Formalizing smart contract design patterns with DCR graphs

Software and Systems Modeling (2026) 25:1083–1121
https://doi.org/10.1007/s10270-026-01366-w

SPECIAL SECTION PAPER

Formalizing smart contract design patterns with DCR graphs

Mojtaba Eshghie[1] · **Wolfgang Ahrendt**[2] · **Cyrille Artho**[1] · **Thomas Troels Hildebrandt**[3] · **Gerardo Schneider**[4]

Received: 26 April 2024 / Revised: 23 February 2026 / Accepted: 24 February 2026 / Published online: 21 May 2026

Abstract

Smart contracts manage blockchain assets and embody business processes. Yet, mainstream languages lack explicit support for process concepts such as roles, action dependencies, and time constraints, leading to increased implementation complexity and analysis challenges. To address this, we use Dynamic Condition Response (DCR) graphs, a formal business process modeling language, to formalize the semantics of smart contract business logic. Modeling smart contracts in DCR graphs involves translating their underlying behavioral logic into a declarative visual model using DCR's explicit constructs for events, roles, data, time, and inter-event relationships. Furthermore, we systematically model 15 common high-level smart contract *design patterns*, representing recurring solutions to business logic-level problems. These formalizations reduce ambiguity compared to informal descriptions and serve as language-independent specifications. We demonstrate the modeling process through three complete smart contract case studies that combine six design patterns. Our modeling methodology, formalizations, and correspondence between smart contract semantics and DCR graphs enable future automated analysis and verification.

Keywords Smart contract modeling · DCR graphs · Design patterns formalization

1 Introduction

A *smart contract* is a reactive program that is deployed as the immutable code section of an account on a distributed ledger (blockchain). Smart contracts are typically used to implement a contractual agreement between multiple parties. Therefore, they are akin to special business processes specifying contractual agreements on actions to be carried out by different roles [1–3]. These contracts offer advantages such as uncompromised (automated) execution even without a trusted party.

However, they can also be complex and difficult to design and understand, a problem exacerbated by the fact that the code section of smart contracts cannot be easily updated or modified once deployed. This immutability demands rigorous design and upfront analysis, since any errors or security flaws discovered post-deployment cannot *simply* be corrected via patching, potentially leading to significant financial loss [4].

Like regular software, smart contracts embody common best practices as *design patterns* [5]. However, the focus shifts: unlike many traditional object-oriented structural patterns concerned with class/object interactions and composition, smart contract patterns primarily govern the dynamic interactions between functions executed by different roles. Consequently, the patterns presented here are predominantly interaction-focused, though some define higher-level system architectures (cf. Sect. 3).

The need for formalized design patterns for smart contracts is evident, as current research on smart contract behavior patterns is mostly informal [6–9]. Traditional software design patterns, which focus on interactions between object instances, contrast with smart contract patterns where the interactions between functions in different roles are crucial due to often adversarial interests [5].

In a normal business process environment, different roles collaborate to achieve a common business goal. In contrast, different roles in a smart contract typically have adversarial interests. Therefore, smart contracts introduce new types of

Communicated by Carla Ferreira and Tim Willemse.

✉ Mojtaba Eshghie
mojtabaeshghie@gmail.com

[1] KTH Royal Institute of Technology, Stockholm, Sweden

[2] Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden

[3] University of Copenhagen, Copenhagen, Denmark

[4] Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden

patterns of behavior, which have so far only been informally described [6–9]. To provide an unambiguous understanding of the patterns that can also provide the basis for formal specifications, we set out to extend the study and formalization of process patterns to include these smart contract patterns.

Solutions to adversarial-interest problems often use time constraints between actions cutting across the process and the more standard use of roles and sequential action dependencies. We find that a declarative notation involving data and time is appropriate for formalizing the new smart contract process patterns. Moreover, smart contract languages exhibit a transactional behavior of actions, where an action may be attempted but aborted if the required constraints for executing it are not fulfilled. This highlights that the executability of actions is governed by potentially complex *pre-conditions* involving roles, state, time, and data. This implies that modeling smart contract behavior requires a formalism adept at explicitly and declaratively specifying this rich set of constraints that determine a valid transaction execution.

For these reasons, we use DCR graphs [10], which are by now a well-established declarative business process notation that has been extended with data [11, 12], time [13, 14], and sub-processes [15, 16].

DCR graphs visually capture properties such as the partial ordering of events, roles of contract users, and temporal function attributes. By integrating these aspects directly into the formalism alongside explicit data handling and declarative rules (cf. Sect. 2.2), DCR graphs provide a means to represent smart contract logic that can be more expressive compared to formalisms lacking native support for these combined features, such as basic Finite State Machines (FSM) or traditional flow-chart notations. Using DCR graphs, we can represent a smart contract with a clear and concise model.

As the design patterns we model concern the high-level behavior of a smart contract under analysis, we elide technical details of the patterns' implementation and execution. Therefore, we use the term *high-level* design pattern for the patterns that DCR graphs capture well, as they represent the underlying business process of the contract.

Our high-level design patterns demonstrate how a system should *behave* rather than be *implemented.* The behavior is described as *interface-level* restrictions on smart contracts. It is therefore possible to use our designs as a platform- and language-independent modeling before the contract development, applicable beyond the Ethereum/Solidity examples used for illustration herein. Our models tell what the users interacting with the smart contract can/cannot do. Activities in our models match the transactional semantics of blockchain, as an event execution in the DCR model can be mapped to a successful/mined transaction in blockchain.

DCR models of smart contracts serve as business logic specifications that guide the implementation process. The constraints defined in the model regarding allowed actions, roles, timing, and data can be translated into enforceable checks in the smart contract code (e. g. `require` statements in Solidity). This translation can be automated or manually done according to our proposed methodology in Sect. 4, ensuring the implementation adheres to the formalized behavioral specification, with runtime enforcement provided by the blockchain's transaction semantics (i.e., reverts upon failed checks). These models can also be used for other types of analysis such as runtime monitoring tools [17] (cf. Sect. 6).

More concretely, our contributions are:

1. We systematically identify and distinguish high-level design patterns from low-level (implementation-specific) patterns in smart contracts. Table 1 contains 15 high-level design patterns we formalized. We demonstrate how these are modeled using DCR graphs. Section 3 contains the pattern motivations, DCR models with description, and examples for each of the patterns.
2. We demonstrate how one can capture the design of three complete contracts, as a combination and extension of design patterns, with the help of DCR graphs (Sects. 5.1, 5.2, and 5.3). We use the mapping between the smart contracts and DCR presented in Sect. 4 to model these contracts. The modeled contracts use in total six of the design pattern models from this paper, which helps to demonstrate the combinability of pattern models to shape the final design of the contract.
3. As a result of a thorough analysis of real-world contracts, including popular contract libraries, we identify and formalize *time incentivization* (Sect. 3.1) as a new design pattern. This pattern is extensively used by the Solidity developer community but is not yet introduced as a design pattern in research literature [7–9, 18].

Our application of these formalized design patterns in Sects. 5.1, 5.2, and 5.3 shows that using DCR graphs can facilitate the development of correct and reliable smart contracts by providing a clear and easy-to-understand specification. Moreover, DCR specifications can provide a basis for automated (dynamic or static) analysis of smart contracts, which we exemplified by preliminary runtime verification infrastructure and experiments in our tool paper [17].

Our usage of DCR graphs to model smart contracts and our focus on high-level (business logic-level) rather than low-level (platform-specific) properties allows us to capture the key semantics of the contract succinctly. We can verify properties (and likewise lack of vulnerabilities pertaining to these properties) related to roles and access control [19–21], partial ordering of actions (function calls and transaction execution) [22], as well as time-based properties [23, 24]. Furthermore, not being concerned with low-level patterns and properties lets our approach remain cross-platform and

not tied to the features and limitations of a certain smart contract execution environment. We believe that these patterns provide a systematic classification of best practices for smart contracts in a similar way that software design patterns shaped the design of traditional software and established a nomenclature for it, while capturing aspects that are unique to smart contracts [5].

This paper builds on our previous work [1], where we modeled 19 design patterns using DCR graphs. Due to space constraints, we could only introduce the patterns briefly there. In this work, we give a systematic and comprehensive description of all patterns, which are now also presented in a general form that can be applied to specific cases. As a consequence of this formalization and generalization, we have also merged four patterns pairwise into two patterns because they can fit the same overall design. Moreover, we have chosen to elide two patterns presented previously [1], as they apply at a relatively low level in the design. Section 3 explains these changes in detail.

This paper is organized as follows: Section 2 introduces smart contracts and DCR graphs. Section 3 gives an overview of 15 high-level smart contract design patterns, which we formalize using DCR graphs. Section 4 provides the correspondence between Solidity constructs and DCR graphs. Section 5 shows three case studies on a casino, an escrow, and a digital marketplace smart contract. Section 6 presents a comprehensive review of related literature, followed by a discussion in Sect. 7. The paper concludes in Sect. 8.

2 Background

2.1 Smart contracts: ethereum and solidity

This work uses Ethereum and its primary language, Solidity, for illustrating examples, although the DCR modeling approach we present is generally applicable and platform-independent. Ethereum [3], with its built-in cryptocurrency Ether, remains a leading blockchain platform supporting decentralized applications executed via *smart contracts* [25]. These contracts are essentially programs deployed to unique addresses on the blockchain, capable of managing assets (like Ether held in the `TimeLockWallet`) and enforcing predefined rules.

The most popular programming language for writing Ethereum smart contracts is Solidity [26]. It is a high-level, statically typed language specifically designed for implementing contracts. Solidity code compiles down to bytecode executable by the Ethereum Virtual Machine (EVM). Solidity supports inheritance, and libraries where "contract" definitions encapsulate persistent state and the functions that operate on that state.

In Solidity, smart contracts serve as the fundamental building blocks for decentralized applications. Each deployed contract resides at a unique blockchain **`address`** and can hold, receive, and send the platform's native cryptocurrency (like Ether). Contracts manage persistent data stored directly on the blockchain in *state variables* (fields).

To illustrate these concepts, we use the `TimeLockWallet` contract (Fig. 1) as an example. This contract implements a simple time-locked vault functionality. Its state is defined by two key variables: **`address public`** `owner` and **`uint256 public`** `unlockTime`. These variables store the designated owner's address and the timestamp until which funds are locked, respectively. Because they are marked **`public`**, Solidity automatically generates getter functions allowing external users to read their values. These state variables collectively represent the contract's current state and correspond directly to the data component (Va) of a DCR graph's marking (see Sect. 2.2).

The contract's logic and state transitions are implemented in *functions*. Interactions occur when users (or other contracts) send transactions targeting these functions. Within a function's execution context, special variables like **`msg.sender`** (the address initiating the call) and **`msg.value`** (the amount of Ether sent with the call) are available. The `TimeLockWallet`'s **`constructor`** is a special function executed only once upon deployment, initializing the state by setting the `owner` to the contract creator (**`msg.sender`**, see constructor logic around Sect. 2.1).

The mentioned contract has two main functions, `lock` and `withdraw`, both restricted to be only callable by the `owner`. The `lock(`**`uint256`** `_unlockTime)` function allows the owner to set a future `unlockTime`. It is marked **`payable`**, enabling the contract to receive and hold Ether sent along with the function call (though the amount sent does not affect the lock duration). Before setting the time, it uses **`require`** statements to validate pre-conditions: the provided `_unlockTime` must be in the future (**`require`**`(_unlockTime > block.timestamp)`), and the wallet must not already be locked (**`require`**`(unlockTime == 0)`).

The `withdraw()` function permits the owner to retrieve the funds, but only after the lock duration expires. It enforces this with **`require`**`(block.timestamp >= unlockTime, "Too early")` (Sect. 2.1), also checking that the wallet is indeed locked (**`require`**`(unlockTime != 0)`). If the conditions hold, it transfers the contract's entire Ether balance to the owner and resets the lock by setting `unlockTime = 0` (Sect. 2.1), making the wallet available for a new lock cycle.

Access control is implemented using the `onlyOwner` *modifier*, attached to both `lock` and `withdraw`. Modifiers encapsulate reusable checks; here, `onlyOwner` contains the crucial **`require`**`(msg.sender == owner)` statement (Sect. 2.1) to ensure only the designated owner can execute these functions. The **`require`** statement is thus the primary mechanism

Fig. 1 Solidity code for the example `TimeLockWallet` contract

```
contract TimeLockWallet {
  address public owner;
  uint256 public unlockTime;    // Va(Lock)
  modifier onlyOwner() {
    require(msg.sender == owner, "Not owner"); // role enforcement
    _;
  }
  constructor() {
    owner = msg.sender;
  }
  function lock(uint256 _unlockTime) external payable onlyOwner {
    require(_unlockTime > block.timestamp, "Unlock in past");
    require(unlockTime == 0, "Already locked");
    unlockTime = _unlockTime;  //  Va(Lock)
  }
  function withdraw() external onlyOwner {
    require(unlockTime != 0, "Not locked");
    require(block.timestamp >= unlockTime, "Too early");
    uint256 amt = address(this).balance;
    unlockTime = 0;   // Prevent rewithdrawl unless "Lock" event is
        executed again.
    payable(owner).transfer(amt);
  }
}
```

for enforcing various pre-conditions—validating inputs, checking permissions (via modifiers), ensuring correct state, and implementing time-based logic—halting execution and reverting the transaction if any check fails. These **`require`** checks directly implement the conditions that determine whether an action (translated into a DCR event) is *enabled* in our DCR models, as detailed in the mapping (Sect. 4).

In Ethereum, interactions with smart contracts occur by calling their **`public`**/**`external`** functions through *transactions*. Users send transactions to a contract's address, specifying the function to call (e.g., `lock` or `withdraw`), any necessary data (arguments like `_unlockTime`), and potentially an amount of Ether (e.g., when calling the **`payable`** `lock` function). Network participants called *miners* (or validators) execute these transactions and include them in new blocks added to the immutable ledger. This execution consumes computational resources, measured in *gas*, for which the transaction initiator pays in Ether, ensuring network security and incentivizing efficient code.

A critical feature of Ethereum is transaction atomicity: a transaction either completes successfully, updating the contract's state, or it fails entirely. Failure can occur due to insufficient gas, invalid operations, or unmet conditions checked within the code (e.g., attempting to call `withdraw` before `unlockTime` has passed in the `TimeLockWallet` example). If a transaction fails, any state changes it attempted are *reverted*, restoring the state as if the transaction never happened (though the gas fee is still consumed). This revert mechanism is fundamental for enforcing contract rules and relates to how non-enabled events are handled in process models like DCR graphs (see *enabledness* in Sect. 2.2).

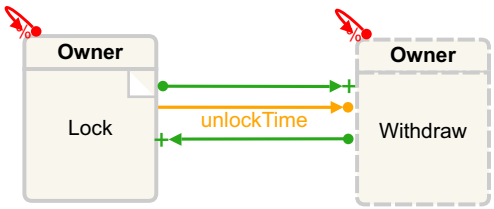


Fig. 2 DCR model for the `TimeLockWallet` contract (Fig. 1)

2.2 Dynamic condition response graphs

A Dynamic Condition Response (DCR) graph defines a process using a declarative, constraint-based approach represented visually as a graph (formal definition in Definition 1, example in Fig. 2). The term "declarative" signifies that the process logic is specified by a set of constraints—such as conditions that must be met for an event to be available for execution (→•), or responses that become required after an event (•→)—rather than by explicitly defining all possible execution sequences as in traditional imperative flowcharts. The process is "dynamic" because its state (marking) evolves over time as events are executed. The set of currently possible events/actions changes dynamically based on the history of occurred events and how the current state satisfies the constraints. DCR graphs offer an alternative to state machines; instead of using transitions to represent events, DCR graphs represent events as nodes (boxes in Fig. 2) and the relation between events through edges in the model (arrows in Fig. 2) [10].

For the example in Fig. 2 (model for the contract in Fig. 1), the nodes of the graph constitute a set $E = \{Lock,\ Withdraw\}$

of events, each labeled with its role (both *Owner* here) and action name. In this example, *Lock* is an input action (flipped corner) that takes the unlock-time as input, and *Withdraw* is a simple action. Input actions receive a value from the environment when the action is executed, which is associated with the event. Computation actions execute a computation expression (that may refer to the current value assigned to itself or other events) when the action is executed, which is then associated with the event.

The directed edges between nodes define the DCR relations that enforce the timelock behavior (Fig. 2). A guarded condition relation $\rightarrow\bullet$ from *Lock* to *Withdraw* with delay `unlockTime` (Fig. 1) ensures that *Withdraw* does not become enabled until the specified timestamp has passed. An include relation $\rightarrow+$ from *Lock* to *Withdraw* makes the withdrawal action available as soon as the lock is set. Finally, reflexive exclude relations $\rightarrow\%$ on each event prevent immediate re-execution of the same event until the other has occurred (so you cannot call *Lock* twice in a row nor *Withdraw* twice in a row).

Excluded events (using the exclude $\rightarrow\%$ relation) cannot be executed and their effect on other events that they have relations with, is *ignored*. The possibility for an event to be excluded makes it easy to express *defeasible rules* [27]. For instance, in Fig. 5, the *Obligor* role can execute *Enforce Penalty* event after the execution of *Initiate* event, except if the *Obligee*, in the meantime, executes *Fulfill Obligation*, in which case the penalty event is excluded.

In our `TimeLockWallet` model (Fig. 2), the condition relation $\rightarrow\bullet$ from *Lock* to *Withdraw* event, ensures that the *Withdraw* event only becomes enabled once the stored `unlockTime` (assigned by *Lock*) has passed.

The execution state of a DCR graph is given by a marking, which assigns state information to each event. In the original version of DCR graphs [10], the marking of the graph assigned three Booleans to each event, denoting respectively if the event had been executed, if it is required to be executed (or excluded) in the future and if it is currently excluded.

In our example, after the *Lock* event executes, $\mathsf{Va}(Lock)$ (i.e., value of the *Lock* event, cf. Definition 1) holds the unlock-time and In (i.e., set of all of events in the model that are currently *included*) contains *Withdraw*. At this point, because of the delayed condition $\rightarrow\bullet$ relation in Fig. 2 (enforced by Sect. 2.1 for `TimeLockWallet` contract in Fig. 1), *Withdraw* is only executable after at least *unlockTime* is passed from the time *Lock* is executed. After the *Owner* role executes *Withdraw*, this event remains disabled until included by another event again by executing *Lock* (enforced by 20 in Fig. 1).

In this paper, we use an extended version of DCR graphs, allowing data [12], time [13], and sub-processes [15], which is supported by the online design tool.[1] The version of DCR graphs used in the online design tool also adds two new effect relations: A *value relation* $\rightarrow\!\!=$, denoted by a gray directed edge with an $=$ symbol at the target, with the effect of updating the value of the target event when the source event is executed, and a *cancel relation* $\bullet\!\!\rightarrow\!\!\times$, denoted by a brown directed edge with a $\times$ symbol at the target, with the effect of removing a possible pending execution requirement (e. g., due to a previous activation of a response rule) of the target event when the source event is executed. For a DCR graph with data, the marking assigns the current data value (if any) associated with each event. For a DCR graph with time, the marking additionally assigns time information to events, concretely, *how long ago an event was executed* (if it has been executed) and a deadline for *when it is required to be executed* (if it is required to be executed in the future).

In Definition 1, we give the formal definition of timed DCR graphs [13, 14] with sub-processes [15] and data [11, 12]. We use a new edge denoting a value effect, making it possible for one event to update the data of another event.

We also assume a discrete-time model where time proceeds in steps represented by natural numbers. Let $\omega = \{0, 1, 2, ...\}$ denote the set of natural numbers including zero. Time values representing minimum delays belong to ω. Maximum deadlines, which can be finite or unbounded, belong to the extended set $\omega \cup \{\infty\}$, where ∞ is a distinct symbol representing infinity.[2] An infinite deadline (∞) is used to represent that an event must eventually be executed (as known from classical liveness properties), and serves as the default deadline for response relations if not otherwise specified.

Definition 1 A *timed DCR graph with sub-processes, data, and roles* G is given by a tuple $(E, E_G, nest, D, M, \rightarrow\bullet, \bullet\rightarrow, \bullet\rightarrow\times, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \rightarrow\!\!=, L, l)$ where

1. E is a finite set of *events*,
2. E_G is a finite set of *event groups*,
3. $nest \in E \cup E_G \rightharpoonup E \cup E_G$ is an acyclic *nesting* function, i. e., for all $k > 1$ $nest^k(e) \neq nest(e)$, if $nest(e)$ is defined.
4. $D : E \rightarrow \mathsf{Exp_E} \uplus \{?\}$ defines an event as either a *computation event* with expression $d \in \mathsf{Exp_E}$ or an *input event* $?$,
5. $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va}) \in \big((E \rightharpoonup \omega) \times (E \rightharpoonup (\omega \cup \{\infty\})) \times \mathcal{P}(E) \times (E \rightharpoonup V)\big)$ is the *timed marking with data*,
6. $\rightarrow\bullet \subseteq E \times \omega \times \mathsf{BExp_E} \times E$, is the *guarded timed condition relation* (where ω specifies minimum delay),
7. $\bullet\rightarrow \subseteq E \times (\omega \cup \{\infty\}) \times \mathsf{BExp_E} \times E$, is the *guarded timed response relation* (where the time component from $\omega \cup \{\infty\}$ specifies the maximum deadline),

[1] Available for free for academic use at http://dcrsolutions.net

[2] The ISO 8601 standard (www.iso.org/iso-8601-date-and-time-format.html) is used in the design tool, allowing the use of years, months, days, and seconds.

8. $\bullet\!\!\rightarrow\!\!\times, \rightarrow\!\!\diamond, \rightarrow\!\!+, \rightarrow\!\!\%, \not\rightarrow \subseteq E \times \mathsf{BExp_E} \times E$ are the *guarded cancel, milestone, include, exclude, and value relations,* respectively,
9. $L = \mathcal{P}(R) \times A$ is the *set of labels*, with R and A sets of roles and actions,
10. $l : E \rightarrow L$ is a *labeling function* between events and labels.

The nesting function $nest(e)$ defines the immediate parent of an event e or group e within the DCR graph's hierarchy. The nature of this parent determines the semantic interpretation:

1. Nesting Using Sub-Processes: If an event e_{child} is nested under another event e_{parent} (where $e_{parent} \in E$, thus $nest(e_{child}) = e_{parent} \in E$), then e_{parent} is defined as a *sub-process event*. This structure represents hierarchical decomposition, where e_{parent} encapsulates a logical phase or composite action, and e_{child} is one of its internal steps. The execution and completion semantics, as detailed below, involve specific interactions between e_{parent} and its nested contents.
2. Nesting Within a Group (Syntactic Shorthand): If an event e is nested under an event group g (where $g \in E_G$, thus $nest(e) = g \in E_G$), this primarily serves as a *syntactic shorthand* for applying relations collectively. We use the notation $e < g$ for this direct containment within a group. This allows DCR relations (like $\rightarrow\!\!\bullet$, $\bullet\!\!\rightarrow$, etc.) defined between groups (or between a group and an event) to implicitly apply to all events contained within those groups. Let $<^*$ denote the reflexive, transitive closure of $<$. Then, writing $(x, k, d, y) \in \rightarrow\!\!\bullet$ (where x, y can be events or groups, and where, following Definition 1, $k \in \omega$ is the time delay and $d \in \mathsf{BExp_E}$ is the guard condition) implies that for any event e_x with $e_x <^* x$ and any event e_y with $e_y <^* y$, the relation $(e_x, k, d, e_y) \in \rightarrow\!\!\bullet$ holds.

This distinction is crucial: nesting under an event ($nest(e) \in E$) creates a semantically significant sub-process structure, while nesting under a group ($nest(e) \in E_G$) provides a notational convenience for managing relations applied to sets of events.

In the case where there exists $e' \in E$ such that $e' <^* e''$ and $nest(e'') = e \in E$, we call the event e a *sub-process* event. Sub-processes are useful for hierarchical modeling in DCR graphs [15], allowing complex workflows, common in smart contracts, to be broken down into manageable modules for distinct phases or composite actions.

Intuitively, a sub-process acts as a container for a related set of activities. Execution effectively "enters" a sub-process when one of its events becomes enabled (which requires the sub-process event itself to be included and enabled) and is executed. Executing an event e_1 nested inside a sub-process

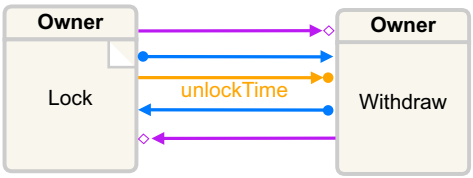


Fig. 3 Alternative DCR model for the `TimeLockWallet` contract (Fig. 1) using milestone $\rightarrow\!\!\diamond$ and response relations $\bullet\!\!\rightarrow$

e can trigger the immediate, atomic completion, and execution marking of e itself (updating $\mathsf{Ex}(e')$). This occurs if, in the state M' resulting directly from e_1's execution, two conditions simultaneously hold: (1) the sub-process event e is enabled, and (2) the sub-process e is "accepting." A sub-process is considered accepting when none of the events directly nested within it are both included (In) and pending (Re).

Like regular events, sub-process events can be targeted by relations; including or excluding a sub-process event controls the overall availability of all activities nested within it.

Whether an event is pending and included is defined by the marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$, which defines the state of the process. Formally, the marking consists of three partial functions, Ex, Re, and Va, and a set In of events. $\mathsf{Ex}(e)$, if defined, yields the time since event e was last **Ex***ecuted*. $\mathsf{Re}(e)$, if defined, yields the deadline for when the event is **Re***quired* to happen (if it is included) and in this case we say the event is pending. The set In is the currently **In**cluded events. Finally, $\mathsf{Va}(e)$, if defined, is the current **Va***lue* of an event.

2.2.1 Enabledness.

The condition $\rightarrow\!\!\bullet$ and milestone $\rightarrow\!\!\diamond$ relations constrain the enabling of events and determine when events can be executed. As exemplified above, a condition $e' \rightarrow\!\!\bullet e$ means that e' must have been executed at least once or currently be excluded for e to be enabled. A milestone $e' \rightarrow\!\!\diamond e$ means that e' must either be currently excluded or not be pending for e to be enabled.

Figure 3 shows an alternative model for `TimeLockWallet` contract (Fig. 1). Instead of inclusion $\rightarrow\!\!+$ and exclusion $\rightarrow\!\!\%$ relations, which are used in the first model of this contract (Fig. 2), the alternative model uses milestone $\rightarrow\!\!\diamond$ and response $\bullet\!\!\rightarrow$ relations to enforce the order *Lock* and *Withdraw* may happen. When executed, initially, the only *enabled* event in this model (Fig. 2) is *Lock*, as there is a $\rightarrow\!\!\bullet$ from *Lock* to *Withdraw* disabling it unless *Lock* is executed. When *Lock* executes, it enables *Withdraw* and at the same time makes *Withdraw* pending (because of the response $\bullet\!\!\rightarrow$ from *Lock* to *Withdraw*). Note that the delay on the condition relation $\rightarrow\!\!\bullet$ stops *Withdraw* being immediately executable. Further-

more, the milestone $\rightarrow\diamond$ relation from *Withdraw* to *Lock* also takes effect, and since the source of this relation *(Withdraw)* is pending, *Lock* cannot execute unless *Withdraw* is executed. After a round of executing *Lock* and then *Withdraw*, the response $\bullet\rightarrow$ relation from *Withdraw* to *Lock* takes effect, making *Lock* pending. This also disables *Withdraw*, as there is another milestone from *Withdraw* to the pending *Lock* event as well. This model shows how we can enforce specific event orderings only using semantics of *enabledness* instead of inclusion $\rightarrow+$ and exclusion $\rightarrow\%$ relations.

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 $l(e) = (R', a)$ for $r \in R'$ and (1) e is included: $e \in \mathsf{In}$, (2) all conditions for the event are met: $\forall e' \in E.(e', k, d, e) \in \rightarrow\bullet.(e' \in \mathsf{In} \wedge [[d]]_M) \implies \mathsf{Ex}(e') \geq k$ and (3) all milestones for the event are met: $\forall e' \in E.(e', k, d, e) \in \rightarrow\diamond.(e' \in \mathsf{In} \wedge [[d]]_M) \implies \mathsf{Re}(e')$ is undefined and (4) if $e <^* e'$ such that $nest(e') = e'' \in E$, then e'' is enabled and can be executed by role r.

In the DCR graph in Fig. 2, the only event initially enabled is the event *Lock*. It is enabled because it is included and it is not under the influence of any other relation from any other included event. The *Withdraw* event is disabled because it is initially excluded (marked by a dashed border). The lack of a role written on the event box means the event can be executed by all roles in the model (in Figs. 2 and 3, only the role *Owner* can execute the events, enforced by Sect. 2.1 in Fig. 1 in the contract implementation).

2.2.2 Brief DCR execution semantics

The execution of an enabled event $e \in E$ (triggered by role r where $l(e) = (R', a)$ and $r \in R'$, and potentially receiving an input value v if e is an input event, $D(e) =?$) transitions the graph from its current marking $M = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$ to a subsequent marking $M' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}', \mathsf{Va}')$. This state transition, denoted $M \xrightarrow{e,[v]} M'$, is defined by updating the components of the marking based on the effects of executing e. We assume a discrete-time model where executing an event advances time by $\Delta t = 1$:

1. Update Executed Time (Ex'): The time since the last execution is updated for all relevant events.
   - The executed event e is marked as just executed: $\mathsf{Ex}'(e) = 0$.
   - For all other events $e' \in E \setminus \{e\}$, if e' had been executed previously (i.e., $\mathsf{Ex}(e')$ was defined), time progresses: $\mathsf{Ex}'(e') = \mathsf{Ex}(e') + 1$. If $\mathsf{Ex}(e')$ was undefined, $\mathsf{Ex}'(e')$ remains undefined.
2. Update Data Values (Va'): The value associated with the executed event and potentially others are updated. Let $\mathsf{Va}_{new}(e)$ be the new value determined for event e.
   - Determine the value of the executed event e:
     - If $D(e) =?$ (input event), $\mathsf{Va}_{new}(e) = v$.
     - If $D(e) = d_e \in \mathsf{Exp_E}$ (computation event), $\mathsf{Va}_{new}(e) = [[d_e]]_M$ (the expression d_e is evaluated using the values from the marking M *before* the execution).
     - If e is a simple event, $\mathsf{Va}_{new}(e)$ is typically undefined.
   - Initialize $\mathsf{Va}' = \mathsf{Va}$. Set $\mathsf{Va}'(e) = \mathsf{Va}_{new}(e)$ if $\mathsf{Va}_{new}(e)$ is defined.
   - Apply value relations $\rightarrow\!\!=$: For every event e'' such that $(e, d, e'') \in \rightarrow\!\!=$ and the guard condition $[[d]]_M$ evaluates to true (using marking M), update the target's value using the new value of e: $\mathsf{Va}'(e'') = \mathsf{Va}_{new}(e)$.
3. Update Included Set (In'): The set of included events is modified based on include $\rightarrow+$ and exclude $\rightarrow\%$ relations originating from e.
   - Initialize a temporary set $\mathsf{In}_{temp} = \mathsf{In}$.
   - Apply includes: For all e'' such that $(e, d, e'') \in \rightarrow+$ and $[[d]]_M$ is true, add e'' to the set: $\mathsf{In}_{temp} = \mathsf{In}_{temp} \cup \{e''\}$.
   - Apply excludes: For all e'' such that $(e, d, e'') \in \rightarrow\%$ and $[[d]]_M$ is true, remove e'' from the set: $\mathsf{In}_{temp} = \mathsf{In}_{temp} \setminus \{e''\}$. (Excludes typically override includes if both apply from the same source event e).
   - The final set of included events is $\mathsf{In}' = \mathsf{In}_{temp}$.
4. Update Required/Pending Deadlines (Re'): The set of pending events and their deadlines are updated based on response $\bullet\rightarrow$ and cancel $\bullet\rightarrow\times$ relations originating from e.
   - Initialize $\mathsf{Re}' = \mathsf{Re}$.
   - Apply responses: For all e'' such that $(e, t_{dl}, d, e'') \in \bullet\rightarrow$ and $[[d]]_M$ is true, a pending requirement for e'' is created or updated. The new deadline is t_{dl} relative to the current time. If e'' was already pending with deadline t'_{dl} (i.e., $\mathsf{Re}(e'')$ was defined), the new deadline becomes $\min(t'_{dl}, t_{dl})$. Formally, $\mathsf{Re}'(e'') = \min(\mathsf{Re}(e''), t_{dl})$, where $\mathsf{Re}(e'')$ is treated as ∞ if it was previously undefined.
   - Apply cancels: For all e'' such that $(e, d, e'') \in \bullet\rightarrow\times$ and $[[d]]_M$ is true, any pending requirement for e'' is removed. Formally, $\mathsf{Re}'(e'')$ becomes undefined. This takes precedence over any response relation from the same event e that might target e''.

All guard expressions d on the relations originating from e are evaluated based on the marking M *before* the execution step. The resulting updates to Ex, Re, In, Va are then applied

conceptually simultaneously to determine the new marking M'. For a more comprehensive treatment of the execution semantics, including interactions with time progression and sub-process handling, we refer the reader to [12, 15].

3 Smart contract design patterns as DCR graphs

Due to the high stakes involved in applications, ensuring the safety and security of smart contracts is crucial. To address this, both the Solidity documentation and the developer and smart contract security community have put forth a range of recommendations. A considerable number of these recommendations are now known as *design patterns* [5], because they are widely adopted as a solution to recurring design problems. These patterns promote the creation of contracts that are designed with safety and security in mind, mitigating potential risks and safeguarding users' assets in the design phase.

We collected these design patterns from academic design pattern surveys [6–9, 28–31], smart contract programming language documentations [25, 32, 33], Blockchain platform specifications [3, 34], and smart contract security best practices [4, 35, 36]. The prevalence of these design patterns is also confirmed by their occurrence in popular libraries and contracts such as OpenZeppelin, SolidState Solidity, and Aragon OSx [37–41].

We identify the design patterns representing high-level behavior (Table 1) rather than implementation- and platform-specific patterns. By "low-level," we refer to patterns that primarily dictate "how" a single function should be implemented internally, often focusing on its computation, gas optimization techniques, or language/platform-specific operations within the function's body, rather than the overall interaction flow between functions. The latter (platform-specific patterns) concern features inside a function (the execution of which we model as an event in DCR graphs). It is important to emphasize that this focus means our DCR-based approach is inherently geared toward modeling the workflow and interaction logic between contract functions, rather than the detailed intra-function implementation steps dictated by some lower-level patterns. For example, while our high-level models capture that a function requires a specific role (access control pattern, Sect. 3.6), they abstract away "how" that check is implemented (e.g., using a Solidity `modifier` versus an inline `require` statement). Similarly, patterns concerning specific gas-saving coding tricks or ensuring specific security invariants "during" a single function's execution (like the precise ordering within the checks-effects-interactions pattern to prevent reentrancy [49–51]) fall into this low-level category that we do not directly model with DCR graphs.

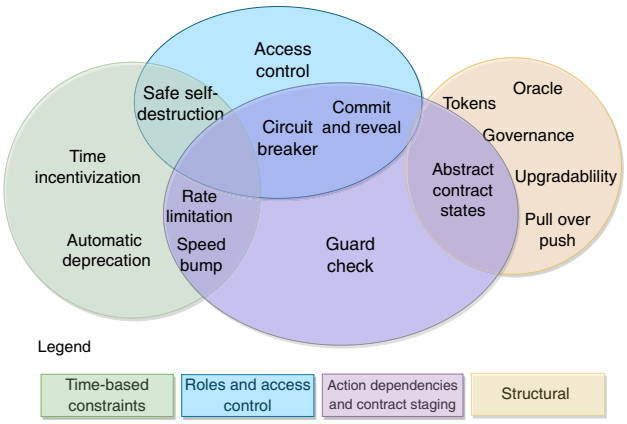


Fig. 4 Classification of smart contract high-level design patterns

The analysis of low-level patterns is orthogonal to our work and can be handled, e. g., by runtime analysis of code [52].

In our current work, two of the 19 design patterns reviewed in our previous work [1] are determined not suitable as high-level patterns, as they pertained to low-level Ethereum-specific operations (the checks, effect, interactions [8] and secure ether transfer [9] patterns). Therefore, they are not included in our current formalized design patterns (Table 1 and Sect. 3). Furthermore, the previously captured *Escapability* in our prior work [1] is now incorporated as a use-case of the formalized circuit breaker pattern (the example in Section 3.8). Likewise, the previously captured *Time Constraint* in our prior work is now incorporated as a specialized version of speed bump design pattern (Sect. 3.4).

We classify the high-level patterns into the following four categories (see Fig. 4):

1. **Time-based constraints**: Time-based patterns impose constraints on when activities can be performed, which typically include deadlines and delays.
2. **Roles and access control**: Role-based access control [53] restricts access to given functions to predefined roles.
3. **Action dependencies and contract staging**: High-level design of a smart contract may impose an ordering on any pair of activities.
4. **Structural patterns**: These patterns impose a certain structure on the contract business process (and the implementation as a result) and are created by combining other design patterns.

Many patterns combine aspects of several categories; Fig. 4 depicts a classification of 15 design patterns we have identified. We elucidate these patterns further below. Table 1 gives an overview of references of the design patterns, libraries that implement them, their respective DCR models (Sect. 3), and their use-case in our full contract DCR models

Table 1 Smart contract business logic design patterns and their respective DCR graph models. These patterns are further categorized in Fig. 4

Design Pattern	Libraries	Model	Use-case	Brief Purpose
Time incentivization[1]	[40, 42–45]	§3.1	§5.1	Motivates timely action via deadlines & penalties/rewards
Automatic deprecation[7]	—	§3.2	—	Disables functions after a set time/block
Rate limitation[8]	[35]	§3.3	—	Limits action frequency/volume per time period
Speed bump[8]	[35]	§3.4	—	Delays critical actions conditionally for review/reaction
Safe self-destruction[6, 7]	[37]	§3.5	—	Provides secure, controlled contract termination/fund recovery
Access Control/Authorization[6, 7, 9]	[37–39]	§3.6	§5.1, 5.3	Restricts execution based on caller roles/permissions
Commit and reveal[7]	—	§3.7	§5.1	Hides inputs until later reveal to prevent front-running
Circuit breaker [8]	—	§3.8	—	Allows pausing critical functions in emergencies
Guard check[9]	[46]	§3.9	§5.3	Validates function inputs/state preconditions
Abstract contract states[7]	[37]	§3.10	§5.1, 5.2, 5.3	Manages allowed actions/order via explicit state tracking
Oracle [6, 7]	—	§3.11	—	Enables contract interaction with off-chain data sources
Token [6]	[37]	§3.12	—	Standardizes on-chain digital asset (FT/NFT) management
Pull over push[7]	[37]	§3.13	§5.3	Favors recipient withdrawal (*pull*) over contract send (*push*)
Upgradability[7, 18]	[37]	§3.14	—	Allows logic changes post-deployment preserving state/address
Governance [6, 28]	[37, 47, 48]	§3.15	—	Enables decentralized control changes via proposals/voting

[1] We classify this recurring real-world solution as a design patterns, despite its lack of recognition in prior literature

(Sect. 5). The correspondence between Solidity constructs and DCR graphs is presented in Sect. 4.

In the following subsections, we delve into each design pattern, motivate it through describing the challenge it offers a solution to, offer the visual representation of each pattern model and a description of the associated DCR graph models.

3.1 Time incentivization

3.1.1 Motivation

In blockchain ecosystems, smart contracts operate as a reactive system, executing transactions in response to the specific function calls. However, certain scenarios require actions to be performed at specific times even when the actors are adversaries. This can lead to potential issues like the stalling of a protocol progress due to a user's inaction, which could result in the eternal locking of assets or services the contract provides. Traditional solutions involve external oracles for scheduling transactions at certain times (Sect. 3.11), increasing complexity and risks [54]. The purpose of the *time incentivization* pattern is to motivate parties to cooperate even in the presence of conflicting interests. The parties are incentivized by stipulating a deadline before which an actor shall make a transaction. The actor that misses the deadline can afterward be penalized by other actors.

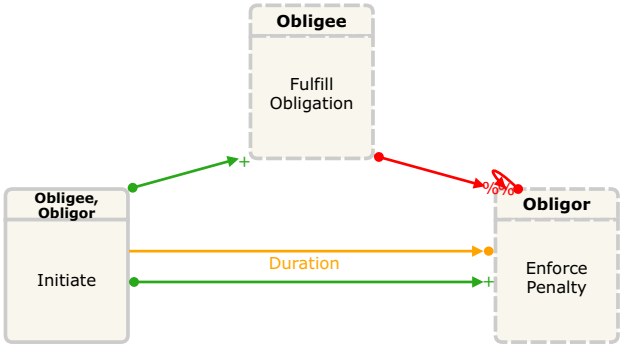


Fig. 5 DCR model of time incentivization design pattern

3.1.2 Model

The time incentivization design pattern, as illustrated in Fig. 5, employs a DCR model to formalize the interaction between Obligee and Obligor in a contract. This model introduces activities such as *Initiate*, *Fulfill Obligation*, and *Enforce Penalty*, guided by condition relation →• with a deadline to ensure obligations are met in a timely manner. The inclusion and exclusion relations (→+ and →%) exist to ensure the correct order of activities which is first either *Obligee* or *Obligor* roles initiating the agreement. Next, *Obligee* has to fulfill the obligation. The model emphasizes the importance of structured incentives to mitigate non-cooperative behavior. The current work is the first to categorize this as a design pattern and formalize it.

3.1.3 Example

The implementation of time incentivization is prevalent in several high-profile smart contracts, including Augur, MakerDAO, Compound, Aragon Court, and Synthetix [38, 40, 42, 45]. We demonstrate usage of this pattern in a game smart contract in Sect. 5.1 to incentivize the game owner to continue the game by forfeiting the bets.

3.2 Automatic deprecation

3.2.1 Motivation

There are scenarios where the contract should not allow a subset of its functionalities after a certain point in time or block number. This is where we use automatic deprecation, which is the opposite design pattern to a time constraint (Sect. 3.4). Automatic deprecation stipulates a deprecation time (block number) after which a function is not executable anymore [7].

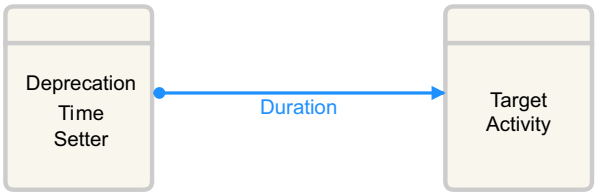


Fig. 6 Automatic deprecation DCR model

3.2.2 Model

To automatically deprecate part of the contract functionality, typically a *require* statement (likewise *assert* in other smart contract languages) checks at the function entry point against the expiration. This means that a smart contract function can be called and reverted, which is different from DCR model semantics, where an event is enabled only if it successfully executes.

Figure 6 demonstrates the DCR model of this design pattern. Benefiting from the response relation •→ from event *Deprecation Time Setter* to *Target Event*, we set a deadline for *Target Event* with the *Duration*. *Target Event* is not available for execution after this deadline.

3.2.3 Example

This pattern is used both standalone and as part of other design patterns (Sect. 3.15). For instance, on-chain voting protocols such as OpenZeppelin's governance library typically use these patterns to deprecate the *vote* function available after a specific deadline [55].

3.3 Rate limitation

3.3.1 Motivation

There are scenarios where the intensity of sensitive operations should be limited within a certain time period. Rate limitation imposes this limit on the number of successful function calls by a participating user during a specific time period [35].

3.3.2 Model

To model this pattern, we assume the sensitive event is the *Target Event* operation. We present the model in Fig. 7. When the model is simulated, the only included available event to execute is *Set Limit*. The activity *New Period* is initially *executed* (green tick on event box). The value relation ⇒ from *New Period* to *Rate Limiter* copies value 0 to *Rate Limiter* every time *New Period* is executed to reset the *Rate Limiter* for the new period.

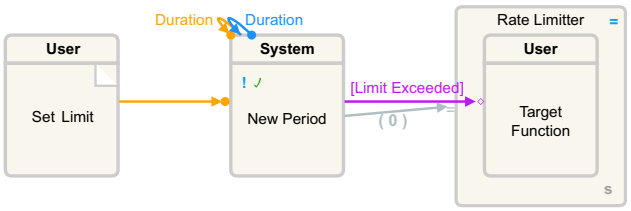


Fig. 7 Rate limitation pattern modeled in DCR graphs

So far, the model keeps the system from exceeding the limits. Without adding other relations to the model, *New Period* would not execute periodically. Therefore, we use both the semantics of delay (on condition →•) and deadline (on response •→) at the same time on *New Period*, to force it to be executed exactly after *Duration* is passed. Each execution of *New Period* sets a deadline and delay of *Duration* for the given event. Having mentioned relations on *New Period* and assigning an automatic agent to the *System* (when simulating the model) ensures that *New Period* is indeed executed at exactly *Duration* intervals. Each execution of *Target Function* by the role *User* updates the value of the *Rate Limiter* (which is a sub-process) via a computation acting event as a counter. The = symbol in the top-right corner of this sub-process indicates that it performs a computation.

Based on the purple milestone relation →◇, if the current period amount does not exceed the limit, role *User* can execute *Target Event*. Furthermore, having the milestone relation from *New Period* to *Rate Limiter* occur periodically ensures that if the current execution of *Target Event* of the period exceeds the limit, *Target Event* is not executable until the next execution of *New Period*.

3.3.3 Example

An example of this pattern is when the contract explicitly limits the total amount of transfers allowed during the defined period. At each request, it is checked whether the request is within the limits of the total amount of assets that can be withdrawn (or the number of any sensitive action). Figure 8 shows this usage of the pattern. For the target-sensitive operation *(Withdraw)* to be available for execution, the current amount that is withdrawn should be less than the *limit* set by event *Set Withdraw Limit.*

3.4 Speed bump

3.4.1 Motivation

In situations involving a sensitive operation, such as the withdrawal of assets or the authorization of significant actions, the smart contract can impose a delay, specifically if the operation satisfies predefined conditions. This delay gives the smart contract *owner* or the monitoring system time to react to the event. This conditional delay mechanism is known as speed bump design pattern. Speed bump imposes a temporal barrier that gives enough time to detect a problematic event and mitigate it.

3.4.2 Model

The DCR model for the Speed Bump pattern is illustrated in Fig. 9. The pattern contains two activities: *Critical Action Request*, initiated by a user, and the subsequent *Critical Action*. The core mechanism of the speed bump, the conditional delay, is modeled using a timed and guarded condition relation (→•) originating from *Critical Action Request* and targeting *Critical Action*. This relation incorporates two things:

- A time delay, labeled *Duration*, specifying the minimum time that must elapse after *Critical Action Request* executes before *Critical Action* can become enabled.
- A boolean guard expression, labeled *Delay Triggering Condition*. This condition is evaluated based on the state at the time *Critical Action Request* is executed.

Consequently, *Critical Action* is only enabled if: (1) the specified *Duration* has passed since the request, and (2) the *Delay Triggering Condition* was true when the request was made. If the condition was false, the delay is bypassed, and the action might become available immediately (subject to other constraints).

In addition to the core conditional delay, other relations manage the process flow and state, as shown in Fig. 9:

- Order of activities: An include relation (→+) from *Critical Action Request* to *Critical Action* makes the action potentially executable after the request. An exclude relation (→%) often excludes *Critical Action Request* itself upon its execution to prevent immediate re-requests.
- Obligation and reusability of the actions: The model in the figure also contains a response relation (•→) from *Critical Action Request* to *Critical Action*. This makes executing *Critical Action* mandatory (pending) once the request is made and the conditions (including time delay, if applicable) are met. Concurrently, a milestone relation (→◇) from the (pending) *Critical Action* to *Critical Action Request* prevents a new request from being initiated while the current action is still pending. This ensures the action completes before a new cycle can begin. Once *Critical Action* executes, it typically includes (→+) *Critical Action Request* again, allowing the pattern to be reused.

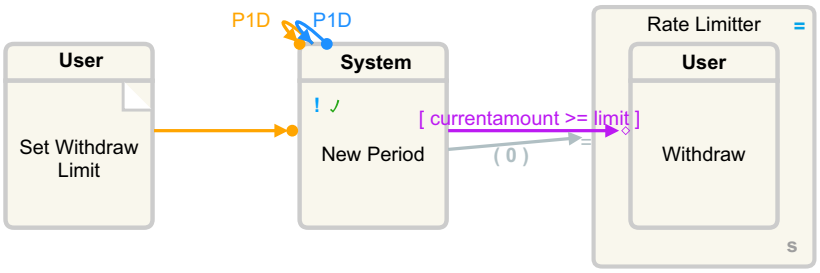


Fig. 8 Rate limitation pattern usage for an example financial application

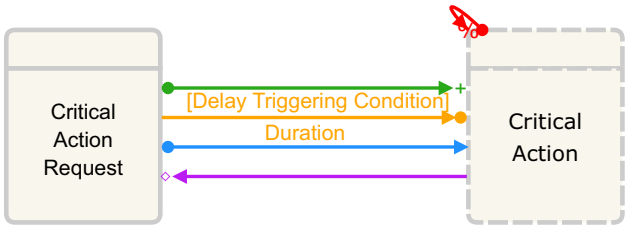


Fig. 9 Speed bump design pattern DCR model

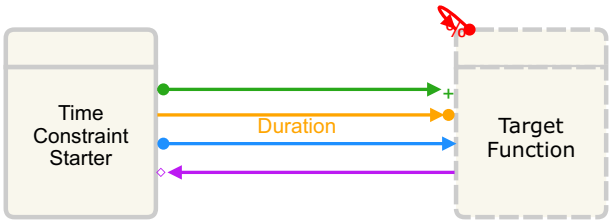


Fig. 10 Time constraint (specialized version of speed bump) DCR model

These supporting relations ensure the proper ordering and state management around the central conditional delay mechanism.

3.4.3 Example

In Decentralized Finance (DeFi) platforms, where users engage in lending, borrowing, and trading of digital assets, the risk of rapid, large-scale withdrawals (also known as "bank runs") can destabilize platforms and markets. By implementing a speed bump on withdrawals, DeFi platforms can introduce a cooling-off period. This not only provides the platform with the necessary time to ensure liquidity but also mitigates the impact of panic selling, preserving stability in volatile market conditions [56].

In multi-stage business processes, ensuring that code execution follows predefined stages is a challenge. We mandate this through time-based dependencies. To model time-based dependencies, we use a specialized version of speed bump pattern called "time constraint" where the condition on →• is always true, and the availability of *Target Function* is only constrained by time elapsed after execution of *Time Constraint Starter* (Fig. 10).

In some cases, *Target Function* is constrained from the commencement of the contract, i. e., *Time Constraint Starter* should be interpreted as execution of *constructor* function of the contract.

Modeling more complex time constraints where only part of a function should be executed or blocked based on time may be less straightforward. One approach is to interpret Solidity's **`require`** statements (**`assert`** in other smart contract languages) as conditions in DCR graphs, connecting multiple activities to shape the business logic [32, 33]. In such cases, a single **`public`**/**`external`** function in the contract interface, especially one with complex internal logic enforcing different execution stages (e.g., via multiple **`require`** statements), might not map directly to a single DCR event. Instead, the behavioral dependencies inherent in such stage-based function execution can be modeled more effectively in DCR. This is achieved by defining multiple distinct activities (representing the key stages or outcomes of the function's logic) and connecting them with the appropriate condition relations (→•) to enforce the required sequence and timing constraints that mirror the function's internal stage-dependent behavior.

The lottery contract source code uses a simple form of time constraint, where the *payoutReady* function reverts the transaction if it is not yet ready [57].

ChainSafe bridge Solidity contract [47] represents a complex usage of time constraint pattern. It enables or disables parts of a function based on a certain (relative) time or stage of the execution, i. e., the success of a function call depends on the time (block number) or contract state. Fig. 11 presents the function `voteProposal` from the mentioned contract. This function has multiple **`require`** and if-else statements that control action dependencies and cause the same function to have different results. In the mentioned contract, the function `voteProposal` based on the time of calling and what actions were carried out so far in the contract's lifetime.

3.5 Safe self-destruction

3.5.1 Motivation

There are certain scenarios where the code of the contract is no longer needed or the contract functions should no longer

Fig. 11 Solidity code of the `voteProposal` function in ChainSafe bridge contract

```
function voteProposal(uint8 domainID, uint64 depositNonce, bytes32
    resourceID, bytes calldata data) external onlyRelayers
    whenNotPaused {
  address handler = _resourceIDToHandlerAddress[resourceID];
  uint72 nonceAndID = (uint72(depositNonce)<<8) | uint72(domainID);
  bytes32 dataHash = keccak256(abi.encodePacked(handler, data));
  Proposal memory proposal = _proposals[nonceAndID][dataHash];
  require(_resourceIDToHandlerAddress[resourceID] != address(0), "no
      handler");
  if (proposal._status == ProposalStatus.Passed) {
    executeProposal(domainID, depositNonce, data, resourceID, true);
    return;
  }
  address sender = _msgSender();
  require(uint(proposal._status) <= 1, "Executed/cancelled");
  require(!_hasVoted(proposal, sender), "Relayer already voted");
  if (proposal._status == ProposalStatus.Inactive) {
    proposal = Proposal({
      _status : ProposalStatus.Active,
      _yesVotes : 0,
      _yesVotesTotal : 0,
      _proposedBlock : uint40(block.number)
    });
  } else if (uint40(sub(block.number, proposal._proposedBlock)) >
      _expiry) {
    proposal._status = ProposalStatus.Cancelled;
  }
  if (proposal._status != ProposalStatus.Cancelled) {
    proposal._yesVotes = (proposal._yesVotes | _relayerBit(sender)).
        toUint200();        proposal._yesVotesTotal++;
    if (proposal._yesVotesTotal >= _relayerThreshold) {
      proposal._status = ProposalStatus.Passed;
    }
  }
  _proposals[nonceAndID][dataHash] = proposal;
  if (proposal._status == ProposalStatus.Passed) {
    executeProposal(domainID, depositNonce, data, resourceID,false);
  }
}
```

be available for execution. This typically happens when the contract is migrated to another address (through upgradability pattern Sect. 3.14) or in the event of a critical bug or vulnerability, contract functionality should be completely removed. Self-destruction operation is provided for the contract *Owner* to delete the contract from blockchain and send the cryptocurrency in the contract to another address at the same transaction. This operation is directly available in Solidity and Vyper languages using *selfdestruct* [25, 33, 58]. In other smart contract ecosystems a similar effect is viable through sending all the cryptocurrency of the contract to another address and putting a permanent lock on all contract functions. Given the irreversible and significant impact of this operation, it should be encapsulated within a distinct function, meticulously protected against unauthorized access and inadvertent executions.

The function that encapsulates self-destruction capability should be executable only by the most privileged role in the contract. Although the unintended trigger of this function by the most privileged role is unlikely, precautions should still be taken to prevent it.

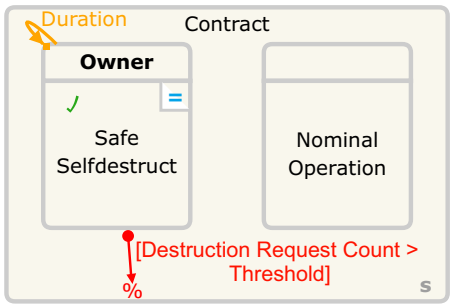


Fig. 12 Safe selfdestruct pattern DCR model

3.5.2 Model

Figure 12 presents safe self-destruction design pattern. The contract itself is modeled using the sub-process *Contract*.

Correct execution of *Safe Selfdestruct* function excludes the contract sub-process which represents the removal of the contract from blockchain. The role *Owner* assigned to *Safe Selfdestruct* is the only role that can execute this function. Furthermore, to safeguard against inadvertent function calls, a time constraint pattern (Sect. 3.4) is used on *Safe Selfdestruct* via the delayed reflexive condition relation $\rightarrow\bullet$ on *Safe Selfdestruct*. Besides, the exclude relation $\rightarrow\%$ from event *Safe Selfdestruct* to *Contract* has a guard condition that only excludes *Contract* if *Safe Selfdestruct* is called more than a predefined *Threshold*. This means if the predefined *Threshold* is bigger than one, each time *Owner* calls *Safe Selfdestruct*, he has to wait *Duration* until the next successful invocation of this function. The *Threshold* could potentially be used to verify enough number of users with *Owner* role have requested for the destruction of the contract.

3.5.3 Example

The *MultiSafeSelfDestructible* contract (Fig. 13) embodies a safe self-destruction usage tailored for scenarios that require consensus among multiple owners before self-destruction. It ensures that the contract can only be terminated once a predefined threshold of approvals from designated owners is met, coupled with a timing constraint to prevent rapid, successive attempts. This particular usage of the safe self-destruction pattern ensures that critical decisions, like contract termination, are made collectively, reducing the risk of unilateral, premature, or accidental execution.

3.6 Access control

3.6.1 Motivation

Unrestricted access to contract functions can lead to unauthorized interactions with the contract and unintended contract behavior. To prevent this, having a robust framework for access control where contract functionalities are securely gated behind roles defined by their specific permissions is crucial.

3.6.2 Model

Access control restricts access to functions to only a subset of accounts, calling them based on the account roles and permissions of each role [6, 7, 9].
Here, we benefit from DCR's native support for access control, modeled directly through roles assigned to activities. Each event in a DCR model can be limited to one or more specific roles. The lack of any role written on an event means every role defined in the model can execute this event. This is very similar to implementing a function in the contract that does not declare any access control.

We have not included a specific model for this pattern as it is used in most of the other models presented in the paper. For instance, Fig. 7 contains two roles *User* and *System* assigned to three activities in the model. When simulating the model, it is possible to assign automatic agents that execute any event having the role as soon as the event is available for execution (depending on relations defined on that event).

In the mentioned scenarios, roles are assigned statically to accounts when a contract is deployed on the blockchain. Certain models may require dynamic assigning or reassigning access rights (execution rights) which is not natively supported by DCR semantics [10]. However, external tools that are built on DCR graphs and extend it enable this operation [59].

3.6.3 Example

A classic example of using access control in smart contracts is the initialization of a variable of type `address` (e.g., `owner`) with the contract deployer's account. Alternatively, for a more democratized solution, Fig. 13 keeps a mapping of owner addresses and checks the access rights at entry point of desired functions using the defined modifier `onlyOwner`. This pattern restricts the execution of certain functions exclusively to the owner, serving as a basic form of access control. For instance, a smart contract governing a voting system might restrict the ability to tally votes or end the voting period to the owner alone, preventing unauthorized manipulation of the voting process.

Similarly, on other blockchain platforms like Algorand [34], this pattern is implemented by asserting who can execute critical functionalities. In Algorand, this is done in PyTeal (a Python library that compiles to Algorand's TEAL [60]) by asserting that the transaction sender matches the application's creator address as line 4 in Fig. 14 presents.[3]

3.7 Commit and reveal

3.7.1 Motivation

In a public permissionless blockchain platform such as Ethereum, transaction data are public [7]. Therefore, if a secret is sent along with a transaction request, participants in the blockchain consensus protocol can see the secret value even before the transaction is finalized. On the other hand, the party holding the secret should commit to it before other parties act, so the secret cannot be changed after the fact.

The commit and reveal pattern addresses this problem and works in two phases. In Phase 1, a piece of data are submitted that depends on the secret (which itself is not yet submitted).

[3] Example is from: https://github.com/barnjamin/pyteal/blob/examples-for-docs/_examples/txn.py#L86-L91.

Fig. 13 Solidity code example of using a safe self-destruction functionality

```
contract MultiSafeSelfDestructible {
  mapping(address => bool) public isOwner;
  mapping(uint256 => mapping(address => bool)) public
      approvedDestruction;
  uint256 public ownerCount = 0; uint256 public destructionAttempt =
       0; uint256 private lastDestructionAttemptTime = 0;
  uint256 public constant THRESHOLD = 2; uint256 public constant
      DURATION = 1 days;
  constructor(address[] memory _owners) {
    require(_owners.length >= THRESHOLD, "Insufficient owners");
    for (uint i = 0; i < _owners.length; i++) {
      require(_owners[i] != address(0) && !isOwner[_owners[i]], "
          Invalid address");
      isOwner[_owners[i]] = true;
      ownerCount++; }}
  modifier onlyOwner() { require(isOwner[msg.sender], "Not an owner"
      ); _; }
  function approveOrInitiateDestruction() external onlyOwner {
    if (lastDestructionAttemptTime + DURATION < block.timestamp) {
      destructionAttempt++;
      lastDestructionAttemptTime = block.timestamp;
      for (uint256 i = 0; i < ownerCount; i++)
        approvedDestruction[destructionAttempt][msg.sender] = false;
    }
    require(!approvedDestruction[destructionAttempt][msg.sender], "
        Already approved");
    approvedDestruction[destructionAttempt][msg.sender] = true;
    uint256 approvalCount = 0;
    for (uint256 i = 0; i < ownerCount; i++) {
      if (approvedDestruction[destructionAttempt][msg.sender])
        approvalCount++;
    }
      if (approvalCount >= THRESHOLD)
        selfdestruct(payable(msg.sender));
  }
}
```

Fig. 14 PyTeal snippet for authorizing application updates

```
def app_update():
  program = Assert(
    Txn.on_completion() == OnComplete.UpdateApplication,
    Global.creator_address() == Txn.sender(),
  )
  # The rest of the logic...
```

Often, that data are the crypto-hash of the secret, such that the secret cannot be reconstructed. Phase 2 is the submission (and reveal) of the secret itself. In between both phases, participants not holding the secret, like the players of a gambling game, can do their actions, e. g., placing a bid.

3.7.2 Model

Figure 15 presents the model of this pattern. Here, event *Reveal* is blocked initially by the condition relation $\rightarrow\bullet$ from *Commit* to *Reveal*, and is enabled once a user commits. The commit makes *Reveal* pending (through the response relation $\bullet\rightarrow$).

The milestone relation $\rightarrow\diamond$ from the pending *Reveal* to *Commit* means that unless a reveal happens, no other commit is possible. When the role *Revealer* executes *Reveal*, it makes event *Decide* pending through the response relation $\bullet\rightarrow$. This again makes it impossible for the *Committer* role to commit another value since there is a milestone relation $\rightarrow\diamond$ from *Decide* to *Commit*. The decision is made using the *Decide* event based on committed and revealed values. After executing *Decide*, the same types of relations prevent a new value being committed before the finalization of decision. Executing either of *Pass* or *Fail*, enables event *Commit* again to commit a new value.

3.7.3 Example

The casino contract presented in Sect. 5.1 incorporates this design pattern. In this contract, bet is placed through the commit phase, and executing *decideBet* function in Fig. 27 reveals the secret.

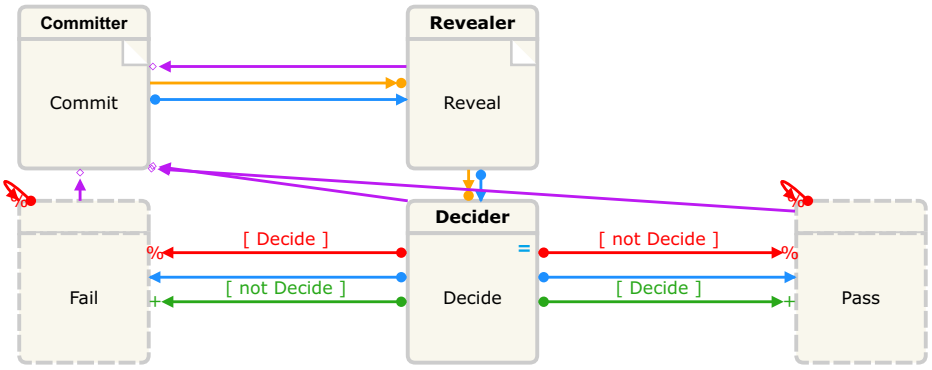


Fig. 15 Commit and reveal design pattern

Blind auction contracts also integrate this design pattern [61]. In a blind auction contract, bidders submit a hash of their bid (commit phase) that keeps the amount secret until the bidding period ends, at which point bids are revealed and the highest bid wins.

3.8 Circuit breaker (emergency stop)

3.8.1 Motivation.

Smart contracts are designed to run autonomously on blockchain platforms. There is no built-in mechanism to stop their execution without modifying the contract code. The circuit breaker pattern, sometimes referred to as *emergency stop* or *pausable*, addresses this by enabling a designated authority (like an *Admin* role in the contract) to temporarily halt the contract's critical functions. This is typically done as an emergency measure in response to potential exploits, discovered bugs, or other severe issues to protecting the contract's state and assets while an investigation or remedy is applied [8].

3.8.2 Model

Figure 16 demonstrates our model for circuit breaker pattern. We have three categories of activities here: activities that are available in the normal execution *(Nominal Operation)* of the contract, those that are only available when the circuit breaker is triggered (activities inside *Circuit Breaker* sub-process), and the event that enables the *Circuit Breaker* (*Panic* event executed by *Monitor* role in Fig. 16). There is a milestone relationship $\rightarrow\diamond$ between *Circuit Breaker* sub-process and all other DCR nodes. The existence of this milestone helps to disable the execution of all of these activities when *Circuit Breaker* is pending (which happens immediately after *Monitor* role executes the *Panic* event). In Fig. 16, event *Panic* executed by *Monitor* role makes *Circuit Breaker* pending. This means that unless event *Revive* is executed, neither *Nominal Operation* nor *Panic* activities are executable anymore. Executing *Contingency* enables a contingency plan.

3.8.3 Example

There have been cases where a vulnerability in the contract triggered by a certain transaction led to funds being locked in the contract [62]. To prevent this, a smart contract may implement a function whose logic is independent of the main contract logic; when triggered, it withdraws all assets in the contract to a certain address. This new address can be the upgraded version of the contract that contains a patch for the vulnerability (cf. Sect. 3.14). This mechanism is known as an "Escapability" or "escape hatch." It serves as a last resort to recover assets or control if the primary contract logic fails or becomes unusable.

The *Contingency* event in our DCR model (Fig. 16) represents the point where such an escape mechanism would be invoked. This is why "Escapability," which we discussed in prior work [1], is now integrated as an example application of the Circuit Breaker pattern rather than being presented as a separate high-level pattern itself. An escape hatch often consists of transferring assets to a predefined safe address [62–64].

3.9 Guard check

3.9.1 Motivation

Smart contracts are reactive services deployed on blockchain that can perform computation and change the global state of blockchain as a result of calls to their functions. When a function is called, the input parameters of the function and contract state variables should be checked before the function body is executed. Besides, the result of the function execution should never violate certain properties within the contract.

Therefore, input validation and invariant checking are performed in functions. This is also known as the guard check design pattern. In Solidity, input validation is performed via the *require* statement and invariants are checked using *assert* [46, 50, 51].

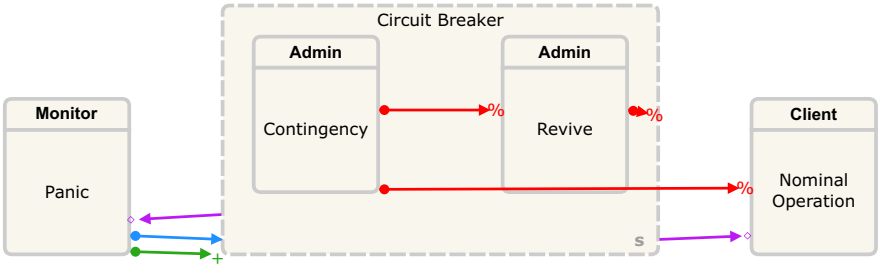


Fig. 16 Circuit breaker design pattern DCR model

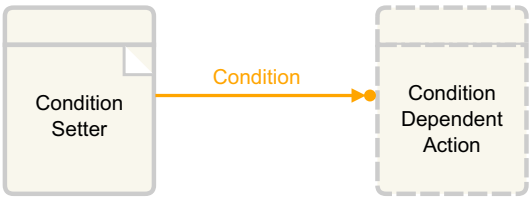


Fig. 17 Model of guard check design pattern

3.9.2 Model

In a smart contract implementation (e.g., using Solidity's `require`), guard checks typically occur *inside* the function body. However, our DCR models capture behavior at a higher level of abstraction, focusing on the interactions and dependencies *between* activities (which map to `public`/`external` function calls or transactions). Therefore, instead of modeling internal checks, the DCR graph represents a guard check as a condition relation →• that must be met for the target event to be enabled. This condition is often determined by the state or data resulting from the execution of previous activities.

To model this, we use the condition relation (→•) in the DCR graph. Specifically, we employ a guarded condition relation, which attaches a boolean expression (the *Condition* in Fig. 17) to the relation's directed edge. In Fig. 17, the *Condition Setter* event represents an action that establishes the relevant state or data. The *Condition Dependent Action* event represents the function being guarded. The guarded →• from *Condition Setter* to *Condition Dependent Action* ensures that *Condition Dependent Action* is only enabled if the specified boolean *Condition* evaluates to true based on the current process state (potentially including the value set by *Condition Setter*).

Figure 17 follows a convention we have adopted for modeling contracts with DCR, which involves assigning the identifier and type of the *Condition Setter* event to match the identifier and type of the bookkeeping variable it affects. This approach allows us to track the impact of activities on bookkeeping variables (utilized in guard checks) without delving into the details of the function(s) the event maps to. The mentioned identifier is useful for specifying the *Condition*.

3.9.3 Example

Examples of this design pattern are used in other patterns. In Fig. 9, the guard *Delay Triggering Condition* on →• is checked before applying the delay. When implemented, this guard check should be performed inside (at the top of) the function that is mapped to *Critical Action*.

3.10 Abstract contract states

3.10.1 Motivation

In most decentralized applications, action dependencies impose a partial order on action executions that a smart contract has to follow. Smart contract programming languages (e. g., Solidity) lack an explicit means to enforce action dependencies. Therefore, a state variable of type enumeration can mimic a finite state automaton, whose state transitions enforce the set of executable functions, encoding a partial order among action executions [7].

3.10.2 Model

DCR models benefit from relations such as condition →•, inclusion →+, and exclude →% that explicitly specify the partial ordering of event executions. Thus, using abstract contract states is unnecessary when modeling contracts in DCR graphs. However, the concept of abstract states can still be explicitly represented in a DCR model. As shown in Fig. 18, this is possible via bundling of multiple activities that are intended to be available during a specific contract state or phase in one sub-process. Figure 18 demonstrates a system with three states of *Initial State*, *Intermediate State*, and *Terminal State*. As the third state does not contain any event, and both other states (and their activities as a result) are excluded by the time *Terminal State* is included, the smart contract remains in *Terminal State* for the rest of its runtime.

3.10.3 Example

Both casino (Sect. 5.1) and escrow (Sect. 5.2) contracts use this design pattern in the implementation. We capture partial action orderings in model of both of these contracts using

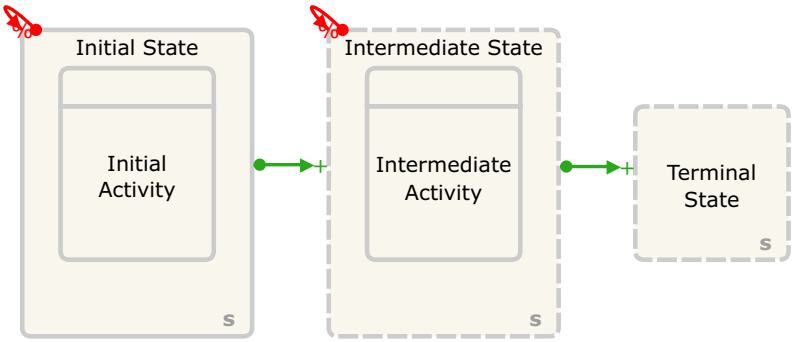


Fig. 18 Abstract contract states design pattern DCR model

DCR relations instead of using abstract contract states. For the escrow example, we include a model that formalizes design of the contract through abstract contract states in DCR graphs (Fig. 28).

3.11 Oracle

3.11.1 Motivation

Blockchains, as the execution platform of smart contracts, separate them from outside networks such as the World Wide Web. Oracles bridge this gap by enabling smart contracts to integrate off-chain data into their execution processes and to export information from blockchains to external systems [6, 7]. A critical requirement for blockchain consensus ecosystem is deterministic execution: every node must compute the exact same result given the same inputs. If smart contracts were to directly read data from variable off-chain sources, different nodes could obtain different data values, leading to divergent execution paths and a failure to reach consensus. Oracles solve this problem by acting as trusted intermediaries that fetch external data and submit it reliably onto the blockchain, ensuring all nodes operate on the same, consistent information [4].

3.11.2 Model

The oracle pattern employs an external call from our main contract to another contract that serves as the on-chain oracle component (data source) to register the request for off-chain data. This registration call information should also be kept in bookkeeping variables inside the oracle contract itself. When the data are ready in the service contract, it informs the main contract about the result by calling a *callback* function from the contract requesting data.

Figure 19 presents our DCR model for the oracle pattern. In case of requesting data from an oracle, data oracles provide data in two ways. Either the data are already published by the oracle contract or the data are to be requested. In the former case, event *Read Data* in the oracle contract should be executed by the *User* role. In case the data are not already published on chain by the oracle ecosystem, *Data User Contract* requests data by executing event *Request Data*. Once the data are prepared by the oracle, it should be pushed to the contract using event *Receive Data*.

3.11.3 Example

A compelling use of oracles is in decentralized finance protocols built using smart contracts. In these protocols, oracles act as a bridge to up-to-date market values of digital assets.

As another use-case, oracle platforms provide an alternative solution for replacing the time incentivization pattern (Sect. 3.1) by triggering smart contract function calls automatically at specific times [54].

3.12 Token design patterns

3.12.1 Motivation

Tokens, serving as the cornerstone of asset representation, behavior, and manageability on distributed ledgers, pave the way for a wide range of applications within smart contract ecosystems. Among these, fungible and non-fungible tokens stand out as two of the primary building blocks for tokenization [6].

Fungible tokens are identical digital assets that can be exchanged without losing value, ideal for currencies and commodities due to their interchangeability, divisibility, and uniformity. Non-Fungible Tokens (NFTs) are unique digital items, each distinct from the other, making them irreplaceable and unsuitable for direct exchange [65].

The design and implementation of tokens on platforms like Ethereum have evolved into standardized patterns, notably the ERC-20 and ERC-721 standards, which address fungible and non-fungible tokens, respectively [66]. These standards facilitate the interoperability and consistency across different applications. There are many more tokenization schemes that include details out-of-scope of the current work. We decided to only include the model related to two FT and NFT token schemes as these two token types are used to build more complex tokenization schemes.

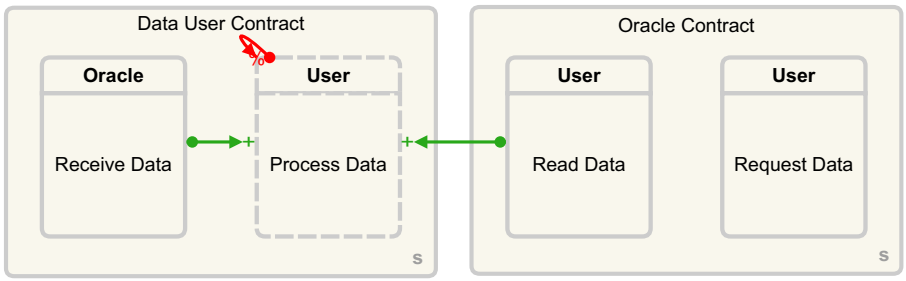


Fig. 19 Oracle design pattern in DCR graphs

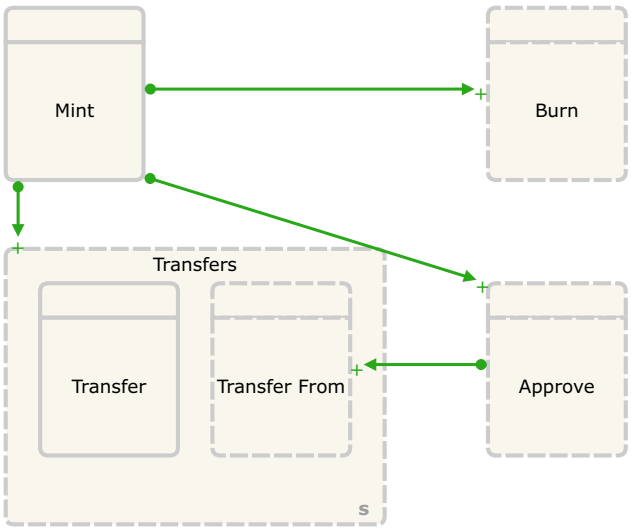


Fig. 20 Token design pattern DCR model

3.12.2 Model

We present our tokenization model in Fig. 20. Both FT and NFT schemes usually depend on a *Mint* event to create new tokens. Depending on the target functionality and type of tokens, *Mint* could be mapped to either a function in the tokenization implementation or the mere action of deploying the contract. It is possible to both transfer a token that belongs to the transaction caller account using event *Transfer* or transfer another account's token by transaction caller account using *Transfer From*. In the latter case, the actual token holder should use *Approve* functionality in the contract to approve the transferring of his tokens by another account. The event *Burn* is used to destroy tokens by the token holder. This event could either be directly mapped to a function with the same name in the contract implementation or to the function implementing event *Transfer*. In latter case, simply the act of transferring the tokens to a burning address acts as burning them.

3.12.3 Example

ERC-721-based NFT scheme empowers users to tokenize digital art, creating unique tokens for each piece to ensure ownership and authenticity. ERC-20 standard allows for creating cryptocurrency fungible tokens that are exchangeable. In both of these scenarios activities in token design patterns—*Mint*, *Transfer*, *Transfer From*, and *Burn*—can be translated to actual functions in contract implementation.

3.13 Pull over push

3.13.1 Motivation

Smart contract functions can be used to send tokens and cryptocurrency to other accounts on blockchain, or call other functions in another contract. In some smart contract execution environments (such as the ones based on Ethereum Virtual Machine), when sending cryptocurrency or tokens via any external call, the receiver may act unexpectedly before returning the control.

The pull over push pattern discourages pushing tokens or cryptocurrency to the destination as a side-effect of calling a function. Rather, it encourages exposing a *Pull* function that users of the contract can call for this reason [7].

3.13.2 Model

As our model in Fig. 21 shows, the contract follows this pattern by providing a function for the users to manually pull the assets (or request the service). In case the continuity of the application is endangered because of lack of interaction from user, one can use time incentivization (Sect. 3.1) or oracle pattern (Sect. 3.11). The difference between the pattern and its anti-pattern (push) is both the role initiating the interaction and function implementation details which is out of scope of DCR graph modeling of activities.

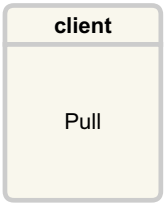


Fig. 21 Pull over push design pattern DCR model

3.13.3 Example

Figure 22 demonstrates both the pull over push pattern and its anti-pattern. Function *distribute* (lines 10–16) which is only callable by the privileged users in the contract contains a call to the *recipient* account holder to *push* the funds, whereas in *withdraw*, the account holder has to *withdraw* manually by calling the function (*pull*).

3.14 Upgradability

3.14.1 Motivation

Smart contracts are implemented within the code segment of blockchain accounts, representing the account's immutable component. This means there is no opportunity for directly patching the contract even after the detection of severe bugs or vulnerabilities.

Upgradability design patterns help with the mentioned problem by segregating the functionality contract from the contract state variables. This typically means keeping the contract state variables in a permanent address and migrating to another account code section to provide the functionality on the permanent contract state variables *(storage* in Ethereum [3]).

3.14.2 Model

Our model in Fig. 23 demonstrates this design pattern. There are three roles here: *Admin, User,* and *System.* We use *System* in modeling smart contracts with DCR graphs whenever we would like to automatically execute an event without the need for interaction (transactions on blockchain) as soon as it is available for execution in the model. This way, when simulating the model, we can automatically execute any event with the role *System*. Role *Admin* can set the logic contracts (either one or two here in our model). Once the logic number (which corresponds to the logic contract address in real blockchain environments) is updated, any call to *Target Function* is forwarded to the event with the same name in the new logic contract. This automatic execution is because of the response relation $\bullet\!\!\rightarrow$ between *Target Function* and the actual function in the new logic contracts. Simply put, when simulated, in a DCR model any pending event (has a $\bullet\!\!\rightarrow$ to it) is executed immediately if automated execution is enabled for the role of that event.

3.14.3 Example

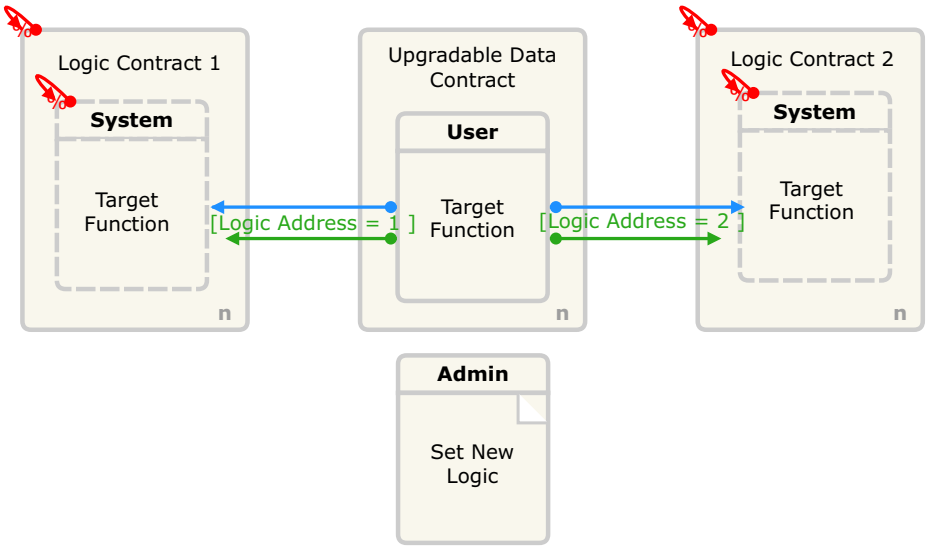
Popular libraries such as OpenZeppelin offer upgradable contract libraries such as proxy upgradable contract [67]. This library performs very similar to our model in Fig. 23 with *proxy* being the same entity as the *Upgradable Data Contract* event. This contract keeps the permanent storage and points to new logic if the contract is upgraded. Since this library is built for Solidity language (EVM-based blockchain platforms), the workaround to implement the proxy contract without actually implementing *Target Function* or determining any information about the function signature, is to capture all calls to the proxy contract using the *fallback* function in *proxy.* Fallback function in Solidity, is the default function that is invoked when no other function matches the *Target Function* signature. Proxy contract forwards the captured call information in the *fallback* function to a function with that same signature in *Logic Contract.* This call forwarding is possible through `delegatecall` that enables the contract to execute the called function within the context of the *proxy* contract's storage, allowing the logic contract to manipulate the proxy's data as if the logic were executing directly within the proxy contract itself. This mechanism ensures seamless upgrades by decoupling the contract's logic from its state, facilitating the evolution of contract functionalities without altering the underlying data structure or contract address.

This pattern should be implemented with caution as malicious or buggy logic contract can potentially overwrite storage slots that belong to access control (Sect. 3.6) [68]. An example of this vulnerability arises from the use of `delegatecall`, where the logic contract executes within the storage context of the proxy contract. This shared storage model can lead to a *storage collision* if the proxy and logic contracts have incompatible assumptions about the layout or types of data stored at specific storage slots [68]. This vulnerability stems from the state variables defined in the logic contract occupying the same storage slots that the proxy contract reserves for its own administrative variables (e.g., `owner` variable). These administrative data often includes the address of the contract owner responsible for authorizing upgrades, as well as the address of the current logic contract implementation itself. If a logic contract writes to a state variable whose storage slot collides with, for example, the proxy's `owner` address slot, it can overwrite this critical access control parameter. An attacker can exploit this by adding their own address as the `owner` to hijack control over the upgrade mechanism and deploy malicious logic updates. Common mitigation strategies include adhering to established proxy standards that reserve specific, well-known storage slots for administrative data (such as those defined in EIP-1967 [69]), ensuring these slots are unlikely to clash with logic contract variables.

Fig. 22 Solidity code example of using an pull over push pattern

```
import "@openzeppelin/contracts/ownership/Ownable.sol";

contract SimpleFundManager is Ownable {
  mapping(address => uint) public balances;

  function deposit() external payable {
    balances[msg.sender] += msg.value;
  }

  function distribute(address recipient) external onlyOwner {
    uint amount = balances[recipient];
    require(amount > 0, "No funds available to push");
    balances[recipient] = 0;
    (bool success, ) = recipient.call{value: amount}("");
    require(success, "Failed to push funds");
  }

  function withdraw() external {
    uint amount = balances[msg.sender];
    require(amount > 0, "No funds available to pull");
    balances[msg.sender] = 0;
    (bool success, ) = payable(msg.sender).call{value: amount}("");
    require(success, "Failed to pull funds");
  }
}
```

Fig. 23 Upgradability design pattern DCR model



3.15 Governance

3.15.1 Motivation

Smart contracts enable decentralized applications by relying on lack of central authority of blockchain. Smart contract logic upgrades (Sect. 3.14) or even pivotal parameter updates (such as predefined *thresholds)* are part of the normal maintenance of the decentralized applications. Enabling a certain role (Sect. 3.6) in the smart contract to arbitrarily perform such code upgrades or parameter updates defeats the purpose of deploying the application in a decentralized architecture.

On-chain governance pattern is used in decentralized protocols to solve the central point of decision making. This pattern allows for decision making on parameter changes and upgrades [6, 28]. This pattern allows token holders or a group of privileged users to submit proposals and vote on them to make decisions that affect the behavior of the contract.

3.15.2 Model

Because of the inherent complexity of this structural design pattern, we present two versions of it: simple governance (Fig. 24) and timed governance (Fig. 25) models.

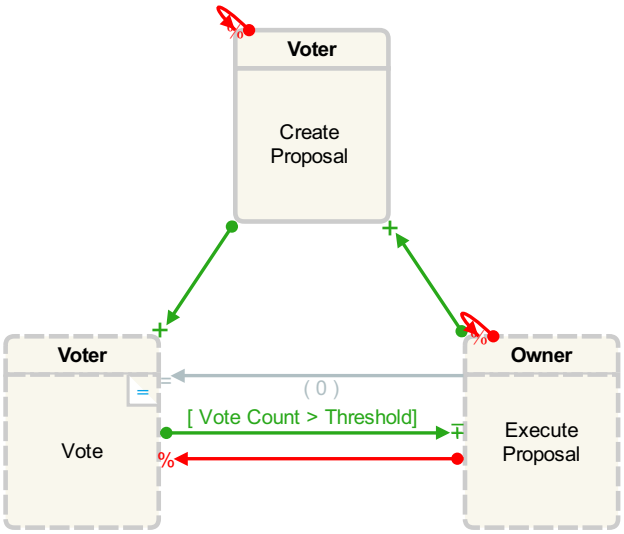


Fig. 24 Simple governance design pattern DCR model

As mentioned earlier this pattern allows for creation of proposals, voting on them and then executing them to take effect. Our model in Fig. 24 contains three activities *Create Proposal, Vote,* and *Execute Proposal.* These activities are directly translatable to functions in contract implementation. We use inclusion →+ and exclusion →% to enforce the correct order of activities (function calls). The event *Vote* is used by token holders or participants of the smart contracts to cast votes. As soon as enough number of votes is cast, *Execute Proposal* is included and activated via guard [*VoteCount* > *Threshold*] on inclusion →+ from *Vote* to *Execute Proposal. Vote Count* denotes the identifier of the *Vote* event to capture the number of votes cast. The equals symbol on top-right corner of *Vote* denotes the computation that happens whenever this event is executed. In this case, the computation is simply increasing the number of total votes by one each time a vote is cast. Furthermore, after executing the proposal *(Execute Proposal)*, value relation →= is used to reset event *Vote* (number of votes) to zero.

The simple governance model (Fig. 24) captures the core functionalities of this pattern. However, the simple version of the model does not handle the following issues well:

- As this pattern allows for submission of proposals, protocol participants can maliciously or inadvertently introduce vulnerabilities or bugs to the protocol by the proposal.
- When voting, a lack of interaction from the participants can endanger the continuation of the governance process.
- As the proposal execution might introduce a significant modification to the application, a mechanism should help the protocol participants to adapt to the modification. For instance, imagine a governance proposal approves a contract upgrade that modifies the application's interface (API). If this change were executed immediately, any external application, script, or integrated service currently calling the old function would instantly fail potentially causing cascading errors. A mechanism should help the protocol participants to adapt to the modification.

To resolve the abovementioned issues, timed governance model in Fig. 25 introduces the following features:

- **Review delay.** To resolve malicious or buggy proposals, a proposal review period is necessary before enabling the proposal voting. Therefore, a time constraint design pattern (Sect. 3.4) is introduced from the *Review* event to the *Vote* event. This pattern is added as *Review Delay Duration* on the mentioned condition relation →•.
- **Voting deprecation.** To resolve the protocol continuity challenge, we introduce an automatic deprecation design pattern (Sect. 3.2) on the *Vote* event. The blue exclamation mark on *Vote* shows this event deprecates in a predefined duration after it is enabled. This duration should be adjusted based on the application.
- **Grace delay.** A mandatory delay after a proposal passes allows participants time to adapt to the changes before new proposal execution. This uses the time constraint pattern (Sect. 3.4). In the model (Fig. 25), this delay is enforced by a timed condition relation (→•) from the intermediate *Grace* event to *Execute Proposal*. This relation, labeled *Grace Delay Duration*, postpones when *Execute Proposal* can be enabled.

The introduction of the intermediate grouping activities *Review* and *Grace* in the timed governance model (Fig. 25) addresses a subtlety in combining timed delays with sequential execution enforced by include/exclude relations in DCR graphs. Consider applying the *Review Delay* directly using a timed condition relation: $\textit{Create Proposal} \xrightarrow{\textit{Review Delay}}\bullet\ \textit{Vote}$. According to DCR semantics (specifically point (2) of the Enabledness criteria described following Definition 1), for *Vote* to become enabled via this timed condition *after* the delay, the source event (*Create Proposal*) must satisfy the check $\textit{Create Proposal} \in \textit{In} \wedge [[d]]_M \implies \mathsf{Ex}(\textit{Create Proposal}) \geq \textit{Review Delay}$. A key part of this check is that the source event (*Create Proposal*) must be included ($\textit{Create Proposal} \in \textit{In}$) when the condition is evaluated after the delay. However, to maintain the correct process flow and prevent actions like creating a new proposal while the first is under review, the *Create Proposal* event usually needs to be excluded (→%) once it has successfully executed. This exclusion might be triggered by *Create Proposal* itself (self-exclusion) or by the event that logically follows it (e.g., the start of the review period). If *Create Pro-*

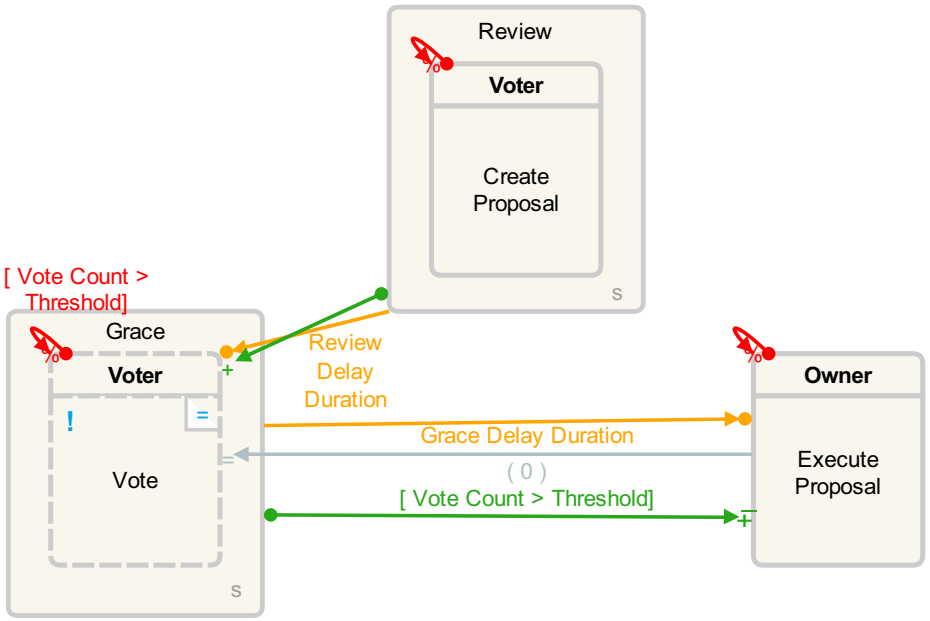


Fig. 25 Timed governance design pattern DCR model

posal becomes excluded *before* the *Review Delay* finishes, the enablement check for *Vote* based on the direct timed condition fails because the prerequisite Create Proposal $\in In$ is no longer true. The *Review* event serves as a persistent state marker or anchor for the delay, circumventing this issue. The sequence works as follows:

- *Create Proposal* executes successfully.
- As an effect, *Create Proposal* includes *Review* ($\rightarrow+$). Now, *Review* $\in In$.
- *Create Proposal* is then excluded (e.g., via *Review*$\rightarrow\%$ *Create Proposal* or a self-exclusion). Now, *Create Proposal* $\notin In$.
- The timed delay is anchored to the *Review* event: *Review* $\overset{Review\ Delay}{\rightarrow\bullet}$ *Vote*.

Since the *Review* event remains included (Review $\in In$) throughout the delay period (it should at least not be excluded until *Vote* potentially occurs), the enablement check (Review $\in In \wedge \ldots$) succeeds after the delay, allowing *Vote* to become enabled. The *Review* event might not represent a distinct user transaction but fulfills this essential structural role within the DCR model to correctly sequence events involving time delays and exclusions. The same logic applies to the *Grace* event, which anchors the *Grace Delay* between the *Vote* event completing and *Execute Proposal* becoming enabled. A similar structural pattern using intermediate activities for timed sequences can be observed in our casino model (Sect. 5.1).

3.15.3 Example

Compound governance is an implementation of the timed governance model (Fig. 25) for decentralized finance (DeFi). It provides a democratic system for managing protocol upgrades, emphasizing community involvement and systematic procedure. In Compound Governance, a token holder with enough COMP tokens (Compound's governance token) or a user sponsored by anyone with enough COMP tokens can submit a proposal [40]. This constraint is not captured by our models (Fig. 24 and Fig. 25).

After a proposal is submitted, it enters a review period, similar to the timed governance model's *Review Delay Duration*. This period allows the community to discuss, debate, and evaluate the merits and potential impact of the proposal before voting begins. This process ensures that proposals are thoroughly vetted and only viable, well-considered changes proceed to the voting phase.

In Compound, voting power is proportional to the amount of COMP tokens a user holds or is delegated. Voting begins after the review period has concluded, marking the start of a fixed voting window. This aligns with the vote event automatic deprecation feature of the timed governance model.

For a vote to be successful, it must not only achieve a majority but also meet a quorum threshold, i. e., a minimum percentage of all voting power must participate. This ensures that approved proposals genuinely represent the community's will. This aligns with guard [*VoteCount* $>$ *Threshold*] in both our governance models.

Once a proposal passes, there is a mandatory delay before the changes are executed, akin to the *Grace Delay Duration*

in the timed governance model. In Compound this feature is referred to as *timelock*.

4 Mapping DCR Models to Solidity Smart Contracts

We present the correspondence between the elements of a DCR graph model (as defined in Definition 1 and used throughout Sect. 3) and the constructs found in typical smart contracts, particularly those written in Solidity for the Ethereum Virtual Machine (EVM). 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.

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, the `createGame` event in the casino model (Fig. 27) maps to the `createGame(bytes32 hash)` function in the Solidity code (Fig. 26).

DCR Role. Roles assigned to DCR events map directly to access control mechanisms within the smart contract (Sect. 3.6). This is commonly implemented using Solidity `modifier`s or `require` statements that check `msg.sender` against authorized addresses (e.g., the `byOp` modifier checking against the `operator` address in Fig. 26, corresponding to the *Operator* role assigned to events like `createGame` in Fig. 27). 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 like the `State` variable, `operator` address, or `hashedNumber` in Fig. 26). 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 (e.g., the `inState(GAME_AVAILABLE)` modifier ensures the `bet` function is only effectively "included" when the contract is in that specific state). 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 $\bullet\!\!\rightarrow$ frequently establishes such pending states (e.g., the obligation to call *decideBet* after *placeBet* is executed, as shown by the $\bullet\!\!\rightarrow$ relation in Fig. 27). Time incentivization patterns (Fig. 5) 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 $\rightarrow\!\!\bullet$ depend on whether or when a previous action occurred (e.g., needed for timed conditions or ensuring *Commit* precedes *Reveal* in the commit-reveal pattern, Fig. 15).

DCR Enabledness (Conditions). The enabledness criteria for a DCR event, determined by relations like condition $\rightarrow\!\!\bullet$ and milestone $\rightarrow\!\!\diamond$, correspond directly to the precondition 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 ($\rightarrow\!\!+$, $\rightarrow\!\!\%$, $\bullet\!\!\rightarrow$, $\rightarrow\!\!=$, $\bullet\!\!\rightarrow\!\!\times$) model the state changes and side effects resulting from the successful execution of a smart contract function's body.

- Include $\rightarrow\!\!+$ / Exclude $\rightarrow\!\!\%$ map to changes in state variables that logically enable or disable the preconditions for future function calls (e.g., updating the `State` enum variable in Fig. 26 effectively includes/excludes functions guarded by the `inState` modifier; see also explicit state transitions modeled in Fig. 18).
- Response $\bullet\!\!\rightarrow$ maps to setting state that signifies an obligation or expected follow-up action is now active.
- Value $\rightarrow\!\!=$ maps to direct assignments (=) that update state variables based on function inputs, internal computations, or values from `msg.value`.
- Cancel $\bullet\!\!\rightarrow\!\!\times$ 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 like the automatic deprecation (Fig. 6) or the time incentivization (Fig. 5) patterns.

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 design patterns (Sect. 3) and the contract models presented in Sect. 5, and it forms the basis for formal analysis and verification techniques, such as the runtime monitoring approach using HighGuard mentioned in Sect. 6. Besides formal analysis, this correspondence can be used in a pipeline to automatically generate boilerplate smart contract code from DCR graph specifications. At a high level, such a pipeline would operate in three stages: **(i) Extraction:** parse the DCR model (e.g., from the DCR Solutions portal's XML export or from DCR-JS) to obtain the set of events, roles, relations, guards, and timing constraints; **(ii) Mapping:** apply the correspondence defined in Section 4 to translate each DCR element into Solidity constructs—events become function signatures, roles become `modifier` definitions checking `msg.sender`, condition relations →• become `require` statements checking execution history and timing via `block.timestamp`, and include/exclude effects become updates to state variables governing function availability; **(iii) Emission:** generate a Solidity contract skeleton containing the function stubs, modifiers, state variables (including an enumeration for abstract contract states where applicable), and the corresponding `require` guards, leaving the developer to fill in application-specific computation logic within each function body. The feasibility of this approach is supported by existing work on translating DCR graphs to smart contract code for other platforms (e.g., the DCR-to-TEAL translation by Xu et al. [70]). Realizing and evaluating such a pipeline for Solidity, including measuring the fraction of contract code that can be automatically generated versus manually written, is a concrete next step.

5 Modeling and analysis of implemented smart contracts

In this section, we delve into the modeling of smart contracts that have been implemented in Solidity. The DCR models presented for these contracts were derived through manual analysis of their source code. This process involved identifying the core business logic, roles, `public`/`external` functions, state transitions, access control mechanisms, and relevant design patterns, then translating this behavior into the DCR graph formalism. Crucially, this modeling adheres to the mapping between DCR elements and smart contract constructs detailed in Sect. 4.

By modeling three specific smart contracts in the upcoming subsections (a casino in Sect. 5.1, an escrow in Sect. 5.2, and a digital marketplace in Sect. 5.3), we demonstrate how a complete smart contract's high-level behavior can be formalized through its DCR model. Furthermore, this endeavor shows how utilizing and combining the DCR models of several design patterns (formalized in Sect. 3) into one cohesive model captures the intended smart contract design.

5.1 Casino

We present the model of a simple casino contract [71].[4] It uses four design patterns identified in Table 1: time incentivization, role-based access control, commit and reveal, and abstract contract states.

The casino contract in Fig. 26 has two explicitly declared roles, `operator` and `player`. It also contains three abstract states (see Sect. 3.10): `IDLE`, `GAME_AVAILABLE`, and `BET_PLACED`. Three modifiers check the pre-conditions →• of each function based on the roles and the state the contract is in.

Figure 27 shows the DCR model of this contract. The activities all reflect functions of the same name, except sub-process *Casino*, which everything is grouped under. This sub-process reflects the behavior of the deployed contract, which includes a suicidal `closeCasino` function that `selfdestruct`s, shown by an exclusion relation from *closeCasino* to the sub-process in Fig. 27. Without a sub-process, an exclusion relation would go from *closeCasino* to all other activities, which is visually unappealing. Furthermore, we do not model the actual states of the contract, instead choosing to order the activities by inclusion →+ and exclusion →% relations.

When the casino contract is deployed, it is initially in abstract *IDLE* state. It is possible to create a game, add to the pot, remove from it, or self-destruct in this state.

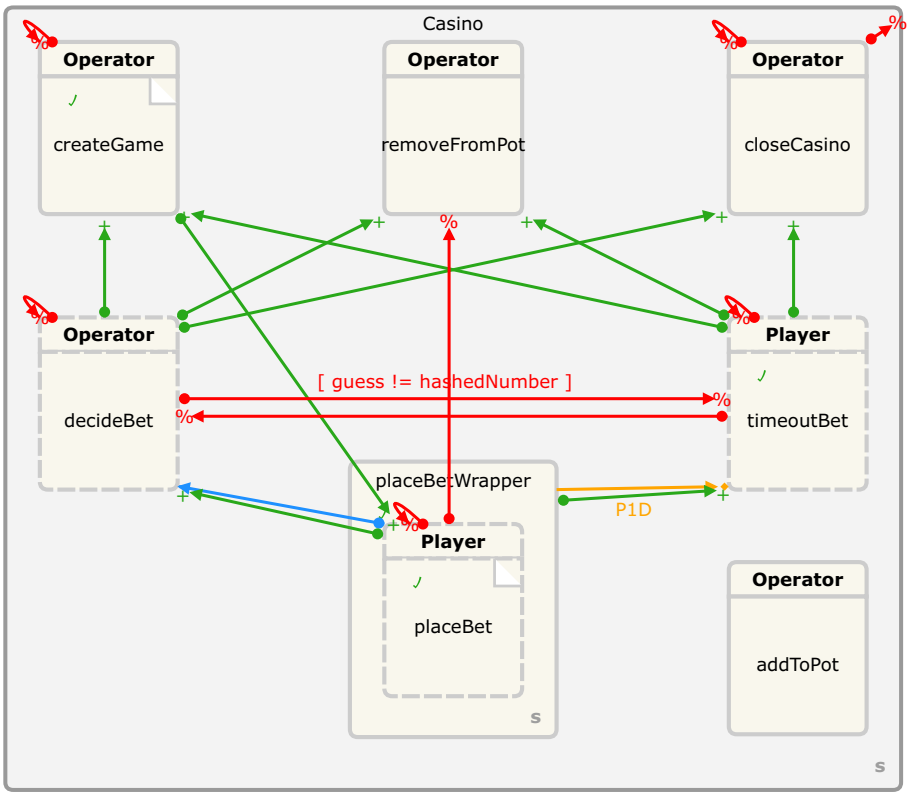
Creating a game changes the abstract state to `GAME_AVAILABLE`, which enables anyone in the Ethereum

[4] The scenario was originally provided by Gordon Pace [72].

Fig. 26 Solidity code for Casino contract (some details and full implementation omitted, minor corrections applied for illustration)

```
contract Casino {
  address public operator, player; bytes32 public hashedNumber;
  enum State { IDLE, GAME_AVAILABLE, BET_PLACED };
  State private state;  uint bet;  Coin guess;
  function addToPot() public payable byOp {...}
  function removeFromPot(uint amt) public byOp, noActiveBet {...}
  function createGame(bytes32 hash) public byOp, inState(IDLE) {
    hashedNumber = hash
    state = GAME_AVAILABLE;}
  function bet(Coin _guess) public payable inState(GAME_AVAILABLE) {
    require (msg.sender != operator);
    require (msg.value > 0 && msg.value <= pot);
    player = msg.sender; bet = msg.value;
    guess = _guess; state = BET_PLACED;}
  function decideBet(uint secret) public byOp, inState(BET_PLACED) {
    require (hashedNumber == keccak256(secret));
    Coin secret = (secret% 2 == 0)? HEADS : TAILS;
    if (secret==wager.guess) {playerWins();} else {operatorWins();}
    state = IDLE;}
}
```

Fig. 27 Casino contract model



network to place a bet and take the role of the player (as a result). The function `decideBet` checks if the player is the winner by comparing the guess with the secret number. This gives both the player and the operator a 50% chance of winning the game. In the model, a response relation $\bullet\!\!\rightarrow$ from *placeBet* to *decideBet* emphasizes that *decideBet* has to execute at some point and should not block the game from continuing. However, since continuing the game at this point depends on the operator making a transaction, it is possible for a malicious operator or buggy reverted `decideBet` function to lock the funds the player puts in the game. Furthermore, after a player places the bet, the operator should not be able to change the actual secret stored. Therefore, three patterns are used in the model to provide the following functionality:

1. A time incentivization pattern (Sect. 3.1) ensures that continuing the game is the favorable option for the casino operator. Figure 27 shows the required mechanism, where *timeoutBet* becomes available with a desirable delay (here

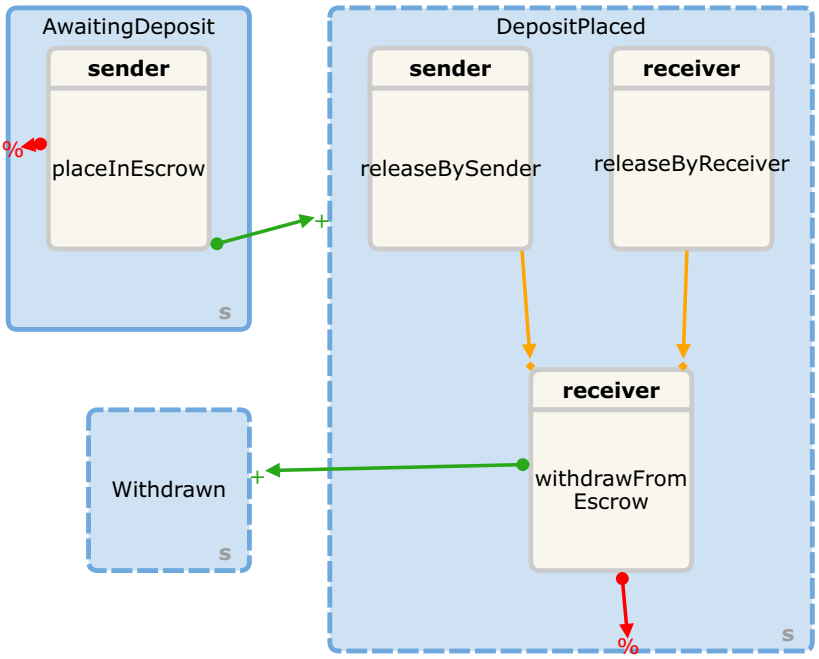


Fig. 28 Escrow contract model using states

P1D, one day) to provide the player with an option when the operator is unable or unwilling to make a transaction to *decideBet*. Calling this function after the timeout guarantees the player wins the game and motivates the operator to decide the game in time.

2. A commit and reveal pattern (Sect. 3.7) is used to ensure when operator *createGame* is called, the operator commits to a secret without sending it. The revealing phase of this pattern is performed in *decideBet*, where the secret is submitted, checked, and compared to the player's guess.
3. A role-based access control pattern (Sect. 3.6) to confine *Player* and *Operator* roles to their respective activities.

The abstract contract states pattern (Sect. 3.10) used in the implementation (Fig. 26) is not needed in the model (Fig. 27): The model's partial ordering provides the same semantics without the complications of abstract contract states.

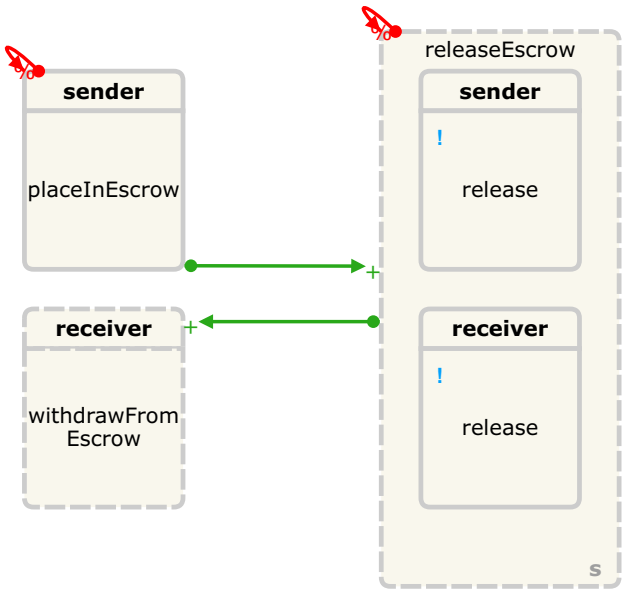


Fig. 29 Escrow contract model using simple partial orderings (inclusion/exclusion)

5.2 Escrow

In finance, an escrow is an entity in which assets are held by a third party on behalf of two other parties that are in the process of completing a transaction. The assets are held in escrow until the terms of the contract or agreement between the two parties are fulfilled, at which point the assets are released to the appropriate party.

We demonstrate the process of modeling a simple escrow contract with two approaches [73]. The escrow contract has two explicitly defined roles *sender* and *receiver*, and three abstract states *AwaitingDeposit*, *DepositPlaced*, and *Withdrawn*. State *Withdrawn* is a sink state and makes the escrow single-use.

As this contract uses abstract contract states (Sect. 3.10), the first approach is to model the abstract states from the implementation. This makes the model less compact but reflects the actual implementation. Each abstract state is modeled with a sub-process that groups all the activities that are available in that state together. As depicted in Fig. 28, when model is simulated the only included-by-default component

is *AwaitingDeposit*, therefore, the only executable actions are the ones included by this sub-process (abstract state).[5] Here, execution of *placeInEscrow* excludes its parent sub-process and includes the next sub-process *DepositPlaced* at the same step. The actual functions in the implementation that are enabled in this state are *releaseEscrowBySender, releaseEscrowByReceiver,* and *withdrawFromEscrow*. Next, to enable a withdrawal by the receiver, both sender and receiver should release it. Each of the release functions can have implementation details related to conditions under which this release happens and are not relevant to the abstract level we are specifying the behavior. Last but not the least, a *withdraw* by the receiver puts the contract in a sink state.

The exact same behavior is specified by only using exclusion →%, inclusion →+, and pending response •→ relations in the next approach in Fig. 29. As it is demonstrated in this figure, modeling partial orderings using inclusion/exclusion of activities makes it much easier to understand the individual behavior of activities (functions in Solidity).

5.3 Decentralized marketplace

The `DecentralizedMarketplace` contract (Figure 31) introduces a platform for trading items managed by transactions between sellers and buyers under the oversight of an arbitrator. The contract functionality includes buying, shipping, and financial settlements, including dispute resolution mechanisms where the arbitrator decides on refunds or seller compensation.

This contract includes seven functions, all of which are present in the DCR model (Fig. 30) as DCR activities with the same name as the function identifier in the contract source code.

Four of the presented high-level design patterns are used in this contract model:

1. A guard check pattern (Sect. 3.9) applies to inclusion →+ arrows from `resolveDispute` function to `refundToBuyer` and `compensateSeller` functions.
2. Pull over push (Sect. 3.13) is used when `arbitrator` role includes →+ other activities to transfer (pull) Ether instead of directly pushing Ether to `seller` or `buyer`. In reality, both activities included by the `arbitrator` role (`compensateSeller` and `refundToBuyer`) in Fig. 31 correspond with the *Pull* event in Fig. 21 as they allow the users to request their assets/Ether rather than pushing them to the user account.
3. Access control (Sect. 3.6) ensures that each role executes only its permissioned activities. In the beginning, `buyItem` does not have a specific role assigned to it, as all parties can initiate buying regardless of their role.
4. Abstract contract states (Sect. 3.10) are used in the contract implementation (Fig. 31) to impose the correct ordering of function calls. As demonstrated in Sect. 5.2, we directly model this event ordering using relation between activities instead.

The model simulation execution allows to both automatically and manually simulate event execution (step-by-step). The swimlane chart in Fig. 32 is generated by the model execution engine as a trace of its execution. This chart demonstrates the roles performing each event in one model execution (in each row). Moreover, events executed by different roles are stacked on top of each other; thus, time is not strictly moving from left to right. As the chart shows (Fig. 32), the user with *seller* role decides to dispute the process after shipping the item, which is followed by *arbitrator* deciding (by executing *resolveDispute*) leading to seller being able to pull (Sect. 3.13) its compensation.

6 Related work

As the current work focuses on both formalization of smart contracts in general and specifically their high-level design patterns, we categorically reviewed both literature on smart contract design patterns as well as general formalization and modeling of smart contracts.

6.1 Formalization of smart contracts

As identified by previous research, modeling formalisms are considered to either belong to high-level (also known as contract-level or business process-level in literature) and low-level (also known as program-level) categories [1, 74]. The high-level models concentrate on the higher-level behavior of a contract, overlooking specific technical details regarding its implementation and execution. As we classify DCR graphs as a high-level formalism, we review the most important formalisms of this type.

State transition systems have been heavily leveraged as one such formalism, a method that resonates with the programming paradigms compatible with Solidity and is used in Obsidian and Bamboo languages [25, 75, 76]. Validating these state transition models typically involves the use of model checkers against temporal logic properties [77]. Alqahtani et al. focus on visual modeling of smart contract interactions using Behavioral Interaction Priority (BIP) [78, 79].

Parallel to state transition systems, Petri nets offer an intuitive way to represent behaviors, facilitating the visualization, creation, and simulation of smart contract models [80, 81].

[5] In this model, sub-processes are presented with blue color to emphasize usage of sub-process semantics in DCR model of this design pattern.

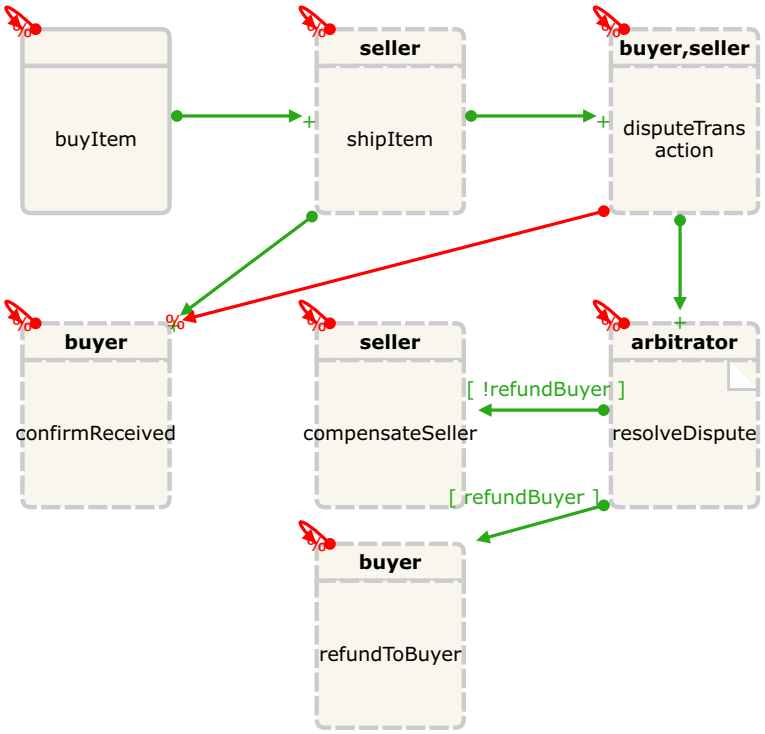


Fig. 30 Decentralized marketplace contract (Fig. 31) DCR model

Garfatta et al. use colored Petri nets to model system dynamics and data dependencies of smart contracts and benefits DCR graphs to capture the business process environment the smart contract is running [82, 83].

Temporal behavior of smart contracts can be accurately captured by timed automata [84, 85]. This approach aids in understanding the impact of time on the execution of a smart contract and the constraints that ensue (time-based design patterns in the current work). Another perspective to this is introduced through the modeling of the nondeterministic execution of smart contracts. This involves utilizing probabilistic transition systems, such as Markov decision processes, to offer a view of potential user behaviors and their repercussions [86–88].

Suvorov and Ulyantsev [89] promote a *correct-by-construction* method for smart contract generation using FSMs. They model contracts with FSMs and linear temporal logic (LTL) specifications and successfully transform these models into Solidity code. VeriSolid [77] presents a framework for designing and verifying Ethereum contracts by representing them as transition systems. It includes an automated verification loop with property translation to Computation Tree Logic (CTL). Specification refinement is allowed if a model fails verification until all properties are satisfied, automatically generating the corresponding Solidity code.

A key advantage of using a declarative formalism like DCR graphs is the ability to adjust the level of modeling detail. This flexibility arises from several aspects of the DCR formalism. Firstly, the granularity of events is adaptable; an event can represent a high-level business phase or, as primarily used in this paper, a specific smart contract function call (Sect. 4). Secondly, DCR naturally abstracts implementation details within an event. For example, an access control restriction (Sect. 3.6) is modeled by assigning a role to an event, capturing the behavioral constraint without specifying how the check is coded (e.g., `modifier` vs. inline `require` in Solidity [25]). Thirdly, the modeler can selectively include data elements and relations only when they are essential for representing control flow logic, such as in guard conditions (Sect. 3.9). Finally, DCR's support for sub-processes (Sect. 2.2) provides a powerful mechanism for hierarchical modeling, allowing complex parts of a process to be represented as a single event at a higher level, thus managing complexity and detail explicitly. Our focus on high-level design patterns (Sect. 3) exemplifies choosing a specific, abstracted level of detail suitable for analyzing the core interaction logic and business process of smart contracts.

6.2 Transaction dependencies

Smart contracts often involve multiple dependent transactions, a challenge that has been addressed through various approaches. Sergey and Hobor discuss the non-deterministic nature of transaction ordering decided by miners [90], while other works focus on commutativity conditions to exploit interleavings [91] or identify serializable transactions in Ethereum [92]. These issues have also been modeled using finite state automata, which can lead to "bad states" in certain

Fig. 31 Solidity code for a decentralized marketplace contract (some details are omitted)

```
contract DecentralizedMarketplace {
  enum State {Listed, Pending, Shipped, Completed, Disputed}
  struct Item {uint id; address payable seller; address payable
      buyer; uint price; State state; bool decisionMade;}
  uint public itemCount; mapping(uint => Item) public items;
  address public arbitrator;
  mapping(uint => bool) public refundDecisions;
  modifier onlySeller(uint _itemId) { require(msg.sender == items[
      _itemId].seller, "Only seller can call this.");_; }
  modifier onlyBuyer(uint _itemId) { require(msg.sender == items[
      _itemId].buyer, "Only buyer can call this.");_; }
  modifier inState(uint _itemId, State _state) { require(items[
      _itemId].state == _state, "Invalid state.");_; }
  function buyItem(uint _itemId) external payable inState(_itemId,
      State.Listed) {
    Item storage item = items[_itemId];
    require(msg.value == item.price, "Incorrect price.");
    item.buyer = payable(msg.sender);
    item.state = State.Pending; }
  function shipItem(uint _itemId) external onlySeller(_itemId)
      inState(_itemId, State.Pending) {
    items[_itemId].state = State.Shipped; }
  function confirmReceived(uint _itemId) external onlyBuyer(_itemId)
       inState(_itemId, State.Shipped) {
    Item storage item = items[_itemId];
    item.seller.transfer(item.price);
    item.state = State.Completed; }
  function disputeTransaction(uint _itemId) external inState(_itemId
      , State.Shipped) {
    require(msg.sender == items[_itemId].buyer  msg.sender == items[
        _itemId].seller, "Not a transaction party.");
    items[_itemId].state = State.Disputed; }
  function resolveDispute(uint _itemId, bool _refundBuyer) external
      inState(_itemId, State.Disputed) {
    require(msg.sender == arbitrator, "Only the arbitrator accepted.
        ");
    Item storage item = items[_itemId];
    item.decisionMade = true;
    refundDecisions[_itemId] = _refundBuyer; }
  function refundToBuyer(uint _itemId) external inState(_itemId,
      State.Disputed) {
    Item storage item = items[_itemId];
    require(item.decisionMade && refundDecisions[_itemId], "Refund
        conditions not met.");
    require(msg.sender == item.buyer, "Only the buyer can initiate
        the refund.");
    item.buyer.transfer(item.price);
    item.state = State.Completed; }
  function compensateSeller(uint _itemId) external {
    Item storage item = items[_itemId];
    require(item.decisionMade && !refundDecisions[_itemId], "
        Compensation conditions not met.");
    require(msg.sender == item.seller, "Only the seller can claim
        compensation.");
    item.seller.transfer(item.price); item.state = State.Completed
        ;}}
```

scenarios (e. g., when most of the actions are not accessible) [93]. DCR graphs offer a more elegant solution in such cases. Chen et al. use graphs to analyze transactional dependencies and security in smart contracts, but their approach is statistical and less precise than ours [94].

6.3 Design patterns

Research on design patterns covers reusable solutions to common challenges in decentralized software design using smart contracts [5].

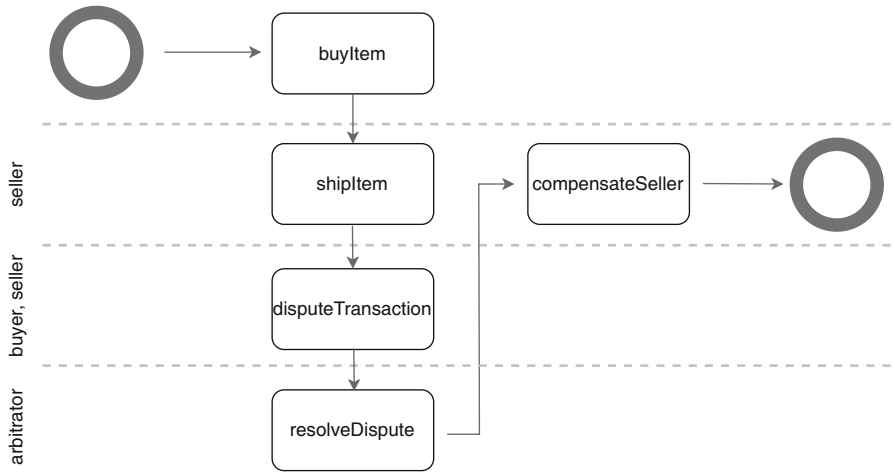


Fig. 32 An execution trace of the decentralized marketplace smart contract model

Most of existing work on smart contract design patterns merely categorize and review patterns with no focus on formalization or verification of the design patterns [6–9, 28, 36, 95].

The work by Wohrer and Zdun [7] is among the only works that formalizes the design pattern "pull over push" (Sect. 3.13). They formalize this design pattern using their domain-specific language. The formalized pattern is then used for automatic Solidity code generation [96].

The upgradability design pattern (Sect. 3.14) allows for patching of contract, but it does not guarantee the correctness of the patched contract. Antonino et al. present a smart contract specification refinement and verification framework based on the upgradability design pattern [97–99]. Their framework is based on a trusted deployer that formally verifies the contract with regards to the specifications.

6.4 Distributed workflow execution on smart contracts

The Business Collaboration Language (BCL) was proposed in 2016 as a data-aware business process language on a blockchain [100]. The mentioned work puts forth the fact that data is not the focus of processes in traditional business process modeling, whereas blockchain-based applications are data-centric. Effectively, a blockchain is a replicated database. To face this issue, we categorize the design patterns in smart contract applications into *high-level* and *low-level* patterns (Table 1). This allows us to put focus on *high-level* design patterns that are akin to traditional business processes and are thus less data-centric.

Frantz and Nowostawski suggest specification and interpretation of smart contracts should be accessible to a broader audience [101]. Hence, they propose *codification of law* directly from natural language. They demonstrate a semi-automated generation of Ethereum smart contracts from an institutional behavior specification. This specification is in the form of natural language that follows the *Grammar of Institutions* [102]. The metaphorical use of *grammar* merely suggests a systematic way of specifying rules that govern institutions. Statements in these specifications are constructed of five types of components which are automatically mapped to Solidity language constructs.

Some works suggest directly executing DCR graph workflows on blockchains, such as Madsen et al. embedding a DCR execution engine on Ethereum [103]. Xu et al. [70] provide an automatic translation of business process models specified in DCR graphs to TEAL programs [34, 60], a Turing-incomplete smart contract language built for the Algorand blockchain virtual machine.

Existing approaches largely concentrate on enabling the execution of business process workflows on blockchains. Compared to these, we formalize design patterns commonly used in the smart contract developer community and noticed by prior research [6–8, 28–31]. We demonstrate the usefulness of these patterns by modeling real-world contracts that use them (cf. Sect. 5).

Instead of focusing on distributed workflow execution, our work facilitates specification and formal analysis of smart contract correctness and security.

6.5 Formal analysis using DCR models

As mentioned in Sect. 1, DCR specifications provide a basis for automated analysis, to *verify* that the implementation of the contracts adheres to their models.

We have implemented HIGHGUARD [17], a runtime monitoring tool that captures transactions from the Ethereum client and, in tandem, executes the corresponding actions within an instance of the DCR graph model. For each Ethereum transaction, HIGHGUARD checks if the DCR graph model permits a corresponding action in the model. If this is not the case, the tool reports a violation. When leveraging runtime information, our framework enables auto-

mated runtime verification. While runtime verification in the blockchain domain is typically associated with the performance overhead of contract or platform instrumentation [52, 104, 105], HighGuard address this concern by placing the monitor off-chain. If any deviations from the specification are detected, HighGuard generates alerts that can be used to enhance the contract's implementation or enable a circuit breaker pattern (cf. Sect. 3.8).

Machine-readable DCR specifications for a decentralized application can serve as an oracle for synthesis tasks. XploGen [106] uses the program comprehension and synthesis capabilities of large language models (LLMs) given the formal model of the smart contract in DCR graphs and inject business logic vulnerabilities in the contract source code that divert the behavior of the contract from its intended business process model [106, 107]. These vulnerable contract versions and the corresponding concrete exploits generated by XploGen are useful to benchmark vulnerability detection tools for business logic vulnerabilities [50, 58].

7 Discussion

Formalizing smart contract patterns using DCR graphs requires a discussion of the challenges and complexities in our declarative modeling approach. We discuss the issues such as combinability of the models and non-interference. Additionally, we cover practical considerations such as model usability, learning curve, and scalability in model representation and execution.

7.1 Model composition and interference

The composition of DCR models representing different patterns involves merging their respective graph structures (taking the union of events, roles, relations, etc.) and ensuring their initial markings are compatible (i.e., agree on the state of shared events). Formally, this corresponds to the composition operator ("||") defined in a prior work [108]. The behavior of the resulting composite graph is governed by the standard DCR execution semantics (Sect. 2.2.2) : an event e is enabled in the composed system only if it satisfies the conditions imposed by *all* relations involving e originating from *any* of the merged component graphs. When an enabled event e executes, its resulting effects on the shared marking (e.g., making other events included, excluded, or pending) are the combined union (denoted $\oplus$ in [108]) of the effects specified by each component relation triggered by e.

While syntactic composition is straightforward, semantic interference is a significant concern. Constraints introduced by one pattern (e.g., a timing constraint, an access control rule, an exclusion relation) can interact with the events and constraints of another pattern, potentially leading to unintended consequences like the violation of a pattern's intended logic. For example, a time incentivization deadline (Sect. 3.1) might conflict with a speed bump delay (Sect. 3.4). In practice, when composing pattern models for a specific contract (as demonstrated in Sects. 5.1 to 5.3), we recommend the following lightweight checks to mitigate interference risks: **(1) Shared-event audit:** identify all events that appear in more than one pattern and verify that their combined include/exclude effects are consistent (i.e., no pattern excludes an event that another pattern requires to remain included for correctness); **(2) Timing compatibility:** for events subject to both a delay (from a condition relation $\rightarrow\bullet$) and a deadline (from a response relation $\bullet\rightarrow$ originating in a different pattern), confirm that the delay does not exceed the deadline, which would make the event unreachable; **(3) Role consistency:** ensure that roles assigned to shared events across patterns do not inadvertently restrict access beyond what each pattern individually intends. These checks can be performed manually during design or partially automated using the static analysis capabilities of existing DCR tooling (cf. Sect. 7.3). While they do not constitute a full formal non-interference proof, they address the most common sources of unintended interaction observed in our case studies.

Identifying all interfering compositions requires analyzing the composite DCR model. Formal approaches exist, such as checking for refinement, which ensures that the composed process does not introduce new behaviors observable via the alphabet of the original process [108, Definition 24]. However, verifying refinement is computationally complex (NP-hard for the basic DCR), highlighting the difficulty of guaranteeing non-interference formally [108, Theorem 27]. Our case studies demonstrate that several patterns can be successfully combined (cf. Sect. 5). However, we do not provide a systematic analysis of all possible interactions between the 15 formalized patterns. A deeper investigation is needed to understand the compositional properties of these patterns. This includes developing stricter non-interference conditions—possibly going beyond existing notions like *non-invasiveness*, which syntactically prevents new components from adding include ($\rightarrow+$) or exclude ($\rightarrow\%$) relations targeting events in existing components [108]. Formally verifying whether pattern-specific properties are preserved when patterns are composed remains an important direction for future work.

7.2 Limitations of DCR pattern modeling

By focusing on platform-independent, business logic patterns, our DCR models abstract away implementation details. This provides clarity on the intended workflow, interactions, and temporal constraints. However, it is important to acknowledge the trade-offs. While DCR models the *behav-*

ior of patterns like circuit breaker (Sect. 3.8) or upgradability (Sect. 3.14), the implementation complexities and risks associated with these patterns, such as storage collisions [68] in proxy patterns or the centralization concerns [4] inherent in upgradability and oracle design patterns remain considerations during protocol design and implementation that fall outside the scope of this work.

Even regardless of implementation-level considerations, the literature highlights the insufficiency of relying solely on design patterns for security (without considering their abstraction level), as they cover only a limited subset of known vulnerabilities [31, 109]. This reinforces the value of formalizing full contract designs (cf. Sect. 5) to enable further analysis and verification tasks.

Furthermore, a perfect one-to-one correspondence between DCR constructs and the full spectrum of smart contract language features does not always exist. For instance, while DCR's guarded relations (Sect. 3.9) can model more than one pre-condition, expressing highly complex state dependencies, involving intricate calculations across multiple state variables sometimes seen in DeFi protocols might become cumbersome within DCR's declarative framework, potentially requiring complex guard expressions or numerous intermediate computation events. Similarly, DCR's data model typically abstracts the specifics of complex on-chain data structures, like nested mappings or large arrays of structures. Cryptographic primitives essential to some patterns (e.g., *keccak256* in Sect. 3.7 or signature checks) are usually treated as opaque functions within the DCR model rather than being supported natively. Likewise, while interactions with external contracts or oracles (Sect. 3.11) can be represented by events like *Request* and *Receive*, the model simplifies the asynchronous nature and potential failure modes. This shows a potential mismatch where the richer computational and data manipulation capabilities of languages like Solidity (which is built for the data-centric blockchain ecosystem) may exceed what can be elegantly captured solely through DCR's primary focus on process flow and event dependencies.

7.3 Smart contract model complexity

Recognizing the potential complexity of large, flat models, the DCR graph formalism incorporates hierarchical modeling features to structure models and manage complexity [15]. Research explicitly identifies hierarchy as a key mechanism for handling complexity in process modeling [110]. DCR supports sub-processes as a form of hierarchical modeling (cf. Sect. 2.2). Sub-processes allow complex workflows to be broken down into smaller, potentially reusable modules.

Furthermore, a transaction to smart contract may involve nested function calls, which make it challenging to model as DCR events. We approach this problem by only modeling the business logic of the contract using DCR graphs and overlooking the details of the function implementations. This means we only model the partial-order dependencies at the level of transactions rather than function calls. To achieve this, we conventionally only model `public`/`external` functions as activities in DCR graphs (cf. Sect. 4).

7.3.1 The learning curve

Translating an informal design of the contract into a precise, unambiguous formal specification demands a grasp of both the pattern's semantics and the formal language's constructs. DCR explicitly models concepts like roles, partial ordering of events, and temporal attributes, which are often core components of smart contract logic and high-level patterns. These features map more naturally to smart contract semantics. Moreover, prior research indicates DCR is perceived as more useful than other formalisms such as Declare [111, 112]. The development of user-friendly tools, such as the web-based DCR-JS editor [113], and the availability of training courses and resources [114] also facilitate learning and using DCR graphs for smart contract modeling.

7.3.2 Tooling for DCR graphs modeling and analysis

The practical application and management of complexity in any formal notation heavily depend on the capabilities of its tool ecosystem. DCR graphs benefit from mature tooling developed over years of academic research and commercial application [114]. Key tools include the DCR Solutions portal,[6] and libraries like DCR4Py [115] and DCR-JS [113]. These provide features for managing model complexity:

- *Simulation* allows users to explore dynamic behavior step-by-step, observe enabled and pending events, and validate logic through interactive scenarios [116].
- *Static analysis* modules support formal verification of properties such as liveness and deadlock-freedom [116].
- *Process discovery* techniques, such as DisCoveR [117], can infer constraints from execution traces, aiding in mining formal models.
- *Graphical editors* with features like syntax highlighting, state feedback, and visualizations improve usability [114].

To illustrate how these capabilities apply concretely, consider the casino model (Fig. 27). A key safety property is that the player's funds are never permanently locked. Formally, whenever the event *placeBet* has been executed, and the contract has not been closed, it must always be possible to eventually reach a state where either *decideBet* or *timeoutBet* is enabled. In DCR terms, this is a liveness property:

[6] https://dcrgraphs.net

Table 2 Runtime impact of off-chain vs. on-chain DCR graphs

Aspect	Off-Chain	On-Chain
Verification	External to blockchain	In-contract enforcement
Performance	No gas cost [17]	High gas usage [118]
Scalability	Depends on off-chain infrastructure	Limited by chain resources
Use-Case	HighGuard [17], XploGen [106]	–

after *placeBet* executes, *decideBet* becomes pending (via the response relation $\bullet\!\!\rightarrow$), and the model must guarantee that this pending obligation can be resolved. Using the static analysis module, one can verify that no reachable marking exists in which both *decideBet* and *timeoutBet* are both excluded and non-enabled, while *decideBet* remains pending—confirming that the time incentivization pattern (Sect. 3.1) indeed prevents fund locking. Similarly, a basic safety property—that *decideBet* is never enabled before *createGame* has been executed—follows directly from the condition relation $\rightarrow\!\!\bullet$ from *createGame* to *decideBet* and can be checked by the tool's reachability analysis. These examples demonstrate that even without full model checking against temporal logic formulas, the structural properties of DCR graphs and their existing tool support enable meaningful verification of critical smart contract properties.

7.3.3 Effects on smart contract runtime

Scalability of using DCR-based solutions with smart contract runtime is also another concern. As Table 2 demonstrates, putting the DCR model execution directly on-chain (embedded in smart contracts) incurs gas costs but allows for direct access to runtime information. We suggest separating design and analysis of DCR graphs and their execution platform (off-chain column in Table 2) to avoid additional fees or limit the analysis capabilities.

8 Conclusion and future work

Smart contracts are critical yet complex pieces of software that encode business processes in an executable form on a blockchain. We collected 15 high-level smart contract design patterns that dissect complex smart contracts into smaller reusable components, making it easier to reason about them. DCR graphs are an ideal way to formally model the semantics of these patterns, supporting the concepts of time, data, and roles. We therefore formalized these 15 design patterns using DCR graphs in detail. We demonstrate the usefulness of the patterns and their DCR formalization by modeling 3 full contracts that collectively employ 6 of these design patterns.

The contract DCR models serve as reusable, platform-independent templates for developing correct and secure smart contracts across various applications and smart contract execution platforms. This not only aids in the initial design phase but also has uses in monitoring the contract behavior, allowing for automated verification [17, 106], which reduces the risk of vulnerabilities. Future directions of our research include an extensive evaluation of the HighGuard tool, combinations of static or dynamic analysis of low-level patterns [52] with runtime monitoring against our high-level design patterns, as well as automated discovery of the models from the contract transaction history.

As a future direction, we suggest exploring the automatic generation of boilerplate smart contract code based on the DCR graphs. This involves translating the relations captured in the DCR models directly into constructs like `require` statements in Solidity code. This direct boilerplate contract generation ensures the Solidity code satisfies the DCR specifications.

Acknowledgements Part of this work has been funded by grant 2019-0458 ("TEMOS: Temporal Monitoring of Smart Contracts") by the Swedish Science Council (*Vetenskapsrådet*), as well as *Vetenskapsrådet* grant 2019-04951 ("X-LEGAL: Smart Legal Contracts").

Funding Open access funding provided by Royal Institute of Technology.

References

1. Eshghie, M., Ahrendt, W., Artho, C., Hildebrandt, T.T., Schneider, G.: Capturing Smart Contract Design with DCR Graphs. In: Ferreira, C., Willemse, T.A.C. (eds.) Software Engineering and Formal Methods. pp. 106–125. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-47115-5_7
2. Szabo, N.: Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought (16) **18**(2), 28 (1996)
3. Ethereum Yellow Paper (Dec 2022), https://github.com/ethereum/yellowpaper, accessed: 2023–08-29

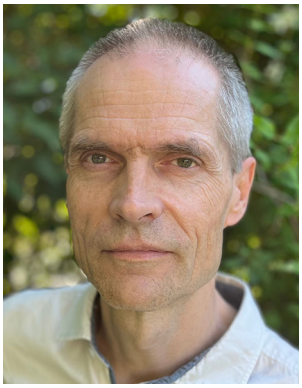
4. Eshghie, M., Jafari, M., Artho, C.: From Creation to Exploitation: The Oracle Lifecycle. In: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (March 2024)
5. Gamma, E., Helm, R., Johnson, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Deutschland GmbH (1995)
6. Bartoletti, M., Pompianu, L.: An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. In: Financial Cryptography and Data Security. pp. 494–509. LNCS, Springer, Cham (2017)
7. Wöhrer, M., Zdun, U.: Design Patterns for Smart Contracts in the Ethereum Ecosystem. In: iThings/GreenCom/CPSCom/SmartData. pp. 1513–1520 (2018)
8. Wohrer, M., Zdun, U.: Smart contracts: security patterns in the Ethereum ecosystem and Solidity. In: IEEE IWBOSE. pp. 2–8 (2018)
9. Fravoll: Solidity Patterns (2023), https://fravoll.github.io/solidity-patterns/, accessed: 2023–08-29
10. Hildebrandt, T.T., Mukkamala, R.R.: Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In: Honda, K., Mycroft, A. (eds.) Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010. EPTCS, vol. 69, pp. 59–73 (2010). https://doi.org/10.4204/EPTCS.69.5
11. Hildebrandt, T.T., Normann, H., Marquard, M., Debois, S., Slaats, T.: Decision Modelling in Timed Dynamic Condition Response Graphs with Data. In: Business Process Management Workshops. pp. 362–374. Springer, Cham (2022)
12. Costa Seco, J., Debois, S., Hildebrandt, T., Slaats, T.: RESEDA: Declaring Live Event-Driven Computations as REactive SEmi-Structured DAta. In: 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC). pp. 75–84 (2018). https://doi.org/10.1109/EDOC.2018.00020
13. Hildebrandt, T., Mukkamala, R.R., Slaats, T., Zanitti, F.: Contracts for cross-organizational workflows as timed dynamic condition response graphs. J. Logic Algebraic Program. **82**(5), 164–185 (2013). https://doi.org/10.1016/j.jlap.2013.05.005
14. Basin, D., Debois, S., Hildebrandt, T.: In the Nick of Time: Proactive Prevention of Obligation Violations. pp. 120–134. IEEE (2016). http://csf2016.tecnico.ulisboa.pt/, iEEE Computer Security Foundations Symposium, CSF; Conference date: 27–06-2016 Through 01–07-2016 https://doi.org/10.1109/CSF.2016.16
15. Debois, S., Hildebrandt, T., Slaats, T.: Hierarchical Declarative Modelling with Refinement and Sub-Processes. In: Sadiq, S., Soffer, P., Völzer, H. (eds.) Business Process Management, pp. 18–33. Springer International Publishing, Cham (2014)
16. Normann, H., Debois, S., Slaats, T., Hildebrandt, T.T.: Zoom and Enhance: Action Refinement via Subprocesses in Timed Declarative Processes. In: BPM 2021. pp. 161–178. Springer, Cham (2021)
17. Eshghie, M., Artho, C., Stammler, H., Ahrendt, W., Hildebrandt, T., Schneider, G.: Highguard: Cross-chain business logic monitoring of smart contracts. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. p. 2378–2381. ASE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3691620.3695356
18. Marchesi, L., Marchesi, M., Destefanis, G., Barabino, G., Tigano, D.: Design Patterns for Gas Optimization in Ethereum. In: IEEE IWBOSE. pp. 9–15 (2020)
19. Liu, Y., Li, Y., Lin, S.W., Artho, C.: Finding permission bugs in smart contracts with role mining. In: SIGSOFT ISSTA 2022. p. 716–727. ACM (2022)
20. Unprotected Ether Withdrawal - SWC-105 - Smart Contract Weakness Classification (SWC), https://swcregistry.io/docs/SWC-105/, accessed: 2023–09-01
21. Unprotected SELFDESTRUCT - SWC-106 - Smart Contract Weakness Classification (SWC), https://swcregistry.io/docs/SWC-106/, accessed: 2023–09-01
22. Transaction Order Dependence - SWC-114 - Smart Contract Weakness Classification (SWC), https://swcregistry.io/docs/SWC-114/, accessed: 2023–09-01
23. Timestamp Dependence - Ethereum Smart Contract Best Practices, https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/timestamp-dependence/#avoid-using-blocknumber-as-a-timestamp, accessed: 2023–09-01
24. SWC-116 - Smart Contract Weakness Classification (SWC), https://swcregistry.io/docs/SWC-116/#time_locksol, accessed: 2023–09-01
25. Solidity documentation (Aug 2023), https://docs.soliditylang.org/en/latest/, accessed: 2023–08-29
26. Morello, G., Eshghie, M., Bobadilla, S., Monperrus, M.: DISL: Fueling Research with A Large Dataset of Solidity Smart Contracts (2024). https://doi.org/10.48550/arXiv.2403.16861
27. Nute, D.: Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 3, chap. Defeasible Logic. Clarendon Press, Oxford University Press (1994)
28. Liu, Y., Lu, Q., Zhu, L., Paik, H.Y., Staples, M.: A Systematic Literature Review on Blockchain Governance. Journal of Systems and Software 197 (2023)
29. Hu, B., Zhang, Z., Liu, J., Liu, Y., Yin, J., Lu, R., Lin, X.: A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. Patterns **2**(2), 100179 (2021). https://doi.org/10.1016/j.patter.2020.100179
30. Mühlberger, R., Bachhofner, S., Castelló Ferrer, E., Di Ciccio, C., Weber, I., Wöhrer, M., Zdun, U.: Foundational Oracle Patterns: Connecting Blockchain to the Off-Chain World. In: Asatiani, A., García, J.M., Helander, N., Jiménez-Ramírez, A., Koschmider, A., Mendling, J., Meroni, G., Reijers, H.A. (eds.) Business Process Management: Blockchain and Robotic Process Automation Forum. pp. 35–51. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58779-6_3
31. Azimi, S., Golzari, A., Ivaki, N., Laranjeiro, N.: A systematic review on smart contracts security design patterns. Empir. Softw. Eng. **30**(3), 95 (2025). https://doi.org/10.1007/s10664-025-10646-w
32. Paritytech/ink (2024), https://github.com/paritytech/ink
33. Vyper, https://docs.vyperlang.org/en/latest/
34. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In: Proceedings of the 26th symposium on operating systems principles. pp. 51–68 (2017)
35. Consensys: Ethereum Smart Contract Best Practices (2023), https://consensys.github.io/smart-contract-best-practices/development-recommendations/precautions/, accessed: 2023–08-29
36. System Design - Smart Contract Security Field Guide, https://scsfg.io/developers/system-design/
37. OpenZeppelin: OpenZeppelin Contracts, https://github.com/OpenZeppelin/openzeppelin-contracts, accessed: 2023–08-29
38. Aragon OSx Protocol (Jun 2023), https://github.com/aragon/osx, accessed: 2023–08-29
39. Solidstate: SolidState Solidity (2023), https://github.com/solidstate-network/solidstate-solidity/blob/de7c9545ac015f42a03aa3a678000ec1ec4c14a4/contracts/access/access_control/AccessControl.sol, accessed: 2023–08-29
40. Compound Protocol (Jun 2023), Compound, accessed: 2023–08-29

41. Smartcontractkit/chainlink (2023), https://github.com/smartcontractkit/chainlink, accessed: 2023–08-29
42. Augur (Aug 2023), https://github.com/AugurProject/augur, accessed: 2023–08-29
43. The Maker Protocol White Paper | Feb 2020, https://makerdao.com/en, accessed: 2023–08-29
44. Aragon/aragon-court (Jul 2023), Aragon, accessed: 2023–08-29
45. Synthetixio/synthetix: Synthetix Solidity smart contracts, https://github.com/Synthetixio/synthetix, accessed: 2023–08-29
46. etherscan.io: HOLDIT | Etherscan, http://etherscan.io/address/0x24021d38DB53A938446eCB0a31B1267764d9d63D, accessed: 2023–08-29
47. Chainbridge-solidity (Aug 2023), https://github.com/ChainSafe/chainbridge-solidity, accessed: 2023–08-29
48. Compound: Compound v2 Governance, https://docs.compound.finance/v2/governance/, accessed: 2023–08-29
49. Eshghie, M., Artho, C., Gurov, D.: Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning. In: EASE 2021. pp. 305–312. ACM (2021)
50. Eshghie, M., Morello, G., Lauretano, M., Bartel, A., Monperrus, M.: Flames: Fine-tuning llms to synthesize invariants for smart contract security (2025), arxiv:2510.21401
51. Eshghie, M., Mazura, M., Bartel, A.: 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 (May 2026), arxiv:2512.22616, to appear. Preprint available at
52. Gao, J., Liu, H., Liu, C., Li, Q., Guan, Z., Chen, Z.: EASYFLOW: Keep Ethereum Away from Overflow. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 23–26 (May 2019). iSSN: 2574–1934 https://doi.org/10.1109/ICSE-Companion.2019.00029
53. Sandhu, R.S.: [role-based access control]. In: Advances in Computers, vol. 46, pp. 237–286. Elsevier (1998)
54. Solidity Cron Jobs, https://docs.chain.link/chainlink-nodes/oracle-jobs/job-types/cron/, accessed: 2023–08-29
55. Openzeppelin-contracts/contracts/governance/Governor.sol at db97666d0bd3794e9880aaa7a3d9c7df148f436b · OpenZeppelin/openzeppelin-contracts, https://github.com/OpenZeppelin/openzeppelin-contracts/blob/db97666d0bd3794e9880aaa7a3d9c7df148f436b/contracts/governance/Governor.sol
56. Harvey, C.R., Ramachandran, A., Santoro, J.: DeFi and the Future of Finance. John Wiley & Sons (2021)
57. etherscan.io: 0x302fE87B56330BE266599FAB2A54747299B5aC5B | Etherscan, http://etherscan.io/address/0x302fE87B56330BE266599FAB2A54747299B5aC5B, accessed: 2023–08-29
58. Rezaei, H., Eshghie, M., Anderesson, K., Palmieri, F.: SoK: Root Cause of $1 Billion Loss in Smart Contract Real-World Attacks via a Systematic Literature Review of Vulnerabilities (2025), arxiv:2507.20175
59. Activity effects, https://documentation.dcr.design/documentation/activity-effect/, accessed: 2025–04-10
60. The Algorand Virtual Machine (AVM) and TEAL. - Algorand Developer Portal. https://developer.algorand.org/docs/get-details/dapps/avm/teal/specification/
61. Solidity by Example – Solidity 0.8.26 documentation, https://docs.soliditylang.org/en/latest/solidity-by-example.html
62. Explained: The Akutars NFT incident (2022) - Halborn Blockchain Security Firm: Ethical hackers, Infosec & pen tests, https://halborn.com/blog/post/explained-the-akutars-nft-incident-april-2022, accessed: 2023–08-29
63. A Decentralized Escape Hatch for DAOs, https://hackingdistributed.com/2016/07/11/decentralized-escape-hatches-for-smart-contracts/, accessed: 2023–08-29
64. Implement escape hatch mechanism contracts · Issue #1 · OpenZeppelin/openzeppelin-contracts, https://github.com/OpenZeppelin/openzeppelin-contracts/issues/1, accessed: 2023–08-29
65. Kugler, L.: Non-fungible tokens and the future of art. Commun. ACM **64**(9), 19–20 (2021). https://doi.org/10.1145/3474355
66. ERC, https://eips.ethereum.org/erc
67. Proxy Upgrade Pattern - OpenZeppelin Docs. https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies
68. Ruaro, N., Gritti, F., McLaughlin, R., Grishchenko, I., Kruegel, C., Vigna, G.: Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) 2024 (2024)
69. ERC-1967: Proxy Storage Slots. https://eips.ethereum.org/EIPS/eip-1967
70. Xu, Y., Slaats, T., Düdder, B., Debois, S., Wu, H.: Distributed and Adversarial Resistant Workflow Execution on the Algorand Blockchain. In: Matsuo, S., Gudgeon, L., Klages-Mundt, A., Perez Hernandez, D., Werner, S., Haines, T., Essex, A., Bracciali, A., Sala, M. (eds.) Financial Cryptography and Data Security. FC 2022 International Workshops. pp. 583–597. Springer International Publishing, Cham (2023). https://doi.org/10.1007/978-3-031-32415-4_35
71. Eshghie, M.: Casino Contract Source, https://github.com/mojtaba-eshghie/HighGuard/blob/925bf9c9afe344c763963e0e40098c66420d1d6a/server/monitor/contracts/source/Casino.sol, accessed: 2025–04-14
72. Ellul, J., Pace, G.J.: Runtime verification of ethereum smart contracts. In: 2018 14th European Dependable Computing Conference (EDCC). pp. 158–163 (2018). https://doi.org/10.1109/EDCC.2018.00036
73. Eshghie, M.: Escrow Contract Source, https://github.com/mojtaba-eshghie/HighGuard/blob/7b81b45e2ceec94da1a69a505bfe7b1369c4361e/server/monitor/contracts/source/escrow.sol, accessed: 2025–04-14
74. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Comput. Surv. **54**(7), 148:1-148:38 (2021). https://doi.org/10.1145/3464421
75. Obsidian: The Obsidian Smart Contract Language - Obsidian 0.1 documentation (2023), https://obsidian.readthedocs.io/en/latest/
76. Hirai, Y.: Bamboo: A language for morphing smart contracts (2023), https://github.com/pirapira/bamboo
77. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: VeriSolid: Correct-by-Design Smart Contracts for Ethereum (Jan 2019). https://doi.org/10.48550/arXiv.1901.01292
78. Alqahtani, S., He, X., Gamble, R., Mauricio, P.: 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
79. Bliudze, S., Cimatti, A., Jaber, M., Mover, S., Roveri, M., Saab, W., Wang, Q.: Formal Verification of Infinite-State BIP Models. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) Automated Technology for Verification and Analysis. pp. 326–343. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_25
80. Duo, W., Xin, H., Xiaofeng, M.: Formal analysis of smart contract based on colored petri nets. IEEE Intell. Syst. **35**(3), 19–30 (2020). https://doi.org/10.1109/MIS.2020.2977594
81. Liu, Z., Liu, J.: Formal Verification of Blockchain Smart Contract Based on Colored Petri Net Models. In: 2019 IEEE 43rd Annual

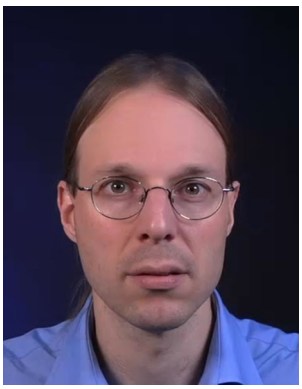
Computer Software and Applications Conference (COMPSAC). vol. 2, pp. 555–560. IEEE, New York, NY, USA (2019). https://doi.org/10.1109/COMPSAC.2019.10265
82. Garfatta, I., Klai, K., Graïet, M., Gaaloul, W.: Model Checking of Solidity Smart Contracts Adopted for Business Processes. In: Service-Oriented Computing. pp. 116–132. Lecture Notes in Computer Science, Springer, Cham (2021). https://doi.org/10.1007/978-3-030-91431-8_8
83. Jensen, K., Kristensen, L.M., Wells, L.: Coloured petri nets and CPN tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transfer **9**(3), 213–254 (2007). https://doi.org/10.1007/s10009-007-0038-x
84. Abdellatif, T., Brousmiche, K.L.: 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). pp. 1–5. IEEE, New York, NY, USA (2018). https://doi.org/10.1109/NTMS.2018.8328737
85. Clack, C.D., Vanca, G.: Temporal Aspects of Smart Contracts for Financial Derivatives (2018). https://doi.org/10.48550/arXiv.1805.11677
86. Ron, M.: On the specification and verification of atomic swap smart contracts (2018). https://doi.org/10.48550/arXiv.1811.06099
87. Laneve, C., Coen, C., Veschetti, A.: 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, pp. 397–415. Lecture Notes in Computer Science, Springer International Publishing, New York, NY, USA (2019). https://doi.org/10.1007/978-3-030-30985-5_23
88. Bigi, G., Bracciali, A., Meacci, G., Tuosto, E.: Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods. In: Bodei, C., Ferrari, G., Priami, C. (eds.) Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday, pp. 142–161. Lecture Notes in Computer Science, Springer International Publishing, (2015). https://doi.org/10.1007/978-3-319-25527-9_11
89. Suvorov, D., Ulyantsev, V.: Smart contract design meets state machine synthesis: case studies (2019). https://doi.org/10.48550/arXiv.1906.02906
90. Sergey, I., Hobor, A.: A Concurrent Perspective on Smart Contracts (2017), arxiv:1702.05511
91. Bansal, K., Koskinen, E., Tripp, O.: Automatic Generation of Precise and Useful Commutativity Conditions. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 115–132. Lecture Notes in Computer Science, Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_7
92. Dickerson, T., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding Concurrency to Smart Contracts. In: PODC. pp. 303–312. ACM (2017)
93. Ellul, J., Pace, G.J.: Runtime Verification of Ethereum Smart Contracts. In: 2018 14th European Dependable Computing Conference (EDCC). IEEE (2018). https://doi.org/10.1109/EDCC.2018.00036
94. Chen, T., Li, Z., Zhu, Y., Chen, J., Luo, X., Lui, J.C.S., Lin, X., Zhang, X.: Understanding ethereum via graph analysis. ACM TOIT **20**(2), 1–32 (2020)
95. Liu, Y., Lu, Q., Xu, X., Zhu, L., Yao, H.: Applying Design Patterns in Smart Contracts. In: Chen, S., Wang, H., Zhang, L.J. (eds.) Blockchain - ICBC 2018. pp. 92–106. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-94478-4_7
96. Wohrer, M., Zdun, U.: From domain-specific language to code: smart contracts and the application of design patterns. IEEE Softw. **37**(5), 37–42 (2020). https://doi.org/10.1109/MS.2020.2993470
97. Antonino, P., Ferreira, J., Sampaio, A., Roscoe, A.W., Arruda, F.: A refinement-based approach to safe smart contract deployment and evolution. Softw. Syst. Model. (2024). https://doi.org/10.1007/s10270-023-01143-z
98. Meyer, B.: Applying design by contract. Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
99. ERC-2535: Diamonds, Multi-Facet Proxy, https://eips.ethereum.org/EIPS/eip-2535
100. Hull, R., Batra, V.S., Chen, Y.M., Deutsch, A., Heath III, F.F.T., Vianu, V.: Towards a Shared Ledger Business Collaboration Language Based on Data-Aware Processes. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) Service-Oriented Computing. pp. 18–36. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-46295-0_2
101. Frantz, C.K., Nowostawski, M.: From Institutions to Code: Towards Automated Generation of Smart Contracts. In: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W). pp. 210–215 (2016). https://doi.org/10.1109/FAS-W.2016.53
102. Crawford, S.E., Ostrom, E.: A grammar of institutions. Am. Polit. Sci. Rev. **89**(3), 582–600 (1995)
103. Madsen, M.F., Gaub, M., Høgnason, T., Kirkbro, M.E., Slaats, T., Debois, S.: Collaboration Among Adversaries: Distributed Workflow Execution on a Blockchain. In: Symposium on Foundations and Applications of Blockchain. vol. 20 (2018)
104. Ma, F., Fu, Y., Ren, M., Wang, M., Jiang, Y., Zhang, K., Li, H., Shi, X.: EVM: From Offline Detection to Online Reinforcement for Ethereum Virtual Machine. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 554–558 (Feb 2019). iSSN: 1534–5351 https://doi.org/10.1109/SANER.2019.8668038
105. Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts (2018). arxiv:abs/1801.04032 [cs], https://doi.org/10.48550/arXiv.1801.04032
106. Eshghie, M., Artho, C.: 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. p. 2240–2248. ASE '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3691620.3695292
107. SunWeb3Sec: DeFi Hacks Reproduce - Foundry, https://github.com/SunWeb3Sec/DeFiHackLabs, Accessed: 26 March 2025
108. Debois, S., Hildebrandt, T.T., Slaats, T.: Replication, refinement & reachability: complexity in dynamic condition-response graphs. Acta Informatica **55**(6), 489–520 (2018). https://doi.org/10.1007/s00236-017-0303-8
109. Zhang, C., Budgen, D.: What Do we know about the effectiveness of software design patterns? IEEE Trans. Software Eng. **38**(5), 1213–1231 (2012). https://doi.org/10.1109/TSE.2011.79
110. De Masellis, R., Di Francescomarino, C., Ghidini, C., Maggi, F.M.: Declarative process models: Different ways to be hierarchical. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) Service-Oriented Computing, pp. 104–119. Springer International Publishing, Cham (2016)
111. Jalali, A.: Evaluating user acceptance of knowledge-intensive business process modeling languages. Softw. Syst. Model. **22**(6), 1803–1826 (2023). https://doi.org/10.1007/s10270-023-01120-6
112. Pesic, M.: Constraint-based Workflow Management Systems: Shifting Control to Users. Ph.d. thesis, Eindhoven University of Technology, Eindhoven, The Netherlands (2008). https://doi.org/10.6100/IR638413

113. Tamo, L.K., Abbad-Andaloussi, A., Trinh, D.M.T., López-Acosta, H.A.: An open-source modeling editor for declarative process models. In: International Conference on Cooperative Information Systems 2023. p. 5. CEUR-WS (2023)
114. Dcr graphs documentation. https://documentation.dcr.design/documentation/
115. Hermansen, S., Jónsson, R., Kjeldsen, J., Slaats, T., Cosma, V., López, H.: DCR4Py: A PM4Py Library Extension for Declarative Process Mining in Python. In: Proceedings of the ICPM 2024 Tool Demonstration Track. CEUR Workshop Proceedings, vol. 3783. CEUR-WS (2024), 6th International Conference on Process Mining, ICPM; Conference date: 14–10-2024 Through 18–10-2024
116. Debois, S., Hildebrandt, T., Marquard, M., Slaats, T.: The DCR Graphs Process Portal. In: Azevedo, L., Cabanillas, C. (eds.) Proceedings - BPM 2016 Demonstration Track. pp. 7–11. CEUR Workshop Proceedings, ceur workshop proceedings (2017), bPM Demo Track 2016, BPMD 2016; Conference date: 21–09-2016 Through 21–09-2016
117. Slaats, T.: DisCoveR: Process Mining for Knowledge-Intensive Processes with DCR Graphs. vol. 3424. ceur workshop proceedings (2023), publisher Copyright: 2022 Copyright for this paper by its authors.; 2023 Joint of the Workshop on Algorithms and Theories for the Analysis of Event Data and the International Workshop on Petri Nets for Twin Transition, ATAED and PN4TT 2023; Conference date: 25–06-2023 Through 30–06-2023
118. Ellul, J., Pace, G.J.: Runtime Verification of Ethereum Smart Contracts. In: 2018 14th European Dependable Computing Conference (EDCC). pp. 158–163 (Sep 2018). https://ieeexplore.ieee.org/abstract/document/8530777https://doi.org/10.1109/EDCC.2018.00036

Mojtaba Eshghie received his PhD in Computer Science from KTH Royal Institute of Technology, Sweden. From September 2025 to February 2026, he was a Postdoctoral Researcher at Umea University, Sweden. His research interests include AI-based software analysis, smart contract security, and formal verification. He has developed tools and methodologies combining machine learning with formal methods, including runtime monitoring tools and vulnerability detection. He has supervised numerous Master's, bachelor's and co-supervised PhD students.



Wolfgang Ahrendt is professor in Computer Science at Chalmers University of Technology, Gothenburg, Sweden. He has a long track record in formal software verification, with a focus on mainstream programming languages. He co-developed the KeY system, the number one deductive verification tool for Java software, and co-authored an according book. Ahrendt also contributed to the integration of formal verification with lighter-weight techniques, like testing and runtime checking. His recent research focus is on smart contract analysis and on combining AI-assisted programming with formal methods. Ahrendt chairs the steering committee of the iFM (Integrated Formal Methods) conference and is on the steering committee of TAP (Test and Proofs).



Cyrille Artho is Associate Professor at KTH Royal Institute of Technology, Sweden. His main interests are software verification and software engineering. In his Master's thesis and his Ph.D. thesis at ETH Zurich, Artho investigated different approaches for finding faults in multi-threaded programs. In particular, in joint work with NASA Ames, Artho found that high-level data races can show potential problems in concurrent software even if individual data accesses are safe. After obtaining his Ph.D., Artho moved to Tokyo, where he worked for two years at the National Institute of Informatics as a Postdoctoral Researcher. From April 2007 till July 2016, Artho worked as Senior Researcher at the National Institute of Advanced Industrial Science and Technology (AIST) in Tokyo and Osaka. Artho's recent work includes model-based testing with the tool "Modbat", smart-contract execution analysis, and IoT security. He is also member of the management group of the KTH Center for Cyber Defense and Information Security.

Thomas Hildebrandt is Associate Professor at KTH Royal Institute of Technology, Sweden. His main interests are software verification and software engineering. In his Master's thesis and his Ph.D. thesis at ETH Zurich, Artho investigated different approaches for finding faults in multi-threaded programs. In particular, in joint work with NASA Ames, Artho found that high-level data races can show potential problems in concurrent software even if individual data accesses are safe. After obtaining his Ph.D., Artho moved to Tokyo, where he worked for two years at the National Institute of Informatics as a Postdoctoral Researcher. From April 2007 till July 2016, Artho worked as Senior

Researcher at the National Institute of Advanced Industrial Science and Technology (AIST) in Tokyo and Osaka. Artho's recent work includes model-based testing with the tool "Modbat", smart-contract execution analysis, and IoT security. He is also member of the management group of the KTH Center for Cyber Defense and Information Security.

Gerardo Schneider is professor of computer science at the Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Sweden. He received a PhD degree in computer science from the Université Joseph Fourier (VERIMAG lab), Grenoble, France. He has held appointments at Uppsala University (Sweden), IRISA/INRIA Rennes (France), and the University of Oslo (Norway). His research interests are in formal methods in a broad sense including software verification (model checking and runtime verification), formal specification and analysis of (legal) contracts, and privacy.