



# Model Checking of Solidity Smart Contracts Adopted for Business Processes

Ikram Garfatta<sup>1,2(✉)</sup>, Kaïs Klai<sup>2</sup>, Mohamed Graïet<sup>3</sup>, and Walid Gaaloul<sup>4</sup>

<sup>1</sup> University of Tunis El Manar, National Engineering School of Tunis, OASIS, Tunis, Tunisia

<sup>2</sup> University Sorbonne Paris North, LIPN UMR CNRS 7030, Villetaneuse, France  
[ikram.garfatta@lipn.univ-paris13.fr](mailto:ikram.garfatta@lipn.univ-paris13.fr)

<sup>3</sup> University of Monastir, Higher Institute for Computer Science and Mathematics, Monastir, Tunisia

<sup>4</sup> Institut Mines-Télécom, Télécom SudParis, SAMOVAR UMR 5157, Évry, France

**Abstract.** Several features of the Blockchain technology are well aligned with critical issues in the Business Process Management (BPM) field, and yet adopting Blockchain for BPM should not be taken lightly. In fact, the security of smart contracts, which are one of the main elements of the Blockchain that make the integration with BPM possible, has proved to be vulnerable. It is therefore crucial for the protection of the designed business processes to prove the correctness of the smart contracts to be deployed on a blockchain. In this paper we propose a formal approach based on the transformation of Solidity smart contracts, with consideration of the BPM context in which they are used, into a Hierarchical Coloured Petri net. We express a set of smart contract vulnerabilities as temporal logic formulae and use the *Helena* model checker to, not only detect such vulnerabilities while discerning their exploitability, but also check other temporal-based contract-specific properties.

**Keywords:** Blockchain · Business process management · Model checking · Solidity · Smart contracts · Hierarchical coloured petri nets · Temporal properties

## 1 Introduction

Initially featured as the technology behind Bitcoin, Blockchain has soon after escaped the box of cryptocurrencies to find its way into a multitude of application domains, including that of Business Process Management (BPM). In fact, its inherent characteristics, namely its decentralized nature, ability to provide trust among trustless parties, immutability and financial transparency seem to deliver the right tools to contrive adequate solutions for existing problems in BPM, especially for collaborations [20]. One of the promising integration possibilities of these two fields is the design of Blockchain-based business processes (BPs). The general preference has been to use an existing modeling language for BPs and adopt Blockchain for different aspects of their management. For instance, Lorikeet [28] is

a tool that leverages Blockchain as a message exchange mechanism for BP choreographies. Caterpillar [16], on the other hand, is used to implement the BP model and deploy it on the chain. This has been possible thanks to the concept of *smart contracts* which allow the execution of sequences of interdependent transactions while complying to the rules implemented within. In general, a BP can be analogously viewed as a sequence of tasks linked by causal relationships with the aim of achieving a business goal. Therefore, smart contracts seem to be ideal candidates for the implementation and automation of BPs.

Despite the advance in the adoption of Blockchain for the BPM context, its state is still nascent, and using smart contracts to carry on BPs cannot be considered safe. Many attacks with significant consequences on several blockchains, exploiting hidden vulnerabilities in smart contracts and exposing the defectiveness of the targeted applications bear witness to such a risk. In 2010, 92 billion BTC were generated out of thin air by exploiting an integer vulnerability on the Bitcoin blockchain [1]. The DAO attack on Ethereum exploited a reentrancy vulnerability and resulted in 3.6M of stolen Ether [25]. A vulnerable blockchain-based application does not have to be the target of an attack to malfunction. For instance, the Parity multisig wallet was subject to an accident caused by a self-destruction vulnerability in 2017 and resulting in freezing 500K of Ether [26].

Informal as well as formal methods have been proposed to ensure the correctness of smart contracts. While informal techniques can test a smart contract under certain scenarios, they cannot be relied on to verify specific properties defining its correctness. We note that we are interested in Ethereum smart contracts as it is currently the second largest cryptocurrency platform after Bitcoin besides being the inaugurator of smart contracts, and more particularly those written in Solidity [2] as it is the most popular language used by Ethereum.

In this paper, we propose a model-checking-based approach for the verification of Solidity smart contracts with a particular focus on those used in the BPM context. Thanks to their ability to combine the analysis power of Petri nets with the expressive power of programming languages, Coloured Petri Nets (CPNs) [11] are suitable candidates for the modeling and verification of large and complex systems, and therefore they are employed in our approach to model the smart contracts execution with respect to a behavior specification defining the workflow within which they are used. The result of this modelling step is a hierarchical CPN (HCPN), on which we define a set of temporal properties to express vulnerabilities as well as contract-specific properties relevant to both data- and control-flows of the modelled smart contracts. We implement a prototype that automates the generation of the HCPN model in the specification language of *Helena* [9], the model checker we use for the verification of the defined properties.

The remainder of this paper will be organized as follows: Sect. 2 provides an overview of related studies on formal verification of Solidity smart contracts. Prerequisites on CPN and a brief overview on the representation of BP models are given in Sect. 3, followed by a use case in Sect. 4. An overview of our proposed approach is given in Sects. 5 followed by its detailed steps in Sect. 6. The formal specification of some vulnerabilities and the application on the use case are presented in Sect. 7. Finally, Sect. 8 concludes the paper.

## 2 Related Work

Existing studies on formal verification of smart contracts follow mainly two streams [10]: The first is based on theorem proving [3, 5]. Approaches based on this technique cannot be fully automated as the user usually has to intervene to assist the prover. The second includes studies based on model checking, which is where our work can be situated. Most of the studies under this second category use symbolic model checking coupled with complementary techniques such as symbolic execution [13] and abstraction [4]. The first attempt was Oyente [17], a tool that targets four vulnerabilities and operates at the EVM bytecode level of the contract. It generates symbolic execution traces and analyzes them to detect the satisfaction of certain conditions on the paths which indicates the presence of corresponding vulnerabilities. Numerous studies followed in the footsteps of this work, some of which exploited some of its components in their implementations like GASPER [6] which reuses Oyente’s generated control flow graph, while others extended it with the aim of supporting the detection of other vulnerabilities, like Osiris [27]. Also based on symbolic model checking, Zeus [12] operates on the source code of the contract. VeriSolid [18] is an FSM-based approach that aims at producing a correct-by-design contract rather than detecting bugs. The authors propose a transformation of a contract modeled as an FSM into a Solidity code and provide the ability to specify intended behavior in the form of liveness, deadlock freedom and safety properties expressed using templates for CTL properties and checked by a backend symbolic model checker. The proposed approaches usually use under-approximation which means that critical violations can be overlooked. This explains the presence of false negatives and/or positives in their reported results. We note that most of the existing studies target specific vulnerabilities in contracts, and few are those that allow expressing customizable control flow-related properties while none target data-related properties.

More recently, other attempts using CPN have been proposed. The work in [15] shows an example of verification of behavioural properties applied manually on a CPN model for a crowdfunding smart contract. It does not, however, propose a complete approach with generic transformation rules that can be automated and applied to any contract. Another CPN-based proposition was presented in [8]. This approach, despite being based on CPN, cannot be used for the verification of data-flow related properties as the generated model focuses on the representation of the workflow extracted from the contract’s CFG.

Our proposed approach aims at overcoming the stated shortcomings by providing the means to elaborate behavioural and contract-specific properties (in the form of temporal properties) that can depend on the data-flow in the contract and hence is not bound to a restricted set of reported vulnerabilities. Besides, our approach relies on explicit model checking and that our transformation algorithm operates on the source code as opposed to the bytecode. Hence, we avoid the consequences of under-approximation and contextual information loss.

### 3 Preliminaries

#### 3.1 On Coloured Petri Nets

A Petri net [22] is a formal model with mathematics-based execution semantics. It is a directed bipartite graph with two types of nodes: places (drawn as circles) and transitions (drawn as rectangles). Despite its efficiency in modelling and analysing systems, a basic Petri net falls short when the system is too complex, especially when representation of data is required. To overcome such limitations, extensions to basic Petri nets were proposed, equipping the tokens with colours or types and hence allowing them to hold values. A large Petri net model can therefore be represented in a much more compact and manageable manner using a *Coloured Petri net* [11]. The formal definition of a CPN is given in Definition 1 and the main concepts needed to define its dynamics are given in Definition 2.

**Definition 1 (Coloured Petri net).** *A Coloured Petri Net is a nine-tuple  $CPN = (P, T, A, \Sigma, V, C, G, E, I)$ , where:*

1.  $P$  is a finite set of places.
2.  $T$  is a finite set of transitions such that  $P \cap T = \emptyset$ .
3.  $A \subseteq (P \times T) \cup (T \times P)$  is a set of directed arcs.
4.  $\Sigma$  is a finite set of non-empty colour sets.
5.  $V$  is a finite set of typed variables such that  $Type[v] \in \Sigma, \forall v \in V$ .
6.  $C : P \rightarrow \Sigma$  is a colour set function that assigns a colour set to each place.
7.  $G : T \rightarrow EXPR_V$ , where  $EXPR_V$  is the set of expressions with variables in  $V$ , is a guard function that assigns a guard to each transition  $t$ .
8.  $E : A \rightarrow EXPR_V$  is an arc expression function that assigns an arc expression to each arc  $a$  such that  $Type[E(a)] = C(p)_{MS}$ .
9.  $I : P \rightarrow EXPR_\emptyset$  is an initialisation function that assigns an initialisation expression to each place  $p$  such that  $Type[I(p)] = C(p)_{MS}$ .

**Definition 2 (CPN concepts).** *For CPN  $(P, T, A, \Sigma, V, C, G, E, I)$ , we note:*

1. A marking is a function  $M$  that maps each place into a multiset of tokens.
2. The initial marking  $M_0$  is defined by  $M_0(p) = I(p)\langle \rangle$  for all  $p \in P$ .
3. The variables of a transition  $t$  are denoted by  $Var(t) \subseteq V$ .
4. A binding of a transition  $t$  is a function  $b$  that maps each variable  $v \in Var(t)$  into a value  $b(v) \in Type[v]$ . It is written as  $\langle var_1 = val_1, \dots, var_n = val_n \rangle$ . The set of all bindings for a transition  $t$  is denoted  $B(t)$ .
5. A binding element is a pair  $(t, b)$  such that  $t \in T$  and  $b \in B(t)$ . The set of all binding elements  $BE(t)$  for a transition  $t$  is defined by  $BE(t) = \{(t, b) | b \in B(t)\}$ . The set of all binding elements in a CPN model is denoted  $BE$ .

A transition is said to be *enabled* if a binding of the variables appearing in the surrounding arc inscriptions exists such that the inscription on each input arc evaluates to a multiset of token colours present on the corresponding input place. *Firing* a transition consists in removing (resp. adding), from each input (resp. to each output) place, the multiset of tokens corresponding to the input (resp. output) arc inscription. For more details on CPN we refer readers to [11].

### 3.2 On Business Process Modeling Representations

When it comes to business process modeling languages, controversy arises as to whether imperative or declarative modeling approaches are better. An empirical investigation [24] states that while imperative languages (e.g., Business Process Model and Notation BPMN [23]) can be considered superior in terms of comprehensibility by end-users, this fact's accuracy can be influenced by the experimental subjects' familiarity with imperative modeling languages. On the other hand, declarative modeling approaches (e.g. Dynamic Condition Response DCR Graphs [21]) are considered less rigid than their counterpart and therefore more suitable for rapidly evolving business processes. In fact, imperative models represent *how* a process is executed by explicitly defining its control flow while declarative models focus on *why* a process is executed in such a way by implicitly defining its control flow as a set of rules. Consequently, making changes to an imperative model is more time-consuming and complex than altering a declarative one, since the former would entail explicitly adding/deleting execution alternatives, which can call into question the correctness of the model, while the latter could be achieved by adding/deleting constraints from the model to discard/add execution alternatives. In our work, we do not support any claims for the supposed superiority of any paradigm over the other.

**Definition 3.** A DCR graph is a tuple  $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\Diamond, l)$  where  $\mathcal{M}(G) =_{def} \mathcal{P}(E) \times \mathcal{P}(E) \times \mathcal{P}(E)$  is the set of all markings:

1.  $E$  is the set of events, ranged over by  $e$ .
2.  $M \in \mathcal{M}(G)$  is the marking of the graph.
3.  $Act$  is the set of actions.
4.  $\rightarrow\bullet, \bullet\rightarrow \subseteq E \times E$  are the condition and response relations, respectively.
5.  $\rightarrow+, \rightarrow\% \subseteq E \times E$  are the dynamic include and exclude relations, respectively, satisfying that  $\forall e \in E. e \rightarrow+ \cap e \rightarrow\% = \emptyset$ .
6.  $\rightarrow\Diamond \subseteq E \times E$  is the milestone relation.
7.  $l : E \rightarrow Act$  is a labelling function mapping every event to an action.

A marking  $M = (Ex, Re, In) \in \mathcal{M}(G)$  is a triplet of event sets where  $Ex$  represents the set of events that have previously been executed,  $Re$  the set of events that are pending responses required to be executed or excluded, and  $In$  the set of events that are currently included. The idea conveyed by the dynamic inclusion/exclusion relations is that only the currently included events are considered in evaluating the constraints. In other words, if  $e$  is a condition for  $e'$  ( $e \rightarrow\bullet e'$ ), but is excluded from the graph then it no longer restricts the execution of  $e'$ . Moreover, if  $e'$  is the response for  $e$  ( $e \bullet\rightarrow e'$ ) but is excluded from the graph, then it is no longer required to happen for the flow to be acceptable. The inclusion relation  $e \rightarrow+ e'$  (resp. exclusion relation  $e \rightarrow\% e'$ ) means that, whenever  $e$  is executed,  $e'$  becomes included in (resp. excluded from) the graph. The milestone relation is similar to the condition relation in that it is a blocking one. The difference is that it is based on the events in the pending response set. In other words, if  $e'$  is a milestone of  $e$  ( $e' \rightarrow\Diamond e$ ), then  $e$  cannot be executed as long as  $e'$  is in  $Re$ . For more details on DCR Graphs we refer the readers to [21].

## 4 Use Case: Blind Auction

Our use case is adapted from [2]. Participants in a blind auction have a bidding window during which they can place their bids. A participant can place more than one bid and the placed bid is blinded. The bidder has to make a deposit along the bid with a value that is supposedly greater than the real bid. Once the bidding window is closed, the revealing window is opened. Participants proceed to reveal their bids by sending the actual values of the bids along with the used keys. The system verifies whether the sent values correspond with the placed blinded bids and potentially updates the highest bid and bidder's values. If the revealed value of a bid does not correspond with its blinded value, or is greater than the deposit, the said bid is considered invalid. Once the revealing window is closed, participants can proceed to withdraw their deposits. A deposit made along a non-winning, invalid or unrevealed bid is wholly restored. In case of a winning bid, the difference between the deposit and the real bid is restored. The auction is terminated when all participants withdraw their deposits. We propose a design for such a blind auction system using a BPMN choreography diagram as well as a DCR graph (Fig. 1). Listing 1.1 is an excerpt of the corresponding Solidity contract. The full Solidity example can be found in our repository<sup>1</sup>.

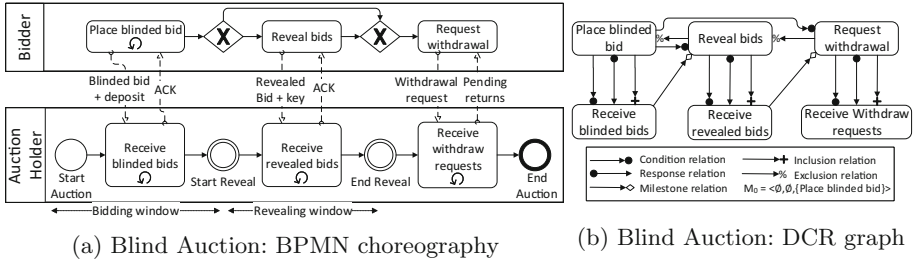


Fig. 1. Blind auction workflow representations

```
contract BlindAuction {
    struct Bid {bytes32 blindedBid; uint deposit;}
    uint public biddingEnd, revealEnd, highestBid;
    mapping(address => Bid[]) public bids;
    address public highestBidder;
    mapping(address => uint) pendingReturns;
    modifier onlyBefore(uint _time) {require(now<_time);;}
    modifier onlyAfter(uint _time) {require(now>_time);;}
    constructor(uint _biddingTime, uint _revealTime) public {...}
    function bid(bytes32 _blindedBid) public payable onlyBefore(
        biddingEnd) {...}
    function reveal(uint[] values, bytes32[] secrets) public
        onlyAfter(biddingEnd) onlyBefore(revealEnd) {...}
    function withdraw() public onlyAfter (revealEnd) {
        uint8 amount = pendingReturns[msg.sender];
        if (amount > 0) {
```

<sup>1</sup> <https://depot.lipn.univ-paris13.fr/garfatta/sol2cpn>.

```

if (msg.sender != highestBidder)
    msg.sender.call.value(amount)("");
else
    msg.sender.call.value(amount-highestBid) ("");
pendingReturns [msg.sender] = 0;}}}

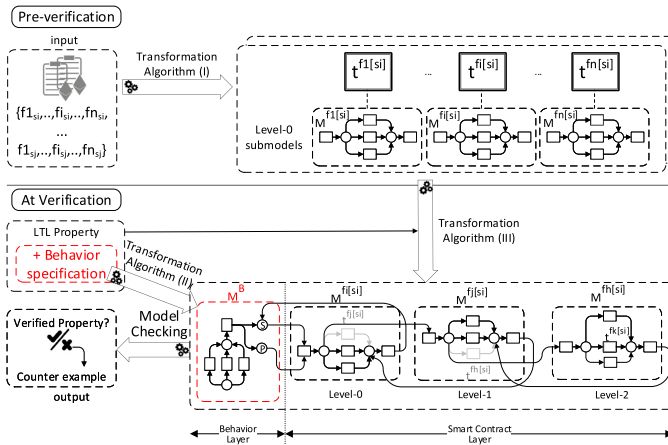
```

**Listing 1.1.** Excerpt of the Blind Auction smart contract in Solidity

## 5 Overview of our Formal Verification Approach

Our proposed approach for the verification of smart contracts is based on model checking of CPN models and comprises mainly two phases:

1. A pre-verification phase: consists in transforming the smart contracts' Solidity code into CPN submodels corresponding to their functions.
2. A verification phase: consists in constructing a CPN model w.r.t an LTL property that can express: (i) a vulnerability in the code or (ii) a contract-specific property, linking it to a CPN model representing the behavior to be considered, and feeding it the model checker to verify the targeted property.



**Fig. 2.** Overview of the approach

More precisely, we opt for a hierarchical CPN to represent the considered smart contracts' execution and interaction w.r.t the provided behavior specification.

As shown in Fig. 2, we represent each function of a smart contract by an *aggregated transition* that encapsulates a submodel corresponding to the internal workflow of the former. In fact, our aim at this pre-verification phase is to get building blocks for the hierarchical model that will be fed to the model checker. Then, given a behavior specification and an LTL property to be verified, the final CPN model is built by (1) linking the aggregated transition representing

the targeted function to the behavioral model and (2) building a hierarchy by explicitly representing function calls in the submodel in question (if the checked property requires it). In fact, function calls are initially abstracted and therefore represented by aggregated transitions in the model (e.g.,  $t^{fj[si]}$  in Fig. 2) under the assumption that they do not present behavioral problems (deadlock-free and strong-livelock-free) which can be separately verified for each function. Depending on the property to be verified, an aggregated transition may need to be *unfolded* if any of its corresponding function's instructions or variables are involved in the property, hence the multi-level hierarchy in the model (e.g., in Fig. 2,  $t^{fj[si]}$  in  $M^{fi[si]}$  is hidden and replaced by its submodel  $M^{fj[si]}$ ). It is kept *folded* otherwise (e.g.,  $t^{fk[si]}$  in  $M^{fh[si]}$ ). This abstraction leads to a reduction in the size of the state space the model checker needs to explore.

## 6 Generation of the Hierarchical CPN Model

In order to implement our approach, we propose a transformation algorithm for the generation of the final HCPN model from the provided input artifacts.

### 6.1 Our HCPN Model: Defining Its Elements

**Transitions  $T$ .** We distinguish two types of transitions in our model:

1. aggregated ( $T^A$ ): used at the level-0 model for the representation of functions, as well as at higher levels for the modular representation of function calls and can be substituted by a submodel.
2. regular ( $T^R$ ): simple unsubstitutable CPN transition.

For a transition  $t \in T$  we note:

- $t.st$ , the Solidity code associated to transition  $t$
- $t.metaColour$ , the metaColour associated to the control flow places of  $t$  (if  $t \in T^A$ )
- $t.data$ , the set of data places associated to transition  $t$  (if  $t \in T^A$ )
- $t.submodel$ , the CPN submodel associated to transition  $t$  (if  $t \in T^A$ ), with  $t.submodel.inTransitions$  (resp.  $t.submodel.outTransitions$ ) designating its input (resp. output) transitions
- $t.guard$ , the guard of the transition  $t$
- $\bullet t[cf], t \bullet [cf] \in P_{CF} \cup P_S$ , the input and output control flow places of  $t$
- $\bullet t[input] \in P_P$ , the input parameters place of  $t$
- $\bullet t[data], t \bullet [data] \subseteq P_{data}$ , the input and output data places of  $t$
- $t \bullet [output] \in P_R$ , the output return place of  $t$

**Places  $P$ .** For level-1 submodels, we define 4 types of places:



- *Control flow places*  $P_{CF}$  are places created to implement the order of execution of the workflow. We also use them to carry data related to the state of the smart contract which can be defined by its balance and the values of its state variables. Such places have a *metaColour* defined at each aggregated transition  $t^a$  of level-0 as the concatenation of the *state* (i.e., the colour of  $\bullet t[cf] \in P_S$ ) and the *input parameters* (i.e., the colour of  $\bullet t[input] \in P_P$ ):  $[uint: contractBalance, type_{v_1}: stateVariable_1, \dots, type_{v_n}: stateVariable_n, type_{p_1}: inputParameter_1, \dots, type_{p_n}: inputParameter_n]$ .
- *Data places*  $P_{data}$  (for internal local variables) where each place is of a colour corresponding to the represented variable's type.
- *Parameter places*  $P_P$  that convey potential inputs of function calls. Each function call has an associated parameter place whose colour is as follows  $[type_{p_1}: inputParameter_1, \dots, type_{p_n}: inputParameter_n]$ .
- *Return places*  $P_R$  that communicate potential functions' returned data and whose colours correspond to the return type of the called functions.

Two input places are created at the behavioral layer:

- a *state place*  $p_s \in P_S$  representing the state of the contract. Its colour is as follows:  $[uint: contractBalance, type_{v_1}: stateVariable_1, \dots, type_{v_n}: stateVariable_n]$
- a *parameters place*  $p_p \in P_P$  representing the input parameters of the function in question.

**Expressions  $E$ .** Expression are constructs made up of literals, variables, function calls and operators, according to the syntax of Solidity, that evaluate to single values:

- expressions with variables  $E_V$ : they make use of at least one local variable. In such an expression  $e_v$ , the set of variables used is accessible via  $e_v.vars$ .
- expressions with function calls  $E_F$ : they make use of at least one function call. In such an expression  $e_v$ , the set of function calls used is accessible via  $e_v.fctCalls$
- explicit expressions  $E_E$ : they do not make use of variables nor function calls.

**Statements  $S$ .** A statement  $st \in \mathbb{S}$  can be either a compound statement  $\{st[1]; st[2]; \dots; st[N]\}$  (where  $\forall i \in [1..N], st[i] \in S$ ), or a simple statement  $(st_{LHS}, st_{RHS})$  (where  $st_{LHS} \in E$  and  $st_{RHS} \in E$ ), or a control statement. A simple statement can be:

- a function call statement, where:
  - $st_{LHS} = \emptyset$
  - $st_{RHS}.vars$  is the set of variables used in the arguments of the call (if  $st_{RHS} \in E_V$ )
- an assignment statement, where:
  - $st_{LHS} \in E_V$  and  $st_{LHS}.vars$  contains the assigned variable
  - $st_{RHS}.vars$  is the set of variables used in the assignment (if  $st_{RHS} \in E_V$ )
  - $st_{RHS}.fctCalls$  is the set of function calls used in the assignment (if  $st_{RHS} \in E_F$ )

- a variable declaration statement, where:
  - $st_{LHS} \in E_V$  and  $st_{LHS}.vars$  contains the declared variable
  - $st_{LHS}.type$  designates the type of the declared variable
  - $st_{RHS}.vars$  designates the set of variables used in the variable initialization expression (if the variable is initialized and  $st_{RHS} \in E_V$ )
  - $st_{RHS}.fctCalls$  is the set of function calls used in the variable initialization (if the variable is initialized  $st_{RHS} \in E_F$ )
- a sending statement, where:
  - $st_{LHS}$  designates the destination account
  - $st_{RHS}.vars$  designates the set of variables in the expression of the value to be sent (if  $st_{RHS} \in E_V$ )
  - $st_{RHS}.fctCalls$  is the set of function calls in the value to be sent (if  $st_{RHS} \in E_F$ )
- a returning statement, where:
  - $st_{LHS} = \emptyset$
  - $st_{RHS}.vars$  is the set of variables in the returned value (if  $st_{RHS} \in E_V$ )
  - $st_{RHS}.fctCalls$  is the set of function calls in the returned value (if  $st_{RHS} \in E_F$ )

A control statement can be:

- a requirement statement of the form  $require(c)$
- a selection statement of the form  $if(c) \text{ then } st_T \text{ [else } st_F]$
- a looping statement which can be:
  - a for loop:  $for(init; c; inc) \text{ } st_T$
  - a while loop:  $while(c) \text{ } st_T$
- where:
  - $c$  is a boolean expression
  - $c.vars$  designates the set of variables used in the condition (if  $c \in E_V$ )
  - $c.fctCalls$  is the set of function calls used in the condition (if  $c \in E_F$ )
  - $st_T, st_F, init$  and  $inc$  are statements

## 6.2 Solidity-to-CPN: Building Blocks for the Smart Contract Layer

The first step is to build the level-0 submodels for the aggregated transitions of the contract's functions. To do so, we propose the algorithm GENERATELEVEL0.

We define a CPN pattern for each Solidity statement type. Considering that a function is a set of statements, CPN snippets are generated according to the defined patterns and linked according to the function's internal workflow. CREATESUBMODEL implements such correspondences. For lack of space, we only include the transformation algorithm for a function call statement and the graphical pattern (Fig. 3) and description of a compound statement. The rest of the algorithms and descriptions are available online (See footnote 1).

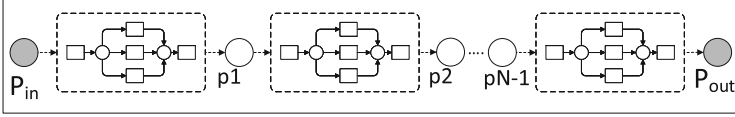


Fig. 3. Compound statement pattern

**Compound statement**  $\{st[1]; st[2]; \dots; st[N]\}$ . The algorithm is re-executed on each component statement  $st[i]$ , after creating  $N-1$  control flow places (of the *metaColour* colour) to interconnect the resulting CPN snippets while merging the entering point of the snippet of  $st[1]$  with the entering point of the snippet of  $st$  and the exiting point of  $st[N]$  to that of the snippet of  $st$ .

**Input** : an aggregated transition  $t^a$   
**Output**: level-0 CPN submodel of  $t^a$

- 1  $P_{data} \leftarrow \emptyset$
- 2 GETLOCALVARIABLES( $t^a.st; P_{data}$ )
- 3  $t^a \leftarrow P_{data}$
- 4  $t^a.sub \leftarrow \text{CREATESUBMODEL}(t^a, \emptyset, \emptyset)$

(a) GENERATELEVEL0

**Input** : transition  $t$ , a function call  
statement  $st = (st_{LHS}, st_{RHS})$ ,  
control flow input place  $p_{in}$ ,  
control flow output place  $p_{out}$

**Output**: submodel for statement  $st$

- 1 create transition  $t^f$
- 2 create place  $p_{param_f}$
- 3 create arc from  $p_{in}$  to  $t^f$
- 4 create arc from  $p_{param_f}$  to  $t^f$
- 5 CONNECTLOCALVARS( $f_{RHS}.vars; t; t^f$ )
- 6 CONNECTFUNCALLS( $f_{RHS}.fctCalls; t$ )
- 7 create arc from  $t^f$  to  $p_{out}$  with a placeholder inscription

(b) BUILDFUNCTIONCALLST

**Input** :  $t$ , statement  $st$ , cf input place  $p_{in}$ , cf output place  $p_{out}$   
**Output**: submodel of transition  $t$

- 1 **switch**  $st$  **do**
- 2   **case** compound statement  $\{st[1];$   
 $st[2]; \dots; st[N]\}$  **do**
- 3     BUILDCOMPOUNDSTATEMENT  
 $(t; st; p_{in}; p_{out})$
- 4   **end case**
- 5   **case** simple statement **do**
- 6     **switch**  $st$  **do**
- 7       **case** ... **do**
- 8         ...
- 9       **end case**
- 10      ...
- 11      **case** function call  
statement **do**
- 12        BUILDFUNCTIONCALLST  
 $(t; st; p_{in}; p_{out})$
- 13      **end case**
- 14    **end switch**
- 15   **end case**
- 16 **end switch**

(c) CREATESUBMODEL

The hierarchy of the CPN model depends on the LTL property to be verified. Such a hierarchy is achieved by *unfolding* targeted aggregated transitions as well as potential aggregated transitions within their submodels<sup>2</sup>.

<sup>2</sup> We note that if a place does not exist ( $p = \emptyset$ ) any arc creation involving it does not take effect.

<b>Input :</b> aggregated transition $t^a, p_{in}, p_{out}$		7 <b>end for</b>
<b>Output:</b> submodel replacement of $t^a$		8 <b>for</b> $t' \in t^a.sub.outTransition$ <b>do</b>
1 <b>for</b> $t' \in t^a.sub.inTransition$ <b>do</b>		9     replicate (arc from $t^a$ to $p_{out}$ ) to
2     replicate (arc from $p_{in}$ to $t^a$ ) to $t'$		$t'$ with the placeholder inscription
3     replicate (arc from $\bullet_t[input]$ to $t^a$ ) to $t'$		replaced by values from
4 <b>for</b> $p \in \bullet^{t^a}[data] \cup \bullet^{t^a}[output]$ <b>do</b>		$\bullet_{t'}[cf]$
5         replicate (arc from $p$ to $t^a$ ) to $t'$		10 <b>end for</b>
6 <b>end for</b>		11 hide transition $t^a$ and all arcs linked to it

(d) UNFOLDTRANSITION

### 6.3 Behavior-to-CPN: Generation of the Behavioral Layer

We consider two types of behavior specifications for smart contracts:

(1) *completely-free* if no information is provided on the execution context of a contract and (2) *constrained* if the context in which a smart contract is used is provided (e.g., as a DCR Graph or a BPMN model). A CPN behavioral model is added as an additional layer and linked to the hierarchical model built using the previously generated CPN submodels.

**Modeling a Completely-Free Behavior.** In case no behavior is provided with the smart contracts to be verified, we define a behavioral model to represent their execution in a completely-free way. In such a model (see Fig. 4a) a place  $S$  is used to represent the global state of the blockchain environment shared by all of the smart contracts' functions. For each function  $f_i$  a place  $P_i$  is used to represent its input parameters. The marking of a place  $P_i$  corresponds to all the possible calling arguments for  $f_i$ .

**Modeling a Constrained Behavior.** The user may want to define the behavior of smart contracts. This can be captured either imperatively or declaratively. Existing BPMN-to-CPN transformations [19] could be leveraged for an imperative representation. For an example of a declarative one, we propose in the following a formal translation of DCR to CPN.

**Definition 4 (CPN4DCR).** *Given a DCR graph  $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$ , a corresponding CPN model  $CPN = (P, T, A, \Sigma, V, C, G, E, I)$  is defined s.t.:*

- $P = \{S\}$
- $T = \{t_i, \forall i \in [1, n]\}$ , with  $n = |E|$  the number of events in  $G$
- $A = \{(t_i, S), \forall i \in T\} \cup \{(S, t_i), \forall i \in T\}$
- $\Sigma = \{C_E, (C_E \times C_E \times C_E)\}$ , where  $C_E$  is a colour defined as an integer type ( $C_E = range\ INT$ ) where each event  $e_i \in E$  is represented in  $C_E$  by its index.
- $V = \{Ex, Re, In, Ex', Re', In'\}$ , with  $Type[v] = C_E, \forall v \in V$
- $C = \{S \rightarrow (C_E \times C_E \times C_E)\}$
- $G = \{t_i \rightarrow guard_i, \forall i \in [1, n]\}$ , with  $n = |E|$

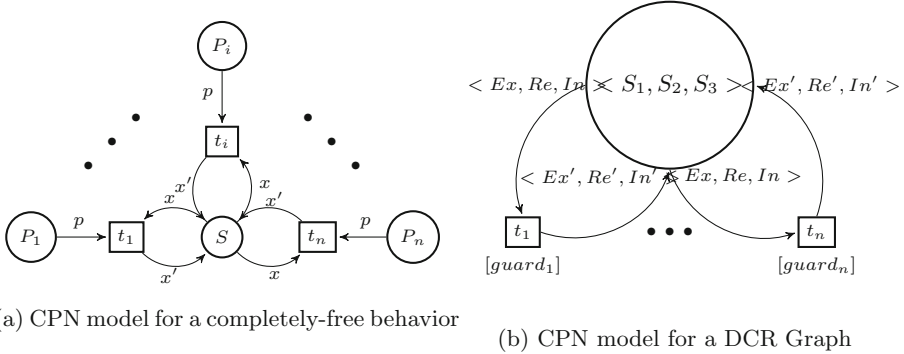
- $E = \{a \rightarrow \langle Ex, Re, In \rangle, \forall a \in A \cap (P \cup T)\} \cup \{a \rightarrow \langle Ex', Re', In' \rangle, \forall a \in A \cap (T \cup P)\}$  with (1)  $Ex' = Ex \cup e_i$ , (2)  $Re' = (Re \setminus e_i) \cup e \bullet \rightarrow$  and (3)  $In' = (In \cup e_i \rightarrow +) \setminus e \rightarrow \%$
- $I = \{S \rightarrow \langle S_1, S_2, S_3 \rangle\}$  with  $\langle S_1, S_2, S_3 \rangle$  the initial marking  $M$  of  $G$

For all  $t_i \in T$  representing an event  $e_i$  in the DCR graph, we further precise that:

- $guard_i$  is the conjunction of the conditions defining the enabling of the corresponding event (1)  $e_i: i \in In$ , (2)  $(\rightarrow \bullet i \cap In) \in Ex$  and (3)  $(\rightarrow \diamond i \cap In) \in E \setminus Re$
- the expression  $\langle Ex', Re', In' \rangle$  on its output arc is defined such that: (1)  $Ex' = Ex \cup i$ , (2)  $Re' = (Re \setminus i) \cup i \bullet \rightarrow$  and (3)  $In' = (In \cup i \rightarrow +) \setminus i \rightarrow \%$

**Theorem 1.** Let  $G$  be a DCR graph and  $C$  the corresponding CPN model generated by following definition 4, then  $G$  and  $C$  are semantically equivalent.

We include a proof of this theorem in our repository (See footnote 1).



**Fig. 4.** Behavior representations

## 7 Model Checking: On the Blind Auction Use Case

Given the HCPN model generated by the application of our transformation algorithm on the input smart contracts along with the LTL property to check and the behavior specification, we use *Helena* [9] to verify the validity of the considered LTL property on our model. Such a property can express either a predefined vulnerability, or a contract-specific property. In fact, many vulnerabilities have been identified in the literature [7], and the user may want to check the presence of certain bugs in a smart contract. To prove the ability of our approach to detect vulnerabilities, we propose LTL formulae to express common vulnerabilities. We then apply our approach on our use case and showcase its capability to detect vulnerabilities as well as check contract-specific properties.

## 7.1 Expressing Vulnerabilities in LTL

We consider here one of the most common vulnerabilities in Solidity smart contracts. More vulnerabilities are explained and expressed in LTL in our repository (See footnote 1). In the following,  $t_{s_i}^f$  denotes the CPN aggregated transition for function  $f$  in smart contract  $s_i$ .

**Integer Overflow/Underflow:** Due to Solidity’s lack of safeguards on mathematical operators, errors such as overflows and underflows may occur as a result of violation of value limitations of integer data types. For instance, the *uint8 amount* variable in the *BlindAuction* contract can be the source of such a vulnerability when the *pendingReturns* of a bidder exceeds 255. Due to Solidity’s wrapping in two’s complement integer representation, *amount* will contain a wrong value, causing an incorrect execution.

In our CPN model, we define correspondences between the types used in the Solidity language and those offered by *helena* so that they cover the same ranges. The model checker is therefore able to detect when the smart contract contains an out-of-range expression. It does not, however, pinpoint the source of the anomaly, so the user does not have much information to go on to track it and try to correct it. To overcome this deficiency, we propose to model integer overflows/underflows as a safety LTL property that can be verified on a specific variable  $x$  to check:

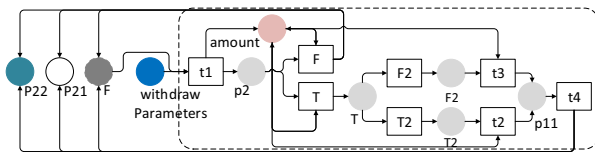
$$IUO_x = \Box \neg x IsOutOfRange$$

where *xIsOutOfRange* is a proposition defining the conditions for overflow and underflow for *x* w.r.t the range of its type which we delimit by defining lower and higher thresholds:

$$xIsOutOfRange = (x < minThreshold) \vee (x > maxThreshold)$$

## 7.2 Application on the Use Case

The application of our approach on the use case (Sect. 4) yields a HCPN model whose level-0 submodels are created by the execution of `CREATESUBMODEL`. For lack of space, we choose to include the submodel for *withdraw* in Fig. 5.



**Fig. 5.** SubModel of transition *withdraw*

Verifying properties of the contract would come down to verifying properties on the corresponding CPN model. For model checking, we chose *Helena* [9] which

offers explicit model checking support for on-the-fly verification of state and LTL properties over CPN models. We have generated the CPN models of our use case in *Helena*'s specification language using our prototype for the transformation algorithm, while considering a free behaviour as well as the BPMN and DCR specifications as presented in Sect. 4. We have then written the corresponding properties in *Helena*'s language for the vulnerabilities in Sect. 7.1 and were able to detect them. We have also established other contract-specific properties that we were able to verify on our example. Figure 6 shows the corresponding property written in *Helena* for the *IUO* LTL property applied on the variable *amount* in *BlindAuction* and Fig. 7 is a snippet of the result of the model checker showing the detection of the vulnerability with a counter example.

```

440 proposition outOfRange : exists (t in S|(t->1).amount > maxThreshold
                                or (t->1).amount < minThreshold);

1 ltl property IUO;
2 [] not outOfRange;

```

**Fig. 6.** The integer overflow/underflow LTL property in *Helena*

```

Search report
-----
Action performed
  property checking
Host machine
  Ikranz (pid = 60511)
Property checked
  IUO
Termination state
  PROPERTY_VIOLATED
Statistics report
-----
Model statistics
-----
  24 places
  28 transitions
  72 arcs
Trace report
-----
The following run invalidates the property.
{
  s = <{ 0, 0, 0, ||, false, 0 } >

```

**Fig. 7.** Model checking result

The artifacts used in this verification as well as a detailed report on the results and the prototype implementation can be found at this repository (See footnote 1).

## 8 Conclusion

The combination of the Blockchain technology and the BPMN domain has been an evident step, especially considering the assets that the former brings to the latter. It is still crucial, however, to guarantee the correctness of the smart contracts involved in this association to ensure its safety. Existing verification approaches are generally designed to target specific vulnerabilities which have been reported

to be the root of some attacks or malfunctions. Checking the absence of vulnerabilities in a smart contract, however necessary, does not guarantee its correctness as a faulty behaviour may stem from a flaw specific to that contract. With our approach we aim to bring a solution to this problem by providing a way to formally verify contracts by both checking for vulnerabilities in the code and offering the possibility to express additional contract-specific properties to check. In this paper, we focus on extending our approach to take into account the context in which the smart contracts to be verified are executed as a behavior specification, while also considering the case where no such specification is provided. To further improve the *Helena's* performance, we intend to work on *Helena's* model checker by embedding it with an extension to an existing technique previously developed to deal with the state space explosion problem in regular PN's [14] and applying it on CPNs.

## References

1. Overflow incident. [en.bitcoin.it/wiki/Value/overflow/incident](https://en.bitcoin.it/wiki/Value/overflow/incident)
2. Solidity documentation. [docs.soliditylang.org/en/latest/](https://docs.soliditylang.org/en/latest/)
3. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in isabelle/hol. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 66–77. NY, USA (2018)
4. Anand, S., Pasareanu, C.S., Visser, W.: Symbolic execution with abstraction. Int. J. Softw. Tools Technol. Transf. **11**(1), 53–67 (2009)
5. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Austria (2016)
6. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Austria, pp. 442–446 (2017)
7. Dingman, W., et al.: Defects and vulnerabilities in smart contracts, a classification using the NIST bugs framework. IJNDC **7**(3), 121–132 (2019)
8. Duo, W., Huang, X., Ma, X.: Formal analysis of smart contract based on colored petri nets. IEEE Intell. Syst. **35**(3), 19–30 (2020)
9. Evangelista, S.: High level petri nets analysis with helena. In: Applications and Theory of Petri Nets 2005, pp. 455–464. Berlin, Heidelberg (2005)
10. Garfatta, I., Klai, K., Gaaloul, W., Graiet, M.: A survey on formal verification for solidity smart contracts. In: ACSW '21: 2021 Australasian Computer Science Week Multiconference, New Zealand, 2021, pp. 1–10. ACM (2021)
11. Jensen, K., Kristensen, L.M.: Coloured petri nets: modelling and validation of concurrent systems, 1st (edn.) Springer Publishing Company, Incorporated (2009)
12. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, 2018 (2018)
13. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Poland, Proceedings (2003)



14. Klai, K., Poitrenaud, D.: MC-SOG: an LTL model checker based on symbolic observation graphs. In: Applications and Theory of Petri Nets, 29th International Conference, PETRI NETS 2008, Xi'an, China, 2008. Proceedings, pp. 288–306 (2008)
15. Liu, Z., Liu, J.: Formal verification of blockchain smart contract based on colored petri net models. In: 43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, USA, vol. 2, pp. 555–560. IEEE (2019)
16. López-Pintado, O., García-Bañuelos, L., Dumas, M., Weber, I., Ponomarev, A.: Caterpillar: A business process execution engine on the ethereum blockchain. *Softw. Pract. Exp.* **49**(7), 1162–1193 (2019)
17. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Austria, 2016, pp. 254–269 (2016)
18. Mavridou, A., Laszka, A., Stachtieri, E., Dubey, A.: Verisolid: correct-by-design smart contracts for ethereum. In: Financial Cryptography and Data Security - 23rd International Conference, FC 2019, St. Kitts and Nevis, 2019, pp. 446–465 (2019)
19. Meghzili, S., Chaoui, A., Strecker, M., Kerkouche, E.: An approach for the transformation and verification of BPMN models to colored petri nets models. *Int. J. Softw. Innov.* **8**(1), 17–49 (2020)
20. Mendling, J., et al.: Blockchains for business process management - challenges and opportunities. *ACM Trans. Manag. Inf. Syst.* **9**(1), 1–16 (2018)
21. Mukkamala, R.R.: A formal model for declarative workflows dynamic condition response graphs. (2012)
22. Murata, T.: Petri nets: properties, analysis and applications. *Proc. IEEE* **77**(4), 541–580 (1989)
23. OMG: Business process model and notation (bpmn) 2.0. (2011). [www.omg.org/spec/BPMN/2.0/](http://www.omg.org/spec/BPMN/2.0/)
24. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus declarative process modeling languages: an empirical investigation. In: Business Process Management Workshops - BPM 2011 International Workshops, pp. 383–394. Clermont-Ferrand, France, 2011 (2011)
25. Siegel, D., et al.: The dao attack: understanding what happened (2020). [www.coindesk.com/understanding-dao-hack-journalists](http://www.coindesk.com/understanding-dao-hack-journalists)
26. Team, S.: Parity multi-sig wallets funds frozen (explained) (2021). [www.springworks.in/blog/parity-multi-sig-wallets-funds-frozen-explained/](http://www.springworks.in/blog/parity-multi-sig-wallets-funds-frozen-explained/)
27. Torres, C.F., Schütte, J., State, R.: Osiris: hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 664–676. ACSAC 2018, PR, USA (2018)
28. Tran, A.B., Lu, Q., Weber, I.: Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In: Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018, vol. 2196, pp. 56–60. Sydney, Australia (2018)