workshops, Springer, Cham, 2022, pp. 362–374.

[6] M. Eshghie, W. Ahrendt, C. Artho, T.T. Hildebrandt, G. Schneider, Capturing smart contract design with DCR graphs, in: C. Ferreira, T.A.C. Willemse (Eds.), Software Engineering and Formal Methods, Springer Nature Switzerland, Cham, 2023, pp. 106–125. https://doi.org/10.1007/978-3-031-47115-5_7
[7] M. Eshghie, C. Artho, H. Stammler, W. Ahrendt, T.T. Hildebrandt, G. Schneider, HighGuard: cross-chain business logic monitoring of smart contracts, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24, ACM, New York, NY, USA, 2024, pp. 1–12. https://doi.org/10.1145/3691620.3695356
[8] M. Eshghie, C. Artho, Oracle-guided vulnerability diversity and exploit synthesis of smart contracts using LLMs, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24, ACM, New York, NY, USA, 2024, pp. 1–10. https://doi.org/10.1145/3691620.3695292
[9] A. Augusto, R. Belchior, M. Correia, A. Vasconcelos, L. Zhang, T. Hardjono, SoK: security and privacy of blockchain interoperability [extended version], 2024, https://doi.org/10.36227/techrxiv.24595764.v2
[10] ChainSafe, Chainbridge-solidity, 2023. https://github.com/ChainSafe/chainbridge-solidity.
[11] N. Bridge, Introduction, 2023. https://docs.nomad.xyz/nomad-101/introduction.
[12] DEUS, DEUS finance, 2024, https://deus.finance/.
[13] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, P. Saxena, et al., Exploiting the laws of order in smart contracts, in: SIGSOFT ISSTA 2019, ACM, New York, NY, USA, 2019, pp. 363–373.
[14] SunWeb3Sec, DeFi hacks reproduce - foundry, 2023. https://github.com/SunWeb3Sec/DeFiHackLabs.
[15] S. Team, Solidity - solidity 0.8.20 documentation, 2023. https://docs.soliditylang.org/en/v0.8.20/.
[16] T.T. Hildebrandt, R.R. Mukkamala, Declarative event-based workflow as distributed dynamic condition response graphs, in: K. Honda, A. Mycroft (Eds.), Proceedings Third Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010, vol. 69 of *EPTCS*, arxiv, 2010, pp. 59–73. https://doi.org/10.4204/EPTCS.69.5
[17] M. Eshghie, W. Ahrendt, C. Artho, T.T. Hildebrandt, G. Schneider, Formalizing smart contract design patterns with DCR graphs, Softw. Syst. Model. (2026) 1–39. https://doi.org/10.1007/s10270-026-01366-w
[18] DCRGraphs, DCR portal dashboard, 2024. https://dcrgraphs.net/.
[19] J. Feist, G. Grieco, A. Groce, Slither: a static analysis framework for smart contracts, in: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), IEEE, 2019, pp. 8–15.
[20] C. Diligence, Mythril: security analysis tool for ethereum smart contracts, 2024, (https://github.com/ConsenSys/mythril). Accessed: 2024-10-10.
[21] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, Z. Li, et al., A survey of smart contract formal specification and verification, ACM Comput. Surv. 54 (7) (2021) 148:1–148:38. https://doi.org/10.1145/3464421
[22] Obsidian, The obsidian smart contract language - obsidian 0.1 documentation, 2023. https://obsidian.readthedocs.io/en/latest/.
[23] Y. Hirai, Bamboo: a language for morphing smart contracts, 2023. https://github.com/pirapira/bamboo.
[24] A. Mavridou, A. Laszka, E. Stachtiari, A. Dubey, VeriSolid: correct-by-design smart contracts for ethereum, 2019. arXiv:1901.01292
[25] W. Duo, H. Xin, M. Xiaofeng, Formal analysis of smart contract based on colored petri nets, IEEE Intell. Syst. 35 (3) (2020) 19–30. https://doi.org/10.1109/MIS.2020.2977594
[26] Z. Liu, J. Liu, Formal verification of blockchain smart contract based on colored petri net models, in: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), vol. 2, IEEE, New York, NY, USA, 2019, pp. 555–560. https://doi.org/10.1109/COMPSAC.2019.10265
[27] S. Alqahtani, X. He, R. Gamble, P. Mauricio, et al., Formal Verification of Functional Requirements for Smart Contract Compositions in Supply Chain Management Systems, ScholarSpace, Blockchain Cases and Innovations, 2020. http://hdl.handle.net/10125/64392.
[28] T. Abdellatif, K.-L. Brousmiche, Formal verification of smart contracts based on users and blockchain behaviors models, in: 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), IEEE, New York, NY, USA, 2018, pp. 1–5. https://doi.org/10.1109/NTMS.2018.8328737
[29] C.D. Clack, G. Vanca, Temporal aspects of smart contracts for financial derivatives, 2018. https://doi.org/10.48550/arXiv.1805.11677
[30] M. Ron, On the specification and verification of atomic swap smart contracts, 2018. https://doi.org/10.48550/arXiv.1811.06099
[31] C. Laneve, C. Coen, A. Veschetti, On the prediction of smart contracts' behaviours, in: From Software Engineering to Formal Methods and Tools, and Back: Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday, Lecture Notes in Computer Science, Springer International Publishing, New York, NY, USA, 2019, pp. 397–415. https://doi.org/10.1007/978-3-030-30985-5_23
[32] G. Bigi, A. Bracciali, G. Meacci, E. Tuosto, et al., Validation of decentralised smart contracts through game theory and formal methods, in: C. Bodei, G. Ferrari, C. Priami (Eds.), Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday, Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 142–161. https://doi.org/10.1007/978-3-319-25527-9_11
[33] D. Suvorov, V. Ulyantsev, Smart contract design meets state machine synthesis: case studies, 2019. https://doi.org/10.48550/arXiv.1906.02906
[34] M. Eshghie, G. Morello, M. Lauretano, A. Bartel, M. Monperrus, FLAMES: fine-tuning LLMs to synthesize invariants for smart contract security, 2025. arXiv:2510.21401
[35] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development: Coq'Art: the Calculus of Inductive Constructions, Springer Science & Business Media, 2013.
[36] Y. Yu, P. Manolios, L. Lamport, Model checking TLA+ specifications, in: Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Springer, 1999, pp. 54–66.
[37] J.E. Zhong, K. Cheang, S. Qadeer, W. Grieskamp, S. Blackshear, J. Park, Y. Zohar, C. Barrett, D.L. Dill, The move prover, in: S.K. Lahiri, C. Wang (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2020, pp. 137–150. https://doi.org/10.1007/978-3-030-53288-8_7
[38] L. Breidenbach, P. Daian, F. Tramèr, A. Juels, Enter the hydra: towards principled bug bounties and exploit-resistant smart contracts, in: Proceedings of the 27th USENIX Conference on Security Symposium, SEC'18, USENIX Association, 2018, pp. 1335–1352.
[39] S. Zhou, Z. Yang, J. Xiang, Y. Cao, M. Yang, Y. Zhang, et al., An ever-evolving game: evaluation of real-world attacks and defenses in ethereum ecosystem, in: Proceedings of the 29th USENIX Conference on Security Symposium, SEC'20, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA, 94710, 2020, pp. 2793–2809.
[40] M. Eshghie, M. Mazura, A. Bartel, Raven: mining defensive patterns in ethereum via semantic transaction revert invariants categories, in: Proceedings of the 19th IEEE International Conference on Software Testing, Verification and Validation (ICST 2026), IEEE, Daejeon, Republic of Korea, 2026. https://arxiv.org/abs/2512.22616.
[41] J. Roethke, Vulnerability update: security improvements to ChainBridge ERC-721 handler, 2022. https://blog.chainsafe.io/vulnerability-update-security-improvements-to-chainbridge-erc-721-handler-c3d1425e71c.
[42] S. Kravchenko, D. Luca, Consensys diligence, chainbridge, 2020, https://consensys.net/diligence/audits/private/adash47d-chainbridge/chainbridge-audit-2020-04.pdf.
[43] S.C. A. S. .S. Finance, Smart contract audit services | solidity finance, 2024, https://solidity.finance/audits/DEUS/.
[44] Compound, Compound governance examples, 2018, Accessed: 2023-10-16, https://github.com/compound-developers/compound-governance-examples.
[45] P. De Filippi, A. Wright, Blockchain and the Law: the Rule of Code, Harvard University Press, 2018. https://www.hup.harvard.edu/books/9780674241596.
[46] Parity Technologies, A postmortem on the parity multi-sig library self-destruct, 2017, Accessed: 2025, https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.
[47] INCIBE-CERT, Parity wallet with a value of 150 million dollars in ethereum blocked, 2017, Accessed: 2025, https://www.incibe.es/en/incibe-cert/publications/cybersecurity-highlights/parity-wallet-value-150-million-dollars-ethereum-blocked.
[48] R. Behnke, Explained: the ronin network hack (August 2024), 2024, Accessed: 2025, https://www.halborn.com/blog/post/explained-the-ronin-network-hack-august-2024.
[49] etherscan.io, Ethereum transaction hash (txhash) details | etherscan, 2024, http://etherscan.io/tx/0x6129dd42778345bc278822a7feadeacb933f5e56ce511 14e686832ad239307a8.
[50] Q.-W. Security, Decoding deus DAO $6.5 million exploit | quillaudits, 2023,

[51] PeckShield Inc., @peckshield public burn vulnerability on DeusDao, 2023, https://x.com/peckshield/status/1654626667787321344.
[52] E. Foundation, ERC-20 token standard, 2024, https://ethereum.org/en/developers/docs/standards/tokens/erc-20/.
[53] W. Entriken, D. Shirley, J. Evans, N. Sachs, ERC-721: non-fungible token standard, 2024, https://eips.ethereum.org/EIPS/eip-721.
[54] OpenZeppelin, Openzeppelin-contracts/contracts/token/ERC20/ERC20. sol at master OpenZeppelin/Openzeppelin-contracts, 2023, https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol.
[55] Ethereum Community, EIP-738: ethereum improvement proposal discussion, 2025, Accessed: 2025-02-05, https://github.com/ethereum/EIPs/issues/738.
[56] International Organization for Standardization, ISO 8601: date and time format, 2025, Accessed: 2025-02-05, https://www.iso.org/iso-8601-date-and-time-format.{html}.
[57] N. Shevchenko, An introduction to model-based systems engineering (MBSE), 2024, (Carnegie Mellon University, Software Engineering Institute's Insights (blog)). Accessed: 2025-Jan-30, https://insights.sei.cmu.edu/blog/an-introduction-to-model-based-systems-engineering-mbse/.
[58] J.A. Estefan, et al., Survey of model-based systems engineering (MBSE) methodologies, Incose MBSE Focus Group 25 (8) (2007) 1–12.
[59] S. Friedenthal, A. Moore, R. Steiner, A Practical Guide to SysML: the Systems Modeling Language, Morgan Kaufmann, 2014.
[60] S. Liu, A.J. Offutt, C. Ho-Stuart, Y. Sun, M. Ohba, SOFL: a formal engineering methodology for industrial applications, IEEE Trans. Softw. Eng. 24 (1) (1998) 24–45. https://doi.org/10.1109/32.663996
[61] S. Liu, Formal Engineering for Industrial Software Development: Using the SOFL Method, Springer Science & Business Media, 2004.
[62] H. Wang, Y. Liu, Y. Li, S. Lin, C. Artho, L. Ma, Y. Liu, Oracle-supported dynamic exploit generation for smart contracts, IEEE Trans. Dependable Secure Comput. 19 (03) (2022) 1795–1809.
[63] M. Eshghie, C. Artho, D. Gurov, Dynamic vulnerability detection on smart contracts using machine learning, in: EASE 2021, ACM, 2021, pp. 305–312.
[64] A. Groce, J. Feist, G. Grieco, M. Colburn, What are the actual flaws in important smart contracts (and how can we find them)?, in: J. Bonneau, N. Heninger (Eds.), Financial Cryptography and Data Security, Lecture Notes in Computer Science, Springer International Publishing, 2020, pp. 634–653. https://doi.org/10.1007/978-3-030-51280-4_34
[65] coderenaAudits, Security audit reports | Code4rena, 2024, https://code4rena.com/reports.
[66] Krum, Pashov's security audits, reviews, contributions, 2023, https://github.com/pashov/audits.
[67] C. Diligence, Public smart contract audits and security reviews, 2024, https://consensys.net/diligence/audits/.