Model Checking of Solidity Smart Contracts Adopted For Business Processes Extended Content

This document contains complementary content for the paper “Model Checking of Solidity Smart Contracts Adopted For Business Processes" submitted to ICSOC 2021 (The 19th International Conference on Service Oriented Computing).

Abstract. Several key 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 as the former does not only bring benefits to the latter but also comes with its own set of challenges. 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 (BPs) 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. 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 the BPM research field, especially when it comes to collaborations [19]. One of the promising integration possibilities of these two fields is the design of Blockchain-based business processes. 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 [25] is a tool that leverages Blockchain as a message exchange mechanism for BP choreographies. Caterpillar [15], 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 blockchain platforms, exploiting hidden vulnerabilities in deployed 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. The DAO attack on Ethereum exploited a reentrancy vulnerability and resulted in 3.6M of stolen Ether. 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-destruct vulnerability in 2017 and resulting in freezing 500K of Ether.

Informal as well as formal methods have been proposed to enhance the security of smart contracts and ensure their correctness. While informal techniques can test a smart contract under certain scenarios, they cannot be relied on to verify specific properties defining its correctness (e.g., absence of integer overflow vulnerabilities, deadlock-freedom) which is where formal techniques prove to be efficient. We note that in our work, 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 [1] as it is the most popular language used by Ethereum. We also note that while Ethereum allows smart contracts to be written in a 'Turing complete' language that facilitates semantically richer applications than Bitcoin which allows very simple forms of smart contracts, the former also enlarges the threat surface, as evidenced by the many high-profile attacks.

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) [10] 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, 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* [8], the model checker we use for the verification of the defined properties.

The remainder of this paper will be organized as follows: Section 2 provides an overview of related studies on formal verification of Solidity smart contracts. Prerequisites on such contracts as well as our chosen modeling formalism CPN and a brief overview on the representation of BP models are given in Section 3. A use case is presented in Section 4 before an informal description of our approach is given in Section 5. The different steps of our approach are detailed in Section 6 while the formal specification of some smart contracts' vulnerabilities and the application of the approach on the use case is presented in Section 7. Finally, the paper is concluded by Sections 8.

2 Related Work

Existing studies on formal verification of smart contracts follow two main streams [9]: The first is based on theorem proving [4, 2]. 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 [12] and abstraction [3]. The first attempt was Oyente [16], a tool that targets 4 vulnerabilities, namely transaction order dependence, timestamp dependence, mishandled exceptions and reentrancy. It operates at the EVM bytecode level of the contract, generating symbolic execution traces and analyzing 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 [5] which reuses Oyente's generated control flow graph for the detection of bytecode patterns with high gas costs, while others extended it with the aim of supporting the detection of other vulnerabilities, like Osiris [24].
Also based on symbolic model checking, Zeus [11] operates on the source code of the contract. VeriSolid [17] 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 (e.g., in the form of loop bounds) which means that critical violations can be overlooked. This explains the presence of false negatives and/or positives in their reported results. We also note that most of the existing studies target specific vulnerabilities in smart contracts, and few are those that allow expressing customizable control flow-related properties. In fact, none of these studies target data-related properties.

More recently, attempts have been made to use CPN for the verification of smart contracts. The work in [14] shows an example of verification of behavioural properties applied manually on a CPN model for a case study of a crowdfunding smart contract. It does not, however, propose a complete approach with generic transformation rules that can be automated and applied to any smart contract. Another CPN-based proposition was presented in [7]. 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 to be checked and hence is not bound to a restricted set of reported vulnerabilities. Besides, we note that our approach relies on an explicit model checking technique 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 [21] 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*.

A CPN [10] combines the capabilities of Petri nets, from which its graphical notation is derived, with those of CPN ML, a functional programming language based on Standard ML, to define data types. 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. Σ *is a finite set of non-empty* colour sets.
5. V *is a finite set of* typed variables *such that* $Type[v] \in \Sigma$ *for all variables* $v \in V$.
6. $C : P \to \Sigma$ *is a* colour set function *that assigns a colour set to each place.*
7. $G : T \to EXPR_V$*, where* $EXPR_V$ *is the set of expressions provided by CPN ML with variables in* V*, is a* guard function *that assigns a guard to each transition* t.
8. $E : A \to EXPR_V$ *is an* arc expression function *that assigns an arc expression to each arc* a *such that* $Type[E(a)] = C(p)_{MS}$ *(i.e., the type of the arc expression is a multiset type over the colour set of the place* p *connected to the arc* a*).*
9. $I : P \to EXPR_\emptyset$ *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. $\bullet p$ *and* $p\bullet$ *respectively denote the sets of input and output transitions of a place* p.
2. $\bullet t$ *and* $t\bullet$ *respectively denote the sets of input and output places of a transition* t.
3. *A* marking *is a function* M *that maps each place* $p \in P$ *into a multiset of tokens* $M(p) \in C(p)_{MS}$.
4. *The* initial marking M_0 *is defined by* $M_0(p) = I(p)\langle\rangle$ *for all* $p \in P$.
5. *The* variables of a transition t *are denoted by* $Var(t) \subseteq V$ *and consist of the free variables appearing in its guard and in the arc expressions of its connected arcs.*
6. *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* $\langle var_1 = val_1, ..., var_n = val_n \rangle$*. The set of all bindings for a transition* t *is denoted* $B(t)$.

7. *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*.*
8. *A* step $Y \in BE_{MS}$ *is a non-empty, finite multiset of binding elements.*

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 and the formal definition of its semantics, we refer readers to [10].

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 [23] states that while imperative languages 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 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, but we rather aim to exploit the best in both of them. We consider Business Process Model and Notation (BPMN) [22], one of the most widely used standards for business process modeling, as an imperative modeling language, and Dynamic Condition Response (DCR) Graphs [20] as a declarative modeling technique.

Definition 3. *A dynamic condition response 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*
3. Act *is the set of actions*
4. $\rightarrow\bullet, \bullet\rightarrow \subseteq E \times E$ *are the condition relation and response relation*
5. $\rightarrow+, \rightarrow\% \subseteq E \times E$ *are the dynamic include relation and exclude relation satisfying that* $\forall e \in E \,.\, e \rightarrow+ \cap\, e \rightarrow\% = \emptyset$
6. $\rightarrow\diamond \subset E \times E$ *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 an event e is a condition for an event e' ($e \rightarrow\bullet e'$), but is excluded from the graph then it no longer restricts the execution of e'. Moreover, if event e' is the response for an event 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 an event b is a milestone of an event a ($b \rightarrow\diamond a$), then the event a cannot be executed as long as the event b is in the set of pending responses (Re).For more details on DCR Graphs and the formal definition of their semantics which establishes the dynamics of the graphs, we refer the readers to [20].

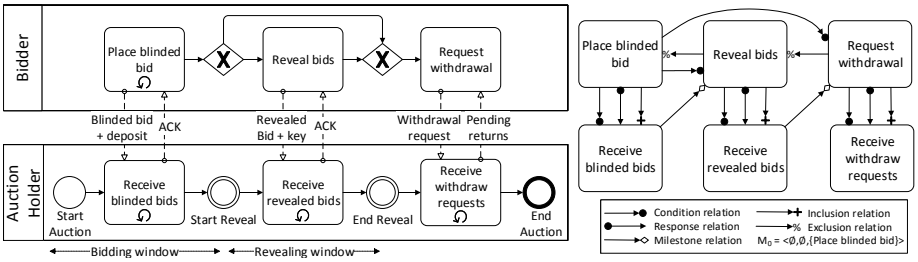
4 Use Case: Blind Auction

Our use case is adapted from [1]. 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 with the blinded 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 (Figure 1). Listing 1.1 is the corresponding Solidity contract.

```
contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;}
    uint public biddingEnd;
    uint public revealEnd;
    mapping(address => Bid[]) public bids;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;
    modifier onlyBefore(uint _time) {require(now<_time);_;}
    modifier onlyAfter(uint _time) {require(now>_time);_;}
    constructor(uint _biddingTime, uint _revealTime) public {
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }
```

```
    function bid(bytes32 _blindedBid) public payable onlyBefore(
        biddingEnd) {
        bids[msg.sender].push(Bid({blindedBid: _blindedBid, deposit: msg
            .value}));
    }
    function reveal(uint[] values, bytes32[] secrets) public onlyAfter(
        biddingEnd) onlyBefore(revealEnd) {
        require (values.length == secrets.length);
        for(uint i = 0; i < values.length && i < bids[msg.sender].length
            ; i ++) {
            var bid = bids[msg.sender][i];
            var (value, secret) = (values[i], secrets[i]);
            if(bid.blindedBid == keccak256(value, secret) && bid.deposit
                >= value && value > highestBid) {
                highestBid = value;
                highestBidder = msg.sender;
            }
        }
    }
    function withdraw() public onlyAfter
        (revealEnd) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            if (msg.sender != highestBidder)
                msg.sender.transfer(amount);
            else
                msg.sender.transfer(amount - highestBid);
            pendingReturns [msg.sender] = 0;}}}stBid) ("");
            pendingReturns [msg.sender] = 0;
        }
    }
}
```

Listing 1.1: Excerpt of the Blind Auction smart contract in Solidity



(a) Blind Auction: BPMN choreography (b) Blind Auction: DCR graph

Fig. 1: Blind Auction Workflow Representations

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 with regard to 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 provided behavior to be considered, and feeding it the model checker to verify the targeted property.

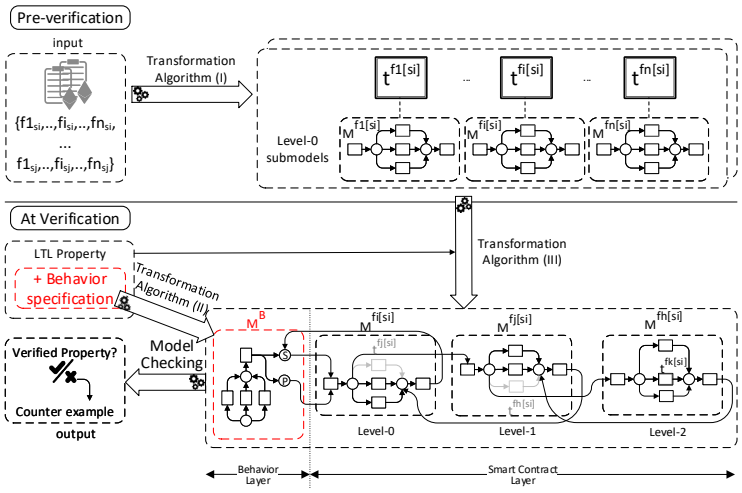


Fig. 2: Overview of the approach

More precisely, we opt for a hierarchical CPN model to represent the considered smart contracts' execution and interaction with respect to the provided behavior specification.

As shown in Figure 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 Figure 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., $t^{fj[si]}$ 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 hierarchical CPN model from the provided input artifacts.

6.1 Our HCPN Model: Defining its Elements

In this section, we give details on the elements of our proposed model.

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

1. aggregated transitions (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. They are transitions that can be substituted by submodels.
2. regular transitions (T^R): are simple unsubstitutable CPN transitions.

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

- $t.name$, the name of the transition t
- $t.statement$, the Solidity code associated to transition t
- $t.metaColour$, the metaColour associated to the control flow places of transition 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 $t.submodel.inTransitions$ designating its input (source) transitions and $t.submodel.outTransitions$ designating its output (sink) transitions
- $t.guard$, the guard of the transition t
- $\bullet t[cf] \in P_{CF} \cup P_S$, the input control flow place of t
- $\bullet t[input] \in P_P$, the input parameters place of t
- $\bullet t[data] \subseteq P_{data}$, the input data places of t
- $t \bullet [cf] \in P_{CF} \cup P_S$, the output control flow place of t
- $t \bullet [output] \in P_R$, the output return place of t
- $t \bullet [data] \subseteq P_{data}$, the output data places of t

Places P
At the first step of our modelling approach, no places are created at the level-0 model. For level-1 submodels, we define 4 types of places according to the role they play:

- *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 of level-0 as the concatenation of the *state* and the *input parameters*: *[uint: contractBalance, type$_{v_1}$: stateVariable$_1$, ..., type$_{v_n}$: stateVariable$_n$, type$_{p_1}$: inputParameter$_1$, ... , type$_{p_n}$: inputParameter$_n$]* (which corresponds to the concatenation of the colour of the input control flow place $\bullet t[cf] \in P_S$ and the colour of the input parameters place $\bullet t[input] \in P_P$ of the transition in question at the second step of our modelling approach).

- *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*$_1$*, ...,* $type_{p_n}$*: inputParameter*$_n$*]*.
- *Return places* P_R that communicate potential functions' returned data. Each function call has an associated return place whose colour corresponds to the return type of the called function.

At the second step of our modelling approach, two input places are created at the level-0 model for the aggregated transition corresponding to the function to be verified:

- a *state place* $p_s \in P_S$ representing the state of the smart contract. Its colour is as follows: *[uint: contractBalance,* $type_{v_1}$*: stateVariable*$_1$*, ...,* $type_{v_n}$*: stateVariable*$_n$*]*
- a *parameters place* $p_p \in P_P$ representing the input parameters of the function in question.

A *return place* $p_r \in P_R$ might also be created if the function has a return type.

Expressions E

An expression is a construct that can be made up of literals, variables, function calls and operators, according to the syntax of Solidity, that evaluates to a single value. For ease of representation later, we define three types of expressions:

- expressions with variables E_V: are expressions that 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: are expressions that 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: are expressions that do not make use of any variables nor function calls.

We note that an expression e can of course have both variables and function calls ($e \in E_V \wedge e \in E_F$).

Statements S

A statement $st \in \mathbb{S}$ can be either a compound statement $\{st[1]; st[2]; \ldots; st[N]\}$ (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:
  - $st_{LHS} = \emptyset$
  - $st_{RHS}.vars$ designates 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 one variable that designates the assigned one

  - $st_{RHS}.vars$ designates the set of variables used in the assignment expression (if $st_{RHS} \in E_V$)
  - $st_{RHS}.fctCalls$ designates the set of function calls used in the assignment expression (if $st_{RHS} \in E_F$)
- a variable declaration statement, where:
  - $st_{LHS} \in E_V$ and $st_{LHS}.vars$ contains one variable that designates the declared one
  - $st_{LHS}.type$ 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 $st_{RHS} \in E_V$)
  - $st_{RHS}.fctCalls$ designates the set of function calls used in the variable initialization expression (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 $st_{RHS} \in E_V$)
  - $st_{RHS}.fctCalls$ designates the set of function calls in the expression of the value to be sent (if $st_{RHS} \in E_F$)
- a returning statement, where:
  - $st_{LHS} = \emptyset$
  - $st_{RHS}.vars$ designates the variables in the expression of the returned value (if $st_{RHS} \in E_V$)
  - $st_{RHS}.fctCalls$ designates the function calls in the expression of 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 which can have:
  - a single-branching form: $if(c)\ then\ st_T$
  - a double-branching form: $if(c)\ then\ st_T\ else\ st_F$
- a looping statement which can be:
  - a for loop: $for(init;c;inc)\ st_T$
  - a while loop: $while(c)\ st_T$
- where:
  - c is a boolean expression
  - $c.vars$ designates the set of variables used in the condition (if $c \in E_V$)
  - $c.fctCalls$ designates 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 generate the aggregated transitions for the smart contracts' functions along with their level-0 submodels. To do so, we propose the following algorithms.

GENERATEAGGREGATIONS Algorithm

procedure GENERATEAGGREGATIONS(SC)
  Input: a Solidity smart contract *SC*
  Output: the aggregated transitions the CPN model of *SC*
  $metaColour \leftarrow [uint : contractBalance]$
  for $v \in SC.vars$ **do**
    add $(v.type : v.name)$ to $metacolour$
  end for
  for $f \in SC.fcts$ **do**
    create aggregated transition t^a
    $t^a.name \leftarrow f.name$
    $t^a.statement \leftarrow f.body$
    $newColour \leftarrow metaColour$
    for $p \in f.params$ **do**
      add $(p.type : p.name)$ to $newColour$
    end for
    $t^a.metaColour \leftarrow newColour$
  end for
end procedure

GENERATELEVEL0 Algorithm

procedure GENERATELEVEL0(t^a)
  Input: an aggregated transition t^a
  Output: the level-0 CPN submodel of t^a
  $P_{data} \leftarrow \emptyset$
  GETLOCALVARIABLES$(t^a.statement; P_{data})$
  $t^a \leftarrow P_{data}$
  $t^a.submodel \leftarrow$ CREATESUBMODEL$(t^a, \emptyset, \emptyset)$
end procedure

GETLOCALVARIABLES creates a set of places to be used in the submodel of a transition t^a, corresponding to the local variables used in its function. To do so, the statements in the function's body are recursively investigated in search for variable declaration statements. For each variable declaration statement found, a place bearing the name of the variable and its type as its name and colour is created and added to the set P_{data}. In addition to standalone variable declarations, we note that we can also find variables declared in the initialization of a For loop.

We opt for the construction of this set of places beforehand, as opposed to on the fly during the construction of the submodel, for the following reason. In Solidity, a variable can be used before its declaration (as long as a declaration does exist). Creating its corresponding place on the fly while creating the submodel of a transition would consequently require testing for its existence every time the variable is used in a statement, as the creation of the place in question may have to happen prior to the declaration statement, in any other statement using it (as part of st_{LHS} or st_{RHS}) for the first time. On this account, we judge it more efficient to sweep the code first for the construction of P_{data}.

GETLOCALVARIABLES

1: **procedure** GETLOCALVARIABLES(st; P_{data})
2: **Input:** statement *st*, set of places P_{data} being created
3: **Output:** updated P_{data} with the set of places corresponding to local variables in the statement *st*
4: **if** *st* is a variable declaration statement **then**
5: create place *p*
6: $p.name \leftarrow st_{LHS}.vars.name$
7: $p.colour \leftarrow st_{LHS}.type$
8: add *p* to P_{data}
9: **else if** *st* is a selection statement **then**
10: GETLOCALVARIABLES(st_T, P_{data})
11: **if** *st* is a double-branching selection statement **then**
12: GETLOCALVARIABLES(st_F, P_{data})
13: **end if**
14: **else if** *st* is a looping statement **then**
15: **if** *st* is a for statement: $for(init;c;inc)st_T$ **then**
16: GETLOCALVARIABLES$(init, P_{data})$
17: GETLOCALVARIABLES(st_T, P_{data})
18: **else if** *st* is a while statement: $while(c)st_T$ **then**
19: GETLOCALVARIABLES(st_T, P_{data})
20: **end if**
21: **else if** *st* is a compound statement $\{st[1]; st[2]; \ldots; st[N]\}$ **then**
22: **for** $i = 1..N$ **do**
23: GETLOCALVARIABLES$(st[i], P_{data})$
24: **end for**
25: **end if**
26: **end procedure**

We see a smart contract function as a set of statements. To each one of the statement types we define a corresponding pattern in CPN, according to which a snippet of a CPN model is generated. The resulting snippets are linked according to the function's internal workflow. The *createSubModel* implements such correspondences[1].

CREATESUBMODEL Algorithm

1: **procedure** CREATESUBMODEL(t; st; p_{in}; p_{out})
2: **Input:** transition *t*, statement *st*, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** submodel of transition *t*
4: **switch** *st* **do**
5: **case** compound statement $\{st[1]; st[2]; \ldots; st[N]\}$
6: BUILDCOMPOUNDSTATEMENT$(t;st;p_{in};p_{out})$
7: **case** simple statement

[1] We note that in case a place does not exist ($p = \emptyset$) then any arc creation involving that place does not take effect.

```
            switch st do
                case assignment statement
                    BUILDASSIGNMENTSTATEMENT(t;st;p_in;p_out)
                case variable declaration statement
                    BUILDVARIABLEDECLARATIONSTATEMENT(tst;;p_in;p_out)
                case sending statement
                    BUILDSENDINGSTATEMENT(t;st;p_in;p_out)
                case returning statement
                    BUILDRETURNINGSTATEMENT(t;st;p_in;p_out)
                case function call statement
                    BUILDFUNCTIONCALLSTATEMENT(t;st;p_in;p_out)
            end switch
        case control statement
            switch st do
                case requirement statement
                    BUILDREQUIREMENTSTATEMENT(t;st;p_in;p_out)
                case selection statement
                    BUILDSELECTIONSTATEMENT(t;st;p_in;p_out)
                case looping statement
                    switch st do
                        case for statement
                            BUILDFORLOOPSTATEMENT(t;st;p_in;p_out)
                        case while statement
                            BUILDWHILELOOPSTATEMENT(t;st;p_in;p_out)
                    end switch
            end switch
    end switch
end procedure
```

BUILDCOMPOUNDSTATEMENT Algorithm

```
1:  procedure BUILDCOMPOUNDSTATEMENT(t; st; p_in; p_out)
2:      Input: transition t, a compound statement st ={st[1]; st[2]; ...; st[N]}, control
        flow input place p_in, control flow output place p_out
3:      Output: submodel for statement st
4:      for i = 1..N − 1 do
5:          create place p_i
6:      end for
7:      CREATESUBMODEL(t;st[1];p_in;p_1)
8:      for i = 2..N − 1 do
9:          CREATESUBMODEL(t;st[i];p_i-1;p_i)
10:     end for
11:     CREATESUBMODEL(t;st[N];p_N-1;p_out)
12: end procedure
```

BUILDASSIGNMENTSTATEMENT Algorithm

procedure BUILDASSIGNMENTSTATEMENT(t; st; p_{in}; p_{out})
  Input: transition t, an assignment statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
  Output: submodel for statement st
  create transition t'
  create arc from p_{in} to t'
  CONNECTLOCALVARIABLES$(st_{RHS}.vars \backslash \{st_{LHS}.vars\}; t; t')$
  CONNECTFUNCTIONCALLS$(st_{RHS}.fctCalls; t)$
  if $st_{LHS}.vars$ is a local variable **then**
    create arc from $t.data[st_{LHS}.vars]$ to t'
    create arc from t' to $t.data[st_{LHS}.vars]$ with inscription st_{RHS}
    create arc from t' to p_{out}
  else
    create arc from t' to p_{out} with inscription $outInsc \leftarrow inInsc$ in which the variable corresponding to $st_{LHS}.vars$ is replaced by st_{RHS}
  end if
end procedure

BUILDVARIABLEDECLARATIONSTATEMENT Algorithm

procedure BUILDVARIABLEDECLARATIONSTATEMENT(t; st; p_{in}; p_{out})
  Input: transition t, a variable declaration statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
  Output: submodel for statement st
  create transition t'
  create arc from p_{in} to t'
  CONNECTLOCALVARIABLES$(st_{RHS}.vars; t; t')$
  CONNECTFUNCTIONCALLS$(st_{RHS}.fctCalls; t)$
  create arc from t' to $t.data[st_{LHS}.vars]$ with inscription st_{RHS}
  create arc from t' to p_{out}
end procedure

BUILDSENDINGSTATEMENT Algorithm

procedure BUILDSENDINGSTATEMENT(t; st; p_{in}; p_{out})
  Input: transition t, a sending statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
  Output: submodel for statement st
  create transition t'
  create arc from p_{in} to t'
  CONNECTLOCALVARIABLES$(st_{RHS}.vars; t; t')$
  CONNECTFUNCTIONCALLS$(st_{RHS}.fctCalls; t)$
  create arc from t' to p_{out} with inscription $outInsc \leftarrow inInsc$ in which the variable corresponding to the sender's (respectively the contract's) *balance* is incremented (respectively decremented) by st_{RHS}
end procedure

BUILDRETURNINGSTATEMENT Algorithm

procedure BUILDRETURNINGSTATEMENT(t; st; p_{in}; p_{out})
    Input: transition t, a returning statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
    Output: submodel for statement st
    create transition t'
    create arc from p_{in} to t'
    CONNECTLOCALVARIABLES($st_{RHS}.vars;t;t'$)
    CONNECTFUNCTIONCALLS($st_{RHS}.fctCalls;t$)
    create arc from t' to $t \bullet [cf]$
    create arc from t' to $t \bullet [output]$ with inscription $outInsc \leftarrow$ [$inInsc.sender$, $inInsc.balance$, st_{RHS}]
end procedure

BUILDFUNCTIONCALLSTATEMENT Algorithm

procedure BUILDFUNCTIONCALLSTATEMENT(t; st; p_{in}; p_{out})
    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
    create transition t^f
    create place p_{param_f}
    create arc from p_{in} to t^f
    create arc from p_{param_f} to t^f
    CONNECTLOCALVARIABLES($f_{RHS}.vars;t;t^f$)
    CONNECTFUNCTIONCALLS($f_{RHS}.fctCalls;t$)
    create arc from t^f to p_{out} with a placeholder inscription
end procedure

BUILDREQUIREMENTSTATEMENT Algorithm

procedure BUILDREQUIREMENTSTATEMENT(t; st; p_{in}; p_{out})
    Input: transition t, a requirement statement $st = require(c)$, control flow input place p_{in}, control flow output place p_{out}
    Output: submodel for statement st
    create transition t_{revert}
    $t_{revert}.guard \leftarrow !c$
    create arc from p_{in} to t_{revert}
    create arc from t_{revert} to $\bullet t[cf]$
    CONNECTLOCALVARIABLES($c.vars;t;t_{revert}$)
    CONNECTFUNCTIONCALLS($c.fctCalls;t_{revert}$)
    create transition $t_{!revert}$
    $t_{!revert}.guard \leftarrow c$
    create arc from p_{in} to $t_{!revert}$
    create arc from $t_{!revert}$ to p_{out}
    CONNECTLOCALVARIABLES($c.vars;t;t_{!revert}$)

15: CONNECTFUNCTIONCALLS*(c.fctCalls;t$_{!revert}$)*
16: **end procedure**

BUILDSELECTIONSTATEMENT Algorithm

1: **procedure** BUILDSELECTIONSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, a selection statement $st = if(c)\ then\ st_T\ [else\ st_F]$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** submodel for statement *st*
4: create place p_T
5: create transition t_T
6: $t_T.guard \leftarrow c$
7: create arc from p_{in} to t_T
8: create arc from t_T to p_T
9: CONNECTLOCALVARIABLES*(c.vars;t;t$_T$)*
10: CONNECTFUNCTIONCALLS*(c.fctCalls;t$_T$)*
11: CREATESUBMODEL*(t;st$_T$;p$_T$;p$_{out}$)*
12: create transition t_F
13: $t_F.guard \leftarrow !c$
14: create arc from p_{in} to t_F
15: CONNECTLOCALVARIABLES*(c.vars;t;t$_F$)*
16: CONNECTFUNCTIONCALLS*(c.fctCalls;t$_F$)*
17: **if** *st* is a selection statement: $if(c)\ then\ st_T$ **then**
18: create arc from t_F to p_{out}
19: **else if** *st* is a selection statement: $if(c)\ then\ st_T\ else\ st_F$ **then**
20: create place p_F
21: create arc from t_F to p_F
22: CREATESUBMODEL*(t;st$_F$;p$_F$;p$_{out}$)*
23: **end if**
24: **end procedure**

BUILDFORLOOPSTATEMENT Algorithm

1: **procedure** BUILDFORLOOPSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, a *for* looping statement $st = for(init;c;inc)\ st_T$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** submodel for statement *st*
4: create place p_{init}
5: create place p_c
6: create place p_T
7: CREATESUBMODEL*(t;init;p$_{in}$;p$_{init}$)*
8: create transition t_T
9: $t_T.guard \leftarrow c$
10: create arc from p_{init} to t_T
11: CONNECTLOCALVARIABLES*(c.vars;t;t$_T$)*
12: CONNECTFUNCTIONCALLS*(c.fctCalls;t$_T$)*
13: create arc from t_T to p_c

14: create transition t_F
15: $t_F.guard \leftarrow !c$
16: create arc from p_{init} to t_F
17: CONNECTLOCALVARIABLES*(c.vars;t;t_F)*
18: CONNECTFUNCTIONCALLS*(c.fctCalls;t_F)*
19: create arc from t_F to p_{out}
20: CREATESUBMODEL*(t;st_T;p_c;p_T)*
21: CREATESUBMODEL*(t;inc;p_T;p_{init})*
22: **end procedure**

BUILDWHILELOOPSTATEMENT Algorithm

1: **procedure** BUILDWHILELOOPSTATEMENT(t; st; p_{in}; p_{out})
2: **Input:** transition t, a *while* looping statement $st = while(c)\ st_T\ st_T$, control flow input place p_{in}, control flow output place p_{out}
3: **Output:** submodel for statement st
4: create place p_T
5: create transition t_T
6: $t_T.guard \leftarrow c$
7: create arc from p_{in} to t_T
8: CONNECTLOCALVARIABLES*(c.vars;t;t_T)*
9: CONNECTFUNCTIONCALLS*(c.fctCalls;t_T)*
10: create arc from t_T to p_T
11: create transition t_F
12: $t_F.guard \leftarrow !c$
13: create arc from p_{in} to t_F
14: CONNECTLOCALVARIABLES*(c.vars;t;t_F)*
15: CONNECTFUNCTIONCALLS*(c.fctCalls;t_F)*
16: create arc from t_F to p_{out}
17: CREATESUBMODEL*(t;st_T;p_T;p_{in})*
18: **end procedure**

CONNECTLOCALVARIABLES Algorithm

1: **procedure** CONNECTLOCALVARIABLES(V; t; t')
2: **Input:** set of local variables V, transition t, transition t'
3: **Output:** submodel with connections to local variables
4: **for** $v \in V$ **do**
5: create arc from $t.data[v]$ to t'
6: create arc from t' to $t.data[v]$
7: **end for**
8: **end procedure**

CONNECTFUNCTIONCALLS Algorithm

1: **procedure** CONNECTFUNCTIONCALLS(FC; t)
2: **Input:** set of function calls FC, transition t

3: **Output:** submodel with connections to function calls
4: **for** $f \in FC$ **do**
5: create transition t^f
6: create place p_{return_f}
7: create place p_{param_f}
8: CONNECTLOCALVARIABLES($f_{RHS}.vars;t;t^f$)
9: create arc from p_{param_f} to t^f with inscription in which every element of f_{RHS} is replaced by its corresponding argument
10: create arc from t^f to p_{return_f} with a placeholder inscription
11: **end for**
12: **end procedure**

The second step of our modelling approach consists in contextualizing the function to be verified. To do so, two places are created to represent the state of the smart contract and the call arguments for the function in question and are linked to its respective aggregated transition in the level-0 model. This transition, as well as potential aggregated transitions within its submodel are unfolded depending on the property to be verified. In the following, we present the algorithm to apply to unfold an aggregated transition.

UNFOLDTRANSITION Algorithm

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

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 Figure 3a) 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 [18] 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 such that:*

- $P = \{S\}$
- $T = \{t_i, \forall i \in [1,n]\}$, *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* $n = |E|$
- $E = \{a \rightarrow< Ex, Re, In >, \forall a \in A \cap (P \cup T)\} \cup \{a \rightarrow< Ex', Re', In' >, \forall a \in A \cap (T \cup P)\}$ *with (1)* $Ex' = Ex \cup e_i$, *(2)* $Re' = (Re \backslash e_i) \cup e\bullet\rightarrow$ *and (3)* $In' = (In \cup e_i \rightarrow+) \backslash e \rightarrow\%$
- $I = \{S \rightarrow< S_1, S_2, S_3 >\}$ *with* $< S_1, S_2, S_3 >$ *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:

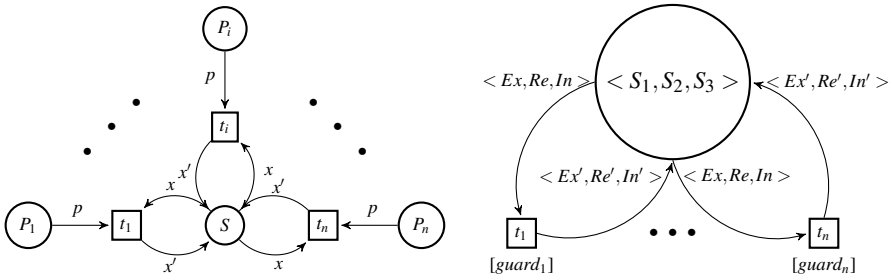
- $guard_i$ *is the conjunction of the conditions defining the enabling of the corresponding event (1)* e_i*:* $i \in In$, *(2)* $(\rightarrow\bullet i \cap In) \in Ex$ *and (3)* $(\rightarrow\diamond i \cap In) \in E \backslash Re$
- *the expression* $< Ex', Re', In' >$ *on its output arc is defined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

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*[2].

7 Model Checking: Application 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* [8] 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 [6], 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.

[2] We include a proof of this theorem in: https://github.com/garfatta/ICSOC21/proof.pdf



(a) CPN model for a completely-free behavior (b) CPN model for a DCR Graph

Fig. 3: Behavior Representations

7.1 Expressing Vulnerabilities in LTL

In the following, $t^f_{s_i}$ designates the CPN aggregated transition corresponding to function f in smart contract s_i.

Integer Overflow/Underflow: 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:

$$IUO(t^f_{s_i}) = \Box\neg outOfRange(x)$$

Where *outOfRange(x)* is a proposition defining the conditions for overflow and underflow for the variable x w.r.t the range of its type which we delimit by defining lower and higher thresholds (*minThreshold* and *maxThreshold* respectively).

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

Reentrancy: This vulnerability is related to functions that contain instructions responsible for Ether transfer. Its checking is therefore applicable on functions in whose body a *sending statement* exists. To detect such a vulnerability, we propose two LTL properties. The first one is a safety property defined as follows:

$$Reentrancy(t^f_{s_i}) = containsSending(t^f_{s_i}) \rightarrow \Box\neg reentrant$$

Where *reentrant($t^f_{s_i}$)* is a proposition defining the necessary condition under which a reentrancy vulnerability can be detected in the function f in the smart contract s_i. This condition can only be defined when the user indicates the variable x serving as a record for balances and whose assignment should be watched. Such a condition expresses the presence of a sending statement which is not preceded by an assignment to x:

$$reentrant = (\neg assignment(x)) \cup sendingTo(s_j)$$

This property is used when we only have the code of the smart contract to be verified (i.e., a totally free behaviour). In case the code of the interacting smart contract s_j is available, we propose the following LTL property:

$$NoReentrancy(t^f_{s_i}) = containsSending(t^f_{s_i}) \rightarrow (sendingTo(s_j) \rightarrow \circ\Box((\neg sendingTo(s_j)) \cup end(t^{fallback}_{s_j})))$$

Using this property we can verify that once the sending statement is executed, it cannot be executed again until the fallback function of the receiving contract has finished and therefore no reentrancy breach can happen.

Self-destruction: This vulnerability is checked by detecting the presence of a test containing *this.balance* in the code of the inspected function:

$$selfDestruction(t^f_{s_i}) = \neg testOnBalance(t^f_{s_i})$$

This detection process can be further enhanced when the code of the interacting smart contract is available. In that case, a function f in s_i is considered safe against this vulnerability if it does not contain a test on *this.balance* or if the interacting contract s_j does not contain a self destruction instruction or if the latter cannot be executed prior to the function under inspection, which is expressed by the following LTL property:

$$selfDestruction(t^f_{s_i}) = \neg(testOnBalance(t^f_{s_i}) \wedge containsSelfDestruct(t^g_{s_j}, s_i)) \vee (\neg selfDestruct(t^g_{s_j}, s_i) \cup start(t^f_{s_i}))$$

We note that even though these properties can detect the presence of the self destruction vulnerability, more information on what the function exactly does needs to be provided in order to be able to assess its harmfulness on the execution. This can still be checked by evaluating a contract-specific property.

Timestamp Dependence: In order to check for this vulnerability, we propose an LTL property to detect the accessibility of *block.timestamp* or its alias *now*:

$$TSD(t^f_{s_i}) = \Box\neg isTimestampDependant$$

Where *isTimestampDependant* defines a state's dependency to the block's timestamp. Similarly to the self destruction vulnerability, the presence of timestamp dependence can be detected using the proposed property, but to check the harm it may incur a more appropriate contract-specific property needs to be evaluated.

Skip Empty String Literal: This can be checked for the function calls contained in the definition of a function f by verifying that no empty string is passed as an argument (except for the last one) to any of the function calls. We express this as follows:

$$SkipEmpty(t^f_{s_i}) = containsFunctionCall(t^f_{s_i}) \rightarrow (\Box\neg(isFunctionCall \wedge \exists arg \in functionCall \backslash (arg = \text{“”} \wedge \neg isLast(arg))))$$

Uninitialized Storage Variable: This can be checked for each variable x of a complex type by the LTL property:

$$UnintializedVariable(t^f_{s_i}) = isOfComplexType(x) \rightarrow (\neg read(x) \cup write(x))$$

Where *read(x)* is true when x is read in a state and *write(x)* is true when it is assigned.

7.2 Application on the Use Case

The application of our approach on the use case presented in Section 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 Figure 4.

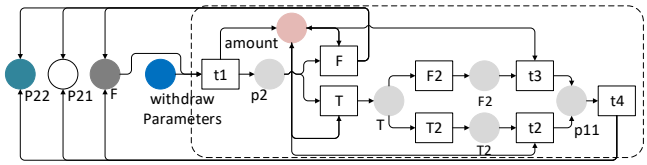


Fig. 4: SubModel of transition *withdraw*

Verifying properties of the contract would come down to verifying properties on the corresponding CPN model. For model checking, we chose *Helena* [8] 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 Section 4. We have then written the corresponding properties in *Helena*'s language for the vulnerabilities in Section 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. 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[3].

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

[3] https://github.com/garfatta/ICSOC21

embedding it with an extension to an existing technique previously developed to deal with the state space explosion problem in regular PNs [13] and applying it on CPNs.

References

1. Solidity documentation. https://docs.soliditylang.org/en/latest/
2. 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. p. 66–77. New York, NY, USA (2018)
3. Anand, S., Pasareanu, C.S., Visser, W.: Symbolic execution with abstraction. Int. J. Softw. Tools Technol. Transf. **11**(1), 53–67 (2009)
4. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Béguelin, S.Z.: 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, October 24 (2016)
5. 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, Klagenfurt, Austria, February 20-24, 2017. pp. 442–446 (2017)
6. Dingman, W., Cohen, A., Ferrara, N., Lynch, A., Jasinski, P., Black, P.E., Deng, L.: Defects and vulnerabilities in smart contracts, a classification using the NIST bugs framework. IJNDC **7**(3), 121–132 (2019)
7. Duo, W., Huang, X., Ma, X.: Formal analysis of smart contract based on colored petri nets. IEEE Intell. Syst. **35**(3), 19–30 (2020)
8. Evangelista, S.: High level petri nets analysis with helena. In: Applications and Theory of Petri Nets 2005. pp. 455–464. Berlin, Heidelberg (2005)
9. 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, Dunedin, New Zealand, 1-5 February, 2021. pp. 3:1–3:10. ACM (2021)
10. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer Publishing Company, Incorporated, 1st edn. (2009)
11. 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, February 18-21, 2018 (2018)
12. 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, April 7-11, Proceedings (2003)
13. 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, June 23-27, 2008. Proceedings. pp. 288–306 (2008)
14. 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, Milwaukee, WI, USA, July 15-19, 2019, Volume 2. pp. 555–560. IEEE (2019)
15. 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)
16. 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, Vienna, Austria, October 24-28, 2016. pp. 254–269 (2016)

17. Mavridou, A., Laszka, A., Stachtiari, 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, February 18-22, 2019. pp. 446–465 (2019)
18. 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)
19. Mendling, J., Weber, I., van der Aalst, W.M.P., vom Brocke, J., Cabanillas, C., Daniel, F., Debois, S., Ciccio, C.D., Dumas, M., Dustdar, S., Gal, A., García-Bañuelos, L., Governatori, G., Hull, R., Rosa, M.L., Leopold, H., Leymann, F., Recker, J., Reichert, M., Reijers, H.A., Rinderle-Ma, S., Solti, A., Rosemann, M., Schulte, S., Singh, M.P., Slaats, T., Staples, M., Weber, B., Weidlich, M., Weske, M., Xu, X., Zhu, L.: Blockchains for business process management - challenges and opportunities. ACM Trans. Manag. Inf. Syst. **9**(1), 1–16 (2018)
20. Mukkamala, R.R.: A Formal Model For Declarative Workflows Dynamic Condition Response Graphs. Ph.D. thesis (06 2012)
21. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
22. OMG: Business process model and notation (bpmn) 2.0. http://www.omg.org/spec/BPMN/2.0/ (2011)
23. 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, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I. vol. 99, pp. 383–394 (2011)
24. 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, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 664–676 (2018)
25. 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 co-located with (BPM 2018), Sydney, Australia, September 9-14, 2018. vol. 2196, pp. 56–60 (2018)