A Solidity-to-CPN Approach Towards Formal Verification of Smart Contracts

Abstract—**With its span of applications widening by the day, particularly thanks to its smart contract feature, the Blockchain technology is found to be of interest not only to investors but also to malicious users whose aim is to exploit its vulnerabilities, putting different Blockchain platforms under attack. It is therefore an inescapable necessity to guarantee the correctness of smart contracts as they are the core of Blockchain applications. Existing verification approaches, however, focus on targeting specific vulnerabilities, seldom supporting the verification of contract-specific properties.**

In this paper, we propose a transformation of Solidity smart contracts into Coloured Petri nets and investigate the capability of CPN Tools to verify CTL properties, which translate properties specific to the modeled contract, on the generated model.

Index Terms—**Blockchain; Formal Verification; Smart Contract; Solidity; Coloured Petri Nets; CTL properties.**

1. Introduction

Within the span of the two last decades, many advances have been made in the world of Blockchain, allowing it to go from being an underlying technology to a decentralized electronic cash system [1] at its inceptive stage, to being the backbone of platforms for the development of decentralized applications. One of the most important innovations that led to second-generation Blockchains is the introduction of smart contracts which first came with the Ethereum Blockchain. Smart contracts are pieces of script code that act like autonomous software agents, used to enforce management rules on the execution of transactions on the Blockchain. They are stored on and executed by the Blockchain and therefore inherit its characteristics, particularly its immutability. This same feature can, however, turn into a weak spot for such contracts. In fact, as a smart contract cannot be altered once it has been deployed on the Blockchain, it goes without saying that it cannot be corrected either, which makes verifying its correctness prior to its deployment an indispensable necessity. Many attacks on multiple Blockchain platforms have stemmed from smart contract vulnerabilities, like the attack exploiting an integer overflow vulnerability on the Bitcoin blockchain in August 2010 and the DAO attack exploiting a reentrancy vulnerability on Ethereum in June 2016.

Researchers have quickly started to address such problems by proposing informal as well as formal verification methods to enhance the security of smart contracts and ensure their correctness. While informal techniques can be useful to test a smart contract under certain scenarios, they cannot be relied on to verify its correctness regarding certain properties, which is where formal techniques prove to be efficient.

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 as it is the most popular language for Ethereum smart contracts. In this paper, we present the first step towards a formal verification approach, based on Coloured Petri Nets [2], for Solidity smart contracts. We propose an algorithm for the transformation of a Solidity smart contract into a CPN model over which CTL properties can be verified.

Our paper is structured as follows. Section 2 sets the context of our work by providing essential prerequisites and Section 3 gives an overview of the existing relative studies. A use case is presented in Section 4 to serve as an application example for our contribution introduced in Section 5. Our verification results are presented in Section 6 followed by a conclusion and thoughts about future work in Section 7.

2. Background

In this section we give an overview of the Blockchain technology with particular focus on Solidity smart contracts, and introduce CPN-related concepts for a better understanding of our contribution.

2.1. On Solidity Smart Contracts

Solidity [3] is the most commonly-used high-level programming language for smart contracts under Ethereum, one of the leading public Blockchain platforms nowadays. The faithful execution of a smart contract is governed and guaranteed by the Ethereum Virtual Machine (EVM) semantics and its immutable nature gives it a sense of finality.
A Solidity smart contract is a collection of code and data, residing at a specific address on the Ethereum Blockchain, which can be invoked either by an internal account (i.e., a smart contract) or directly by an external account (i.e., user). Every account is characterized by a persistent *storage* (null in case of an external account) and a balance in Ether which is adjusted by transactions. A transaction is a message used to send ether from one account to another and/or invoke a smart contract's function if the message includes a payload and the targeted account is an internal one. The execution of such a payload is carried out according to a *stack* machine

called the EVM. Every smart contract features a *memory* which is cleared at each message call. It can also access certain properties of the current block.

A Solidity smart contract may look like a JavaScript or C program syntax-wise, but they are actually dissimilar since the underlying semantics of Solidity functions differently from traditional programs. This naturally calls on more vigilance from the programmers who might be faced by unconventional security issues as vulnerabilities in smart contracts seem to often stem from this gap between the semantics of Solidity and the intentions of the programmer [4].

2.2. On Coloured Petri Nets

A Petri net [5] 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). In the following, we give formal definitions of PN-related concepts with particular focus on the Coloured PN extension, to support the reader's understanding of our contribution.

Definition 1 (Petri net). A Petri net is a tuple $PN = (P, T, F)$ where P is the set of places, T is the set of transitions such that $P \cap T = \emptyset$ and $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs connecting the places and transitions, also referred to as the flow relation.

$\bullet p$ and $p\bullet$ respectively denote the sets of input and output transitions of a place $p \in P$, while $\bullet t$ and $t\bullet$ respectively denote the sets of input and output places of a transition $t \in T$.

The execution semantics of a Petri net are formally defined in terms of its *markings* and *fired transitions*.

Definition 2 (Marked Petri net, enabled and fired transition). A marked Petri net is denoted by $Pn_M = (Pn, M)$ where Pn is a Petri net and $M : P \to \mathbb{N}$ is a function that assigns a number of tokens to each place. A transition $t \in T$ is enabled in the marking M, denoted by $M[t\rangle$ iff $\forall p \in \bullet t : M(p) \geq 1$, in which case t can be fired. Firing a transition t under a marking M results in a new marking M' such that: $\forall p \in \bullet t : M'(p) = M(p) - 1 \ \wedge \ \forall p \in t\bullet : M'(p) = M(p) + 1$. This can be denoted by $M[t\rangle M'$.

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 [6], [7] 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 Coloured Petri Net (CP-net or CPN) [2] 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 [8], [9], to define data types. The formal definition of a Coloured Petri net is given in Definition 3 and the main concepts needed to define its dynamics are given in Definition 4.

Definition 3 (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 such that $Type[G(t)] = Bool$.
8) $E : A \to EXPR_V$ is an *arc expression function* that assigns an arc expression to each arc a such that $Type[E(a)] = C(p)_{MS}$, where p is the place connected to the arc a (i.e., the type of the arc expression is a multiset type over the colour set of the connected place).
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 4 (CPN concepts). For a Coloured Petri Net $CPN = (P, T, A, \Sigma, V, C, G, E, I)$, we define the following concepts:

1) A *marking* is a function M that maps each place $p \in P$ into a multiset of tokens $M(p) \in C(p)_{MS}$.
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 $Var(t) \subseteq V$ and consist of the free variables appearing in the guard of t and in the arc expressions of arcs connected to t.
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 $\langle var_1 = val_1, ..., var_n = val_n\rangle$. 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.
6) A *step* $Y \in BE_{MS}$ is a non-empty, finite multiset of binding elements.

3. Related Works

Existing studies on the formal verification of smart contracts follow two main streams. The first group of studies are based on theorem proving [10], [11], [12], [13]. The general idea here is to transform the smart contract's code (often its corresponding EVM bytecode) into a code processable by a certain theorem prover and use the latter to discharge proofs on the correctness of the generated code. In this case, the verification is not automated and requires the user's expertise in the manipulation of the used theorem prover as well as manual intervention in discharging proofs.

The second group of studies are based on symbolic model checking coupled with complementary techniques such as symbolic execution [14] and abstraction [15]. The first attempt falling under this category is Oyente [16], a tool that targets 4 vulnerabilities, namely transaction order dependence, timestamp dependence, mishandled exceptions and reentrency. 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, as is the case with GASPER [17] which reuses Oyente's generated control flow graph for the detection of bytecode patterns with high gas costs, and some others extended it with the aim of supporting the detection of other vulnerabilities, as is the case with MAIAN [18], SASC [19] and Osiris [20]).
Also based on symbolic model checking, we find Zeus [21] which operates on the source code of the smart contract. The user needs to specify the criteria to be checked as a CHC-based policy written in XACML, according to which the code is instrumented before its translation into a low-level intermediate representation (LLVM bitcode) which is then fed to an existing verification engine.
VeriSolid [22], initially introduced as [23], is an FSM-based approach that unlike the previous ones, aims at producing a correct-by-design smart contract rather than detecting bugs. The authors propose a transformation of a smart contract modeled as an FSM into a Solidity code and provide the user with the ability to specify intended behavior in the form of liveness, deadlock freedom and safety properties that can be expressed using customizable templates for CTL properties and checked by a backend symbolic model checker.

In order to use symbolic execution to generate the traces that would be used for the verification, 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 properties, in which case they are control flow-related properties. In fact, none of these studies target data-related properties.

It is worth mentioning that most of the proposed approaches operate on the EVM bytecode rather than on the Solidity code because of the latter's lack of formal semantics. This, however, results in loss of contextual information, and consequently limits the range of properties that can be verified on the contract.

4. Use Case: Blind Auction

The use case presented in this section is adapted from [22], which was in turn adapted from [24].

Participants in a blind auction have a bidding window during which they can place their bids. A participant can place more than one bid as long as the bidding window is still open. The placed bid is blinded in the sense that only a hashed value is submitted at this stage, and yet it is still binding because the bidder has to make a deposit along the blinded bid, with a value that is supposedly greater than that of the real bid. Once the bidding window is closed, the revealing window is opened. During this stage of the auction, participants with at least one placed bid proceed to reveal them. Revealing a bid consists in the participant sending the actual value of the bid along with the key used in its hash, and the system verifying whether the sent values do correspond with the previously placed blinded bid and potentially updating the highest bid and bidder's values accordingly. If the revealed value of a bid does not correspond with its blinded value, or is greater than the deposit made previously along the blinded bid, the said bid is considered invalid. Once the revealing window is closed, every bid left unrevealed is discarded from the auction. Participants can then 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 value of the bid is restored. The auction is considered to be terminated when all participants withdraw their deposits. Figure 1 describes the workflow of the blind auction system and Listing 1 represents the Solidity smart contract implementing it.

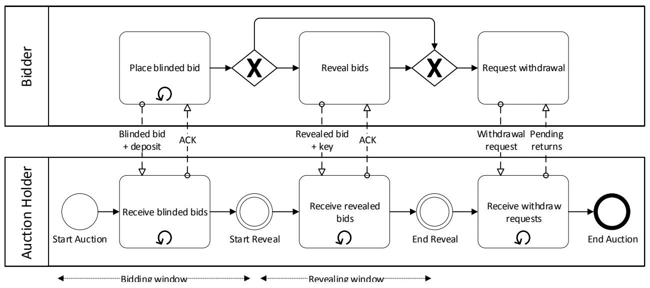


Figure 1. Blind Auction Workflow

```
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;
        }
    }
}
```

Listing 1. Blind Auction smart contract in Solidity

5. Solidity-to-CPN Transformation

This section represents the core of our contribution. Herein we propose an algorithm that transforms a Solidity smart contract into a hierarchical CPN model over which CTL properties can be verified (cf. Section 6). The general idea here is to start from a CPN model representing the general workflow of the smart contract (we refer to this as the level-0 model) and then to build on it by embedding it with submodels representing the execution of the smart contract functions (we refer to these submodels as level-1 models). In a level-0 model, we distinguish 2 parts, namely the *user's behaviour* part which models the way users can interact with the system and the *smart contract's behaviour* part which represents the system. These two are linked via *communication places*. Figure 2 shows the level-0 model of the previously described blind auction use case with the level-1 submodel for transition *bid*.

We lead off by setting some notations on the elements of the model in Section 5.1 before getting into the algorithm in Section 5.2.

5.1. Notations on the Model's Elements

5.1.1. Places P.

- In the Smart Contract's Behaviour part, we define *state places* P_S as places that hold the state of the smart contract, namely the contract's balance and the values of the state (global) variables. Their colour is as follows *[uint: contractBalance, type$_1$: stateVariable$_1$, ..., type$_n$: stateVariable$_n$]*
- Among communication places, we distinguish:
  - *parameter places* P_P: that convey potential inputs of function calls along with the caller's information and the transferred value. Their colour is as follows *[address: sender, uint: balance, uint: value, type$_1$: inputParameter$_1$, ..., type$_n$: inputParameter$_n$]*
  - *return places* P_R: that communicate potential returned data from functions along with the caller's updated information. Their colour is as follows *[address: sender, uint: balance, type$_R$: returnParameter]*
  - *interface places* P_I: that handle handovers between the function calls and their execution by the system. They are uncoloured (basic) places.
- On level 1 (a transition's detailed sub-model), we distinguish 2 types of places:
  - *control flow places* P_{cf}. Their colour is a $metaColour$ defined at each transition of level 0 as the concatenation of the colour of its input control flow place $\bullet t[cf] \in P_S$ and the colour of its input parameters place $\bullet t[input] \in P_P$.
  - *data places* P_{data} (for internal local variables) where each place is of a colour that corresponds to the type of the variable it represents.
- For a place $p \in P$, we can access its identifier ($p.name$) and its colour ($p.colour$)

5.1.2. Transitions T. For a transition $t \in T$ we distinguish:

- $\bullet t[cf] \in P_{cf} \cup P_S$, the input control flow place of t
- $\bullet t[input] \in P_P$, the input parameters place of t
- $\bullet t[com] \in P_C$, the input communication place of t
- $\bullet t[data] \subseteq P_{data}$, the input data places of t (in case of a submodel transition) item $t \bullet [cf] \in P_{cf} \cup P_S$, 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 (in case of a submodel transition)
- $t \bullet [com] \in P_C$, the output communication place of t
- $t^{action} \in S$: action of t (the function's body)

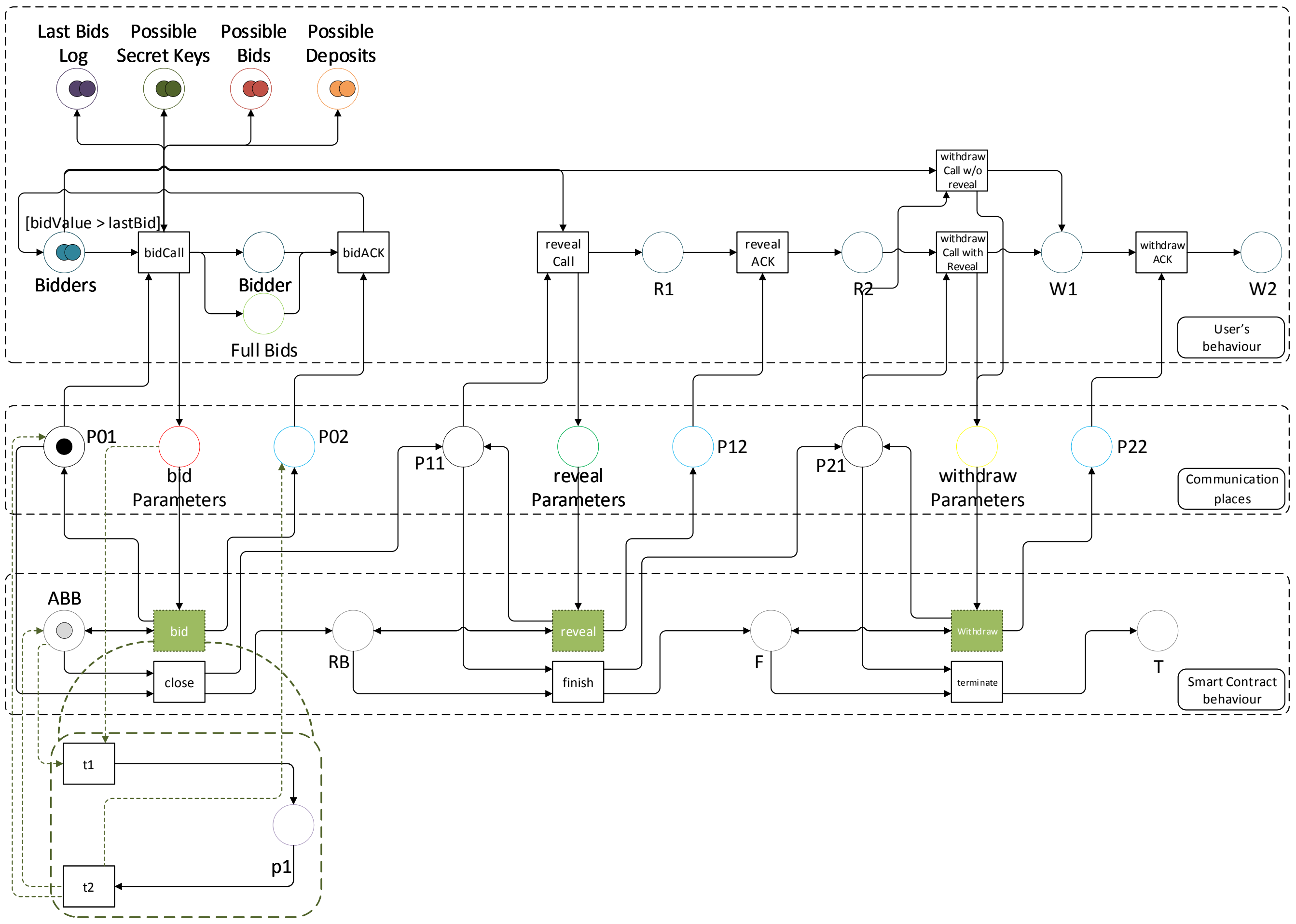


Figure 2. Blind Auction - Level-0 Model with Level-1 submodel for *bid*

5.1.3. Statements S**.** A statement $st \in \mathbb{S}$ can be either a compound statement of the form $\{st[1]; st[2]; \ldots; st[N]\}$, where $st[i]$ is a statement, or a simple statement of the form (st_{LHS}, st_{RHS}), or a control statement. A simple statement can be:

- an assignment statement, where we can access:
  - the assigned variable $st_{LHS}.variable$
  - the set of variables used in the assignment expression $st_{RHS}.variables$ (can be empty)
- a variable declaration statement, where we can access:
  - the declared variable $st_{LHS}.variable$
  - the type of the variable $st_{LHS}.type$
  - the set of variables used in the variable initialization $st_{RHS}.variables$ (can be empty if no initialization is given)
- a sending statement, where we can access:
  - the destination account $st_{LHS}.variable$
  - the set of variables used in the expression of the value sent $st_{RHS}.variables$ (can be empty)
- a returning statement (st_{LHS} is empty), where we can access:
  - the set of variables used in the expression of the returned value $st_{RHS}.variables$ (can be empty)

A control statement can be:

- a requirement statement of the form $require(c)$, where c is a boolean expression in which we can access the set of used variables $c.variables$
- a selection statement which can have:
  - a single-branching form: $if(c)\ then\ st_T$, in which we can access the set of variables used in the condition expression $c.variables$
  - a double-branching form: $if(c)\ then\ st_T\ else\ st_F$, in which we can access the set of variables used in the condition expression $c.variables$

- a looping statement which can be:
  - a for loop: $for(init; c; inc)\ st_T$ in which we can access the set of variables used in the condition expression $c.variables$
  - a while loop: $while(c)\ st_T$ in which we can access the set of variables used in the condition expression $c.variables$

5.2. Transformation Algorithm

The goal of our proposed algorithm is to generate a hierarchical CPN model that extends the level-0 model by level-1 submodels that correspond to the functions of the smart contract in question. To do so, EXTENDMODEL is applied on the level-0 model.

5.2.1. EXTENDMODEL.

```
 1: procedure EXTENDMODEL(P_CPN=(P,T,A,Σ,V,C,G,E,I))
 2:     Input: level-0 CPN model
 3:     Output: hierarchical level-0 CPN model extended with level-1 submodels
 4:     for transition $t \in T$ do
 5:         $metaColour \leftarrow [\bullet t[cf].colour, \bullet t[input].colour]$
 6:         $P_{data} \leftarrow \emptyset$
 7:         $getLocalVariables(t^{action}, P_{data})$
 8:         $insertSubModel(t, P_{CPN})$
 9:     end for
10: end procedure
```

Extending the initial model with the submodel of a transition t comprises 3 main steps:

- defining a colour $metaColour$ that is the concatenation of the colour of the state places in the global model with that of the input parameters place of transition t *(line 5)*
- creating the set of places corresponding to local variables in the function of transition t *(lines 6 and 7)*
- inserting the submodel of transition t into the level-0 model model *(line 8)*

5.2.2. GETLOCALVARIABLES.

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

GETLOCALVARIABLES creates a set of places to be used in the submodel of a transition t, 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} *(lines 5 and 6)*. 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}.

5.2.3. INSERTSUBMODEL.

```
 1: procedure INSERTSUBMODEL(t, P_CPN)
 2:     Input: transition $t$ and CPN model P_CPN
 3:     Output: CPN model embedded with the level-1 submodel for $t$
 4:     hide transition $t$
 5:     subModel = CREATESUBMODEL($t; t^{action}; \bullet t[cf]; t\bullet[cf]; P_{data}; metaColour$)
 6:     for $t' \in subModel.T$ do
 7:         if $\bullet t'[cf] = \bullet t[cf]$ then
 8:             create arc from $\bullet t[input]$ to $t'$
 9:         end if
10:         if $t' \bullet [cf] = t \bullet [cf]$ then
11:             create arc from $t'$ to $t \bullet [com]$
12:             if $t'^{action}$ is not a returning statement then
13:                 create arc from $t'$ to $t \bullet [output]$ (with inscription [inInsc.sender,inInsc.balance]
14:             end if
15:         end if
16:     end for
17: end procedure
```

Replacing a transition t by its corresponding submodel comprises 3 main steps:

- hiding t *(line 4)*
- creating the submodel for t *(line 5)*
- connecting the created submodel to the places connected to t in the level-0 model *(lines 6-16)* by:
    - connecting the input parameters place of transition t to the input transitions of its submodel *(lines 7-9)*
    - connecting the output transitions of the submodel of t to its output communication place and its output return place in case t^{action} does not include a returning statement *(lines 10-15)*. We note that otherwise the connection to the output return place is established as part of the execution of CREATESUBMODEL.

5.2.4. CREATESUBMODEL.

```
1:  procedure CREATESUBMODEL(t; st; p_in; p_out; P_data; metaColour)
2:      Input: transition $t$, statement $st$, control flow input place $p_{in}$, control flow output place $p_{out}$, set of internal data places $P_{data}$, meta colour $metaColour$
3:      Output: submodel of transition $t$
4:      set default place colour to $metaColour$
5:      if $st$ is a compound statement $\{st[1]; st[2]; \ldots; st[N]\}$ then
6:          for $i = 1..N-1$ do
7:              create place $p_i$
8:          end for
9:          CREATESUBMODEL($t;st[1];p_{in};p_1;P_{data};metaColour$)
10:         for $i = 2..N-1$ do
11:             CREATESUBMODEL($t;st[i];p_{i-1};p_i;P_{data};metaColour$)
12:         end for
13:         CREATESUBMODEL($t;st[N];p_{N-1};p_{out};P_{data};metaColour$)
14:     else if $st$ is a simple statement then
15:         if $st$ is an assignment statement then
16:             create transition $t'$
17:             create arc from $p_{in}$ to $t'$
18:             for $v \in st_{RHS}.variables \backslash \{st_{LHS}.variable\}$ do
19:                 create arc from $P_{data}[v]$ to $t'$
20:                 create arc from $t'$ to $P_{data}[v]$
21:             end for
22:             if $st_{LHS}.variable$ is a local variable then
23:                 create arc from $P_{data}[st_{LHS}.variable]$ to $t'$
24:                 create arc from $t'$ to $P_{data}[st_{LHS}.variable]$ with inscription $st_{RHS}$
25:                 create arc from $t'$ to $p_{out}$
26:             else
27:                 create arc from $t'$ to $p_{out}$ with inscription $outInsc \leftarrow inInsc$ in which the variable corresponding to $st_{LHS}.variable$ is replaced by $st_{RHS}$
28:             end if
29:         else if $st$ is a variable declaration statement then
30:             create transition $t'$
31:             create arc from $p_{in}$ to $t'$
32:             for $v \in st_{RHS}.variables$ do
33:                 create arc from $P_{data}[v]$ to $t'$
34:                 create arc from $t'$ to $P_{data}[v]$
35:             end for
36:             create arc from $t'$ to $P_{data}[st_{LHS}.variable]$ with inscription $st_{RHS}$
37:             create arc from $t'$ to $p_{out}$
38:         else if $st$ is a sending statement then
39:             create transition $t'$
40:             create arc from $p_{in}$ to $t'$
41:             for $v \in st_{RHS}.variables$ do
42:                 create arc from $P_{data}[v]$ to $t'$
43:                 create arc from $t'$ to $P_{data}[v]$
44:             end for
45:             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 $st_{RHS}$
46:         else if $st$ is a returning statement then
47:             create transition $t'$
48:             create arc from $p_{in}$ to $t'$
49:             for $v \in st_{RHS}.variables$ do
50:                 create arc from $P_{data}[v]$ to $t'$
51:                 create arc from $t'$ to $P_{data}[v]$
52:             end for
53:             create arc from $t'$ to $t\bullet[cf]$
54:             create arc from $t'$ to $t\bullet[output]$ with inscription $outInsc \leftarrow [inInsc.sender, inInsc.balance, st_{RHS}]$
55:         end if
56:     else if $st$ is a control statement then
57:         if $st$ is a requirement statement: $require(c)$ then
58:             create transition $t_{revert}$ with guard $!c$
59:             create arc from $p_{in}$ to $t_{revert}$
60:             create arc from $t_{revert}$ to $\bullet t[cf]$
61:             for $v \in c.variables$ do
62:                 create arc from $P_{data}[v]$ to $t_{revert}$
63:                 create arc from $t_{revert}$ to $P_{data}[v]$
64:             end for
65:             create transition $t_{!revert}$ with guard $c$
66:             create arc from $p_{in}$ to $t_{!revert}$
67:             create arc from $t_{!revert}$ to $p_{out}$
68:             for $v \in c.variables$ do
69:                 create arc from $P_{data}[v]$ to $t_{!revert}$
70:                 create arc from $t_{!revert}$ to $P_{data}[v]$
71:             end for
72:         else if $st$ is a selection statement then
73:             create place $p_T$
74:             create transition $t_T$ with guard $c$
75:             create arc from $p_{in}$ to $t_T$
76:             create arc from $t_T$ to $p_T$
```

```
77:        for $v \in c.variables$ do
78:          create arc from $P_{data}[v]$ to $t_T$
79:          create arc from $t_T$ to $P_{data}[v]$
80:        end for
81:        CREATESUBMODEL($t;st_T;p_T;p_{out};P_{data};metaColour$)
82:        create transition $t_F$ with guard $!c$
83:        create arc from $p_{in}$ to $t_F$
84:        for $v \in c.variables$ do
85:          create arc from $P_{data}[v]$ to $t_F$
86:          create arc from $t_F$ to $P_{data}[v]$
87:        end for
88:        if $st$ is a selection statement: $if(c)\ then\ st_T$ then
89:          create arc from $t_F$ to $p_{out}$
90:        else if $st$ is a selection statement: $if(c)\ then\ st_T\ else\ st_F$ then
91:          create place $p_F$
92:          create arc from $t_F$ to $p_F$
93:          CREATESUBMODEL($t;st_F;p_F;p_{out};P_{data};metaColour$)
94:        end if
95:      else if $st$ is a looping statement then
96:        if $st$ is a for statement: $for(init;c;inc)st_T$ then
97:          create place $p_{init}$
98:          create place $p_c$
99:          create place $p_T$
100:         CREATESUBMODEL($t;init;p_{in};p_{init};P_{data};metaColour$)
101:         create transition $t_T$ with guard $c$
102:         create arc from $p_{init}$ to $t_T$
103:         for $v \in c.variables$ do
104:           create arc from $P_{data}[v]$ to $t_T$
105:           create arc from $t_T$ to $P_{data}[v]$
106:         end for
107:         create arc from $t_T$ to $p_c$
108:         create transition $t_F$ with guard $!c$
109:         create arc from $p_{init}$ to $t_F$
110:         for $v \in c.variables$ do
111:           create arc from $P_{data}[v]$ to $t_F$
112:           create arc from $t_F$ to $P_{data}[v]$
113:         end for
114:         create arc from $t_F$ to $p_{out}$
115:         CREATESUBMODEL($t;st_T;p_c;p_T;P_{data};metaColour$)
116:         CREATESUBMODEL($t;inc;p_T;p_{init};P_{data};metaColour$)
117:       else if $st$ is a while loop statement: $while(c)\ st_T$ then
118:         create place $p_T$
119:         create transition $t_T$ with guard $c$
120:         create arc from $p_{in}$ to $t_T$
121:         for $v \in c.variables$ do
122:           create arc from $P_{data}[v]$ to $t_T$
123:           create arc from $t_T$ to $P_{data}[v]$
124:         end for
125:         create arc from $t_T$ to $p_T$
126:         create transition $t_F$ with guard $!c$
127:         create arc from $p_{in}$ to $t_F$
128:         for $v \in c.variables$ do
129:           create arc from $P_{data}[v]$ to $t_F$
130:           create arc from $t_F$ to $P_{data}[v]$
131:         end for
132:         create arc from $t_F$ to $p_{out}$
133:         CREATESUBMODEL($t;st_T;p_T;p_{in};P_{data};metaColour$)
134:       end if
135:     end if
136:   end if
137: end procedure
```

CREATESUBMODEL browses the body of the transition's corresponding function recursively, statement by statement, and creates snippets of a CPN model according to the type of the processed statement (cf. figures 3-11) that interconnect to create the transition's submodel. In the following we give the corresponding CPN patterns corresponding to each of the statement types:

- Compound statement $\{st[1]; st[2]; \ldots; st[N]\}$ *(lines 5-13)*: 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.

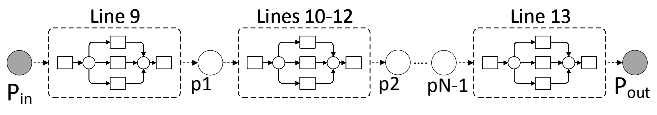
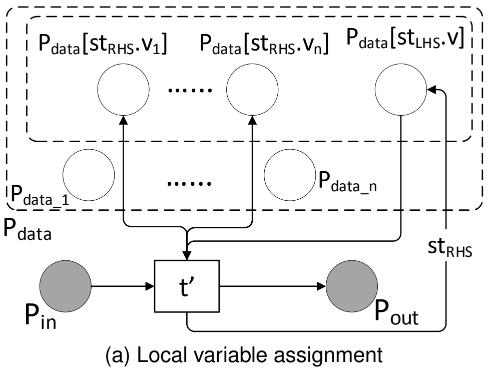


Figure 3. Compound Statement Pattern

- Assignment statement (st_{LHS}, st_{RHS}) *(lines 15-28)*: to represent such a statement, a transition t' is created with input and output links to, respectively, the input (p_{in}) and output (p_{out}) places passed as parameters. t' is also connected to the places in P_{data} that correspond to the variables used in the statement's RHS with input/output links (to read the data). In case of a local variable assignment (Figure 4 (a)), an input/output link is created with the place corresponding to the assigned variable in the statement's LHS with the new value (st_{RHS}) inscribed on the output link. In case of a state variable assignment (Figure 4 (b)), the new value (st_{RHS}) is given in the variable's corresponding placement in the inscription on the link to the output (p_{out}) place.



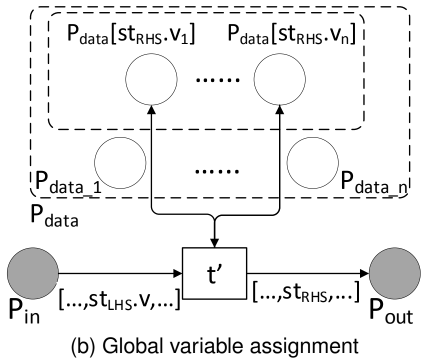


Figure 4. Assignment statement pattern

- Variable Declaration statement (st_{LHS}, st_{RHS}) *(lines 29-37)*: the algorithm creates a transition t' with input and output links to, respectively, the input (p_{in}) and output (p_{out}) places passed as parameters. Input/output links are created from and to the places corresponding to the variables used in the statement's RHS. An output link is also created to the place representing the declared variable with st_{RHS} as inscription.

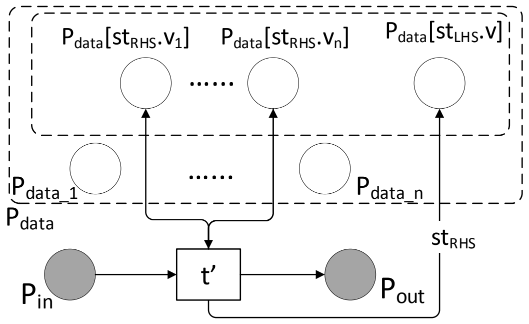


Figure 5. Variable Declaration Statement Pattern

- Sending statement (st_{LHS}, st_{RHS}) *(lines 38-45)*: the CPN snippet for a sending statement is generated in a way that is similar to that of the case of a global variable assignment, except that instead of updating the assigned variable's counterpart on the output link to p_{out} the algorithm updates the $contractBalance$ and $balance$ values from the input of t' by decreasing the first and increasing the second by st_{RHS}.

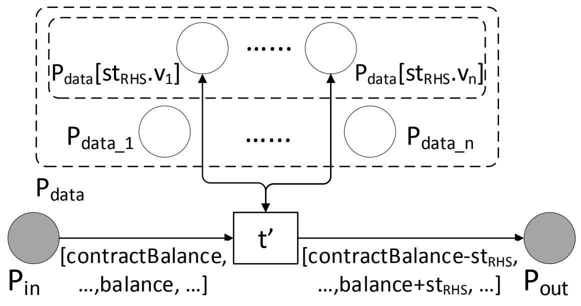

P_data[st_RHS.v_1]
......
P_data[st_RHS.v_n]
P_data_1
P_data_n
P_data
t'
P_in
[contractBalance, ...,balance, ...]
[contractBalance-st_RHS, ...,balance+st_RHS, ...]
P_out

Figure 6. Sending Statement Pattern

- Returning statement $(-, st_{RHS})$ *(lines 46-55)*: this is also treated similarly to a global variable assignment statement, but with the following differences. t' is linked to $t \bullet [cf]$ instead of p_{out} with the same inscription of its input link with p_{in}, and has an additional output link with the output return place $t \bullet [output]$ with the return value set to st_{RHS} in its inscription, with the same sender's address and balance from its input inscription.

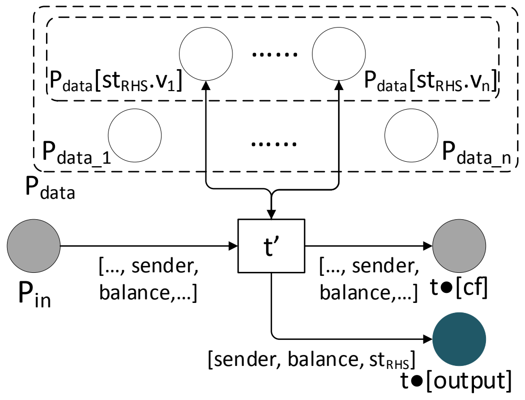


Figure 7. Returning Statement Pattern

- Requirement statement $require(c)$ *(lines 57-71)*: such a statement is transformed by creating two transitions t_{revert} and $t_{!revert}$, with $!c$ and c as respective guards, input links with p_{in} and input/output links with the places representing local variables used in c. Transition t_{revert} has $\bullet t[cf]$ as output place whereas p_{out} is the output place of $t_{!revert}$. We note that, as a requirement statement is often placed at the beginning of the function, p_{in} and $\bullet t[cf]$ are usually the same.

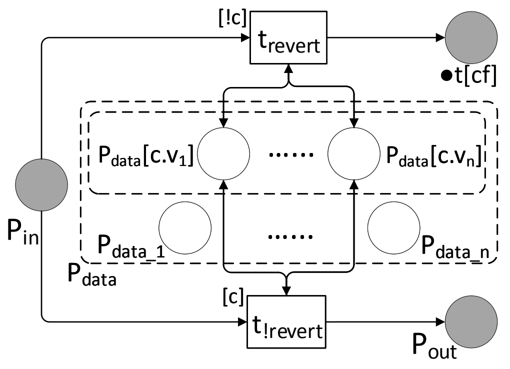
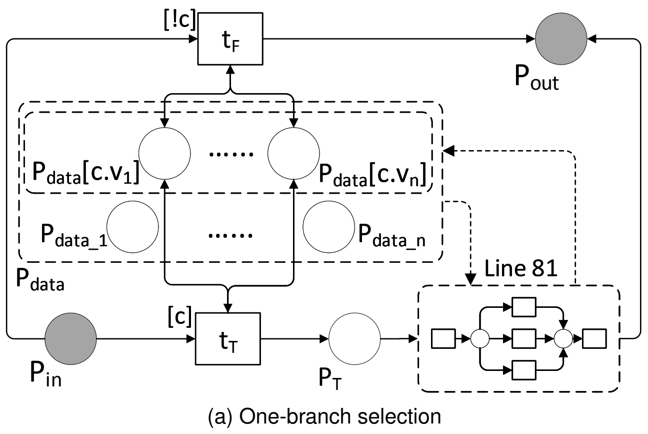


Figure 8. Requirement Statement Pattern



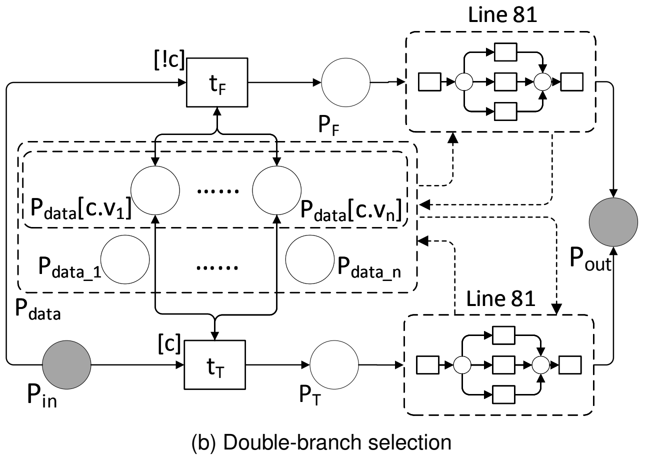


Figure 9. Selection Statement Pattern

- Selection statement $if(c)\ then\ st_T\ [else\ st_F]$ *(lines 72-94)*: the algorithm creates two transitions, t_T with guard c and t_F with guard $!c$, that respectively represent the activation of the true and false (or default) branches of the selection statement. Both transition are linked to the input place p_{in} and any places representing local variables used in c. CREATESUBMODEL is then recursively called for the true-branch statement. In case of a one-branch selection, t_F is linked directly to the output place p_{out}. In case of a double-branch selection, a new place p_F is created to be the output place of t_F and the input place in a recursive call for CREATESUBMODEL on the false-branch statement.
- For Looping statement $for(init; c; inc)\ st_T$ *(lines 96-116)*: three places, p_{init}, p_c and p_T, are created to respectively represent initialization, increments and one pass of the loop's body. Transitions t_T with guard c and t_F with guard $!c$ are created and linked to the input place p_{in} and any places representing local variables used in c. t_T is linked to p_c as output to trigger a counter increment while t_F is linked to p_out as output to leave the loop. CREATESUBMODEL is recursively called with three statements: $init$ which is usually a variable declaration statement with initialization, st_T to develop the CPN snippet for one run of the loop, and inc which is usually an assignment to a local variable type of statement.

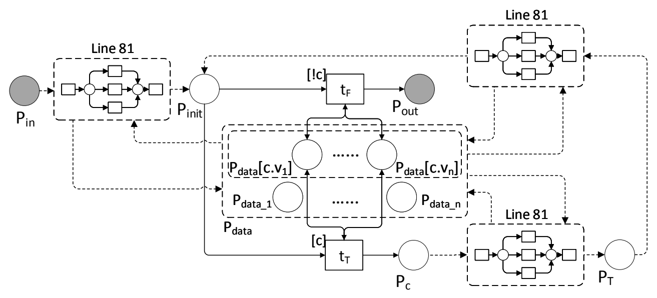


Figure 10. For Looping Statement Pattern

- While Looping statement $while(c)\ st_T$ *(lines 117-134)*: for this loop the algorithm proceeds by creating one new place p_T which will be the output for a new transition t_T with guard c. Another transition t_F is also created with $!c$ as guard and p_{out} as output place. Both transition are linked to the input place p_{in} and any places representing local variables used in c. A recursive call to CREATESUBMODEL on st_T with p_T for input and p_{in} for output places is responsible for the generation of the loop's body corresponding CPN snippet.

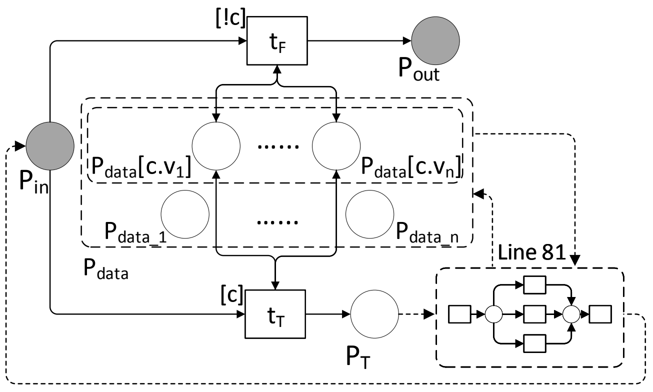


Figure 11. While Looping Statement Pattern

5.3. Application on the Use Case

The application of the algorithm on the level-0 model of the Blind Auction use case (Figure 2) presented in Section 4 yields a hierarchical CPN model whose level-1 submodels are shown in Figures 12-14. These submodels correspond to the transitions representing functions in the Blind Auction smart contract (Listing 1), namely *bid*, *reveal* and *withdraw*. For each submodel, the light-grey-coloured places represent control flow places P_{cf} of a *metaColour* specific to the submodel in question, whereas places of other colours inside the dashed-line box are places of the relative P_{data}. Places outside the box are the input/output places of the corresponding level-0-transition, among which the dark grey places are state places in P_S, uncoloured places are interface places in P_I, dark blue places are return places in P_R and the others (red, green and blue) are parameter places in P_P. The submodels inside the dashed-line boxes are the product of CREATESUBMODEL, and the links with the input/output places of level 0 are generated by INSERTSUBMODEL.

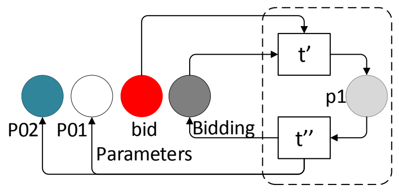


Figure 12. SubModel of transition *bid*

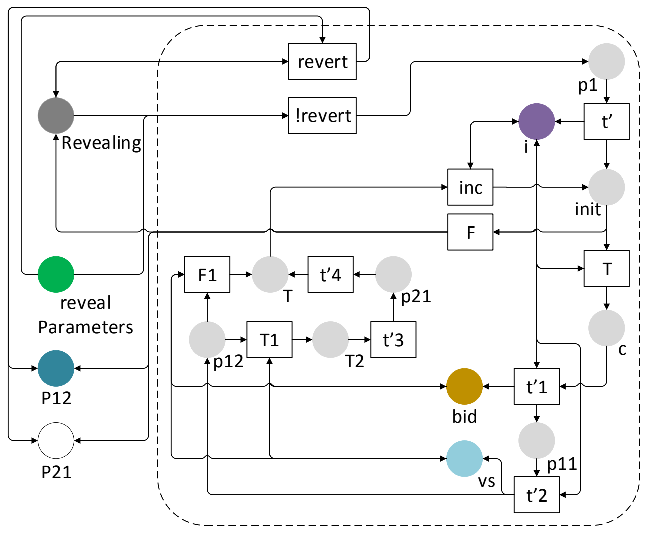


Figure 13. SubModel of transition *reveal*

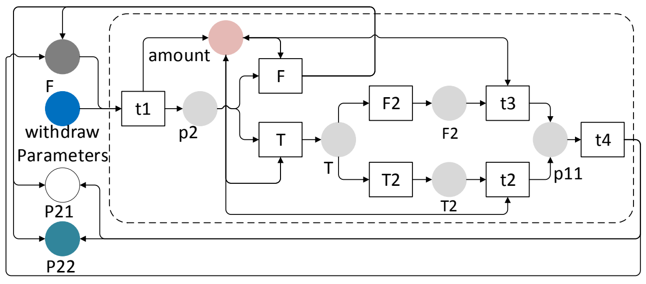


Figure 14. SubModel of transition *withdraw*

6. Smart Contract Verification via CPN Tools

Having established the CPN model for a smart contract, verifying properties of the smart contract would come down to verifying properties on the corresponding CPN model. *CPN Tools* is a tool that, in addition to its editing capabilities, also offers the possibility to analyze the state space of a CPN model through its *State Space* palette as well as its associated *ASK-CTL* library [25]. The latter uses a CTL-like temporal logic interpretation over the state space of the model and leverages explicit model checking techniques for the verification of behavioural properties. We have implemented the CPN model for our Blind Auction use case using CPN Tools and investigated its potential in the verification of behavioural and contract-specific properties.

6.1. State Space Analysis Results

In the following, we present state space analysis statistics for different initial marking values. Table 1 shows the evolution of state space and strongly-connected component generation times, as well as the number of nodes, arcs and dead markings while varying the number of bidders, possible bids, possible secret keys and possible deposits in the system. We note that for 5 bidders (while fixing the other variables at 1), as well as for 1 bidder (while fixing the other variables at 4), the state space generation had not finished after several hours of execution. This is due to the infamous state space explosion problem associated with explicit state space exploration.

TABLE 1. STATE SPACE ANALYSIS RESULTS FOR DIFFERENT INITIAL MARKINGS

Bidders	1	2	3	4	5	1	1	1
PB	1	1	1	1	1	2	3	4
PSK	1	1	1	1	1	2	3	4
PD	1	1	1	1	1	2	3	4
SS gen. time	4s	4s	6s	252s	-	4s	30s	
	5s	5s	10s	1001s	-			
SCC gen. time	0s	0s	0s	2s		0s	1s	
	0s	0s	1s	34s				
Nodes	24	235	3118	47621		484	19984	
	44	583	9166	156117				
Arcs	26	378	7106	145062		555	22980	
	46	900	19784	446326				
Dead mark.	3	10	35	124		49	1999	
	3	10	35	124				

6.2. State Space and ASK-CTL for Verification

State space report allows the deduction of some general behavioural properties... boundedness, deadlock freedom...

More specific properties can be verified by elaborating conceiving) CTL properties... termination...

- Termination: starting from the initial marking, it is always possible to reach a marking the place T is not empty
  $fun\ notEmptyT\ n = size(Mark.Complete\_System'T\ 1\ n) <> 0;$
  $val\ terminationFormula = INV(EV(NF("termination", notEmptyT)));$
  $eval\_node\ terminationFormula\ InitNode;$
- Termination 2: the place T is not empty in every dead marking (final marking)
  $fun\ notEmptyT\ n = size(Mark.Complete\_System'T\ 1\ n) <> 0;$
  $fun\ notEmptyAndDeadT\ n = if\ DeadMarking\ n\ then\ notEmptyT\ n\ else\ true;$
  $val\ terminationFormula2 = INV(EV(NF("termination", notEmptyAndDeadT)));$
  $eval\_node\ terminationFormula2\ InitNode;$

7. Conclusion

Being an important pillar for Blockchain technology, smart contracts need to provide certain guarantees in terms of correctness for them to support its foundation built on trust. Formal approaches for the verification of Solidity smart contracts have been proposed, but they are generally designed to target specific vulnerabilities known in the literature. The need to verify contract-specific properties has proven increasingly necessary in the light of the expanding reach of smart contracts in many application fields. In an effort to bring a solution to this problem, we propose a transformation algorithm that generates a hierarchical CPN model representing a given Solidity smart contract, including both its control-flow as well as data aspects. CTL properties are then verified on the CPN model to check corresponding properties on the smart contract, unrestrictedly to certain predefined vulnerabilities.
In view of the results presented in this paper, it may be concluded that CPN Tools does not hold much potential for the verification of properties on CPN models of smart contracts due to the state space explosion problem. We do prove, however, that the idea of using CPNs as a representation formalism is promising for it allows the consideration of the data aspect, and thus the formulation of contract-specific properties. To overcome the encountered limitations, we intend to investigate the potential of Helena [26] as an analyzer for High Level Nets. This tool offers on-the-fly verification of LTL properties, which unlike the verification of CTL properties offered by CPN Tools, does no always require the generation of the whole state space. To further improve the tool's performance, we also 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 Petri nets [27] and applying it on CPNs.

References

[1] S. Nakamoto and A. Bitcoin, "A peer-to-peer electronic cash system," *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 2008.

[2] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[3] "Solidity — solidity 0.6.11 documentation," https://solidity.readthedocs.io/en/v0.6.11/index.html.

[4] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*.

[5] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[6] K. Jensen, "Coloured petri nets: A high level language for system design and analysis," in *International Conference on Application and Theory of Petri Nets*. Springer, 1989, pp. 342–416.

[7] K. Van Hee and P. Verkoulen, "Integration of a data model and high-level petri nets," in *Proceedings of the 12th International Conference on Applications and Theory of Petri Nets, Gjern*, 1991, pp. 410–431.

[8] R. Milner, M. Tofte, and R. Harper, *Definition of standard ML*. MIT Press, 1990.

[9] J. D. Ullman, *Elements of ML programming - ML 97 edition*. Prentice Hall, 1998.

[10] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Z. Béguelin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, 2016, pp. 91–96.

[11] "Formal verification for solidity contracts — ethereum community forum," https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts, October 2015.

[12] Y. Hirai, "Ethereum virtual machine for coq (v0.0.2)," https://medium.com/@pirapira/ethereum-virtual-machine-for-coq-v0-0-2-d2568e068b18, March 2017.

[13] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 66–77.

[14] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, 2003, pp. 553–568.

[15] S. Anand, C. S. Pasareanu, and W. Visser, "Symbolic execution with abstraction," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 1, pp. 53–67, 2009.

[16] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 254–269.

[17] T. Chen, X. Li, X. Luo, and X. Zhang, "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*, 2017, pp. 442–446.

[18] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, 2018, pp. 653–663.

[19] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *9th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2018, Paris, France, February 26-28, 2018*, 2018, pp. 1–5.

[20] C. F. Torres, J. Schütte, and R. State, "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*, 2018, pp. 664–676.

[21] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "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.

[22] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "Verisolid: Correct-by-design smart contracts for ethereum," in *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, 2019, pp. 446–465.

[23] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," in *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, 2018, pp. 523–540.

[24] "Solidity by example — solidity 0.6.12 documentation," https://solidity.readthedocs.io/en/develop/solidity-by-example.html#blind-auction.

[25] A. Cheng, S. Christensen, and K. H. Mortensen, "Model checking coloured petri nets-exploiting strongly connected components," *DAIMI report series*, no. 519, 1997.

[26] S. Evangelista, "High level petri nets analysis with helena," in *Applications and Theory of Petri Nets 2005*, G. Ciardo and P. Darondeau, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 455–464.

[27] K. Klai and D. Poitrenaud, "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*, 2008, pp. 288–306. [Online]. Available: https://doi.org/10.1007/978-3-540-68746-7_20