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

This document contains complementary content for the paper "A Solidity-to-CPN Approach Towards Formal Verification of Smart Contracts" in which we propose an algorithm for the transformation of a smart contract written in Solidity into a hierarchical Coloured Petri Net model over which CTL properties can be verified. In section 1, we remind the reader of the use case presented in the paper. Section 2 starts by a reminder of the different elements of the CPN model to be generated then presents the full algorithm with detailed descriptions of its different parts followed by an example of its application on the use case. ....

1 Use Case: Blind Auction

The use case presented in this section is adapted from an example in [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 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 1.1 represents the Solidity smart contract implementing it.

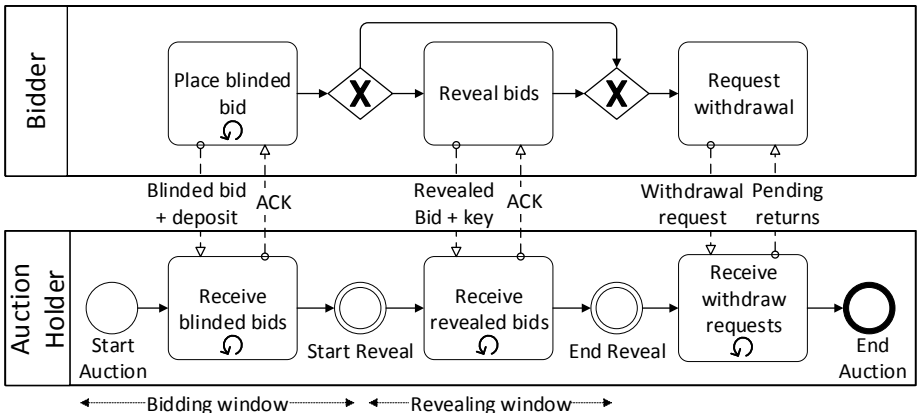


Fig. 1: Blind Auction Workflow

```
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 1.1: Blind Auction smart contract in Solidity

2 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 3). 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. The submodels corresponding to the transitions represented in green will be detailed in Section **??**.

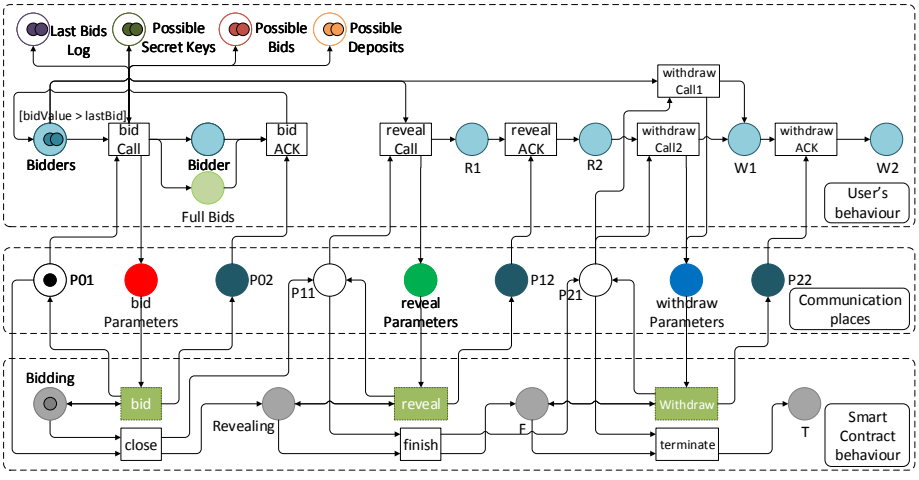


Fig. 2: Blind Auction - Level-0 Model

We lead off by setting some notations on the elements of the model in Section 2.1 before getting into the algorithm in Section 2.2 and applying it on the use case in Section **??**.

2.1 Notations on the Model's Elements

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. 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 corresponding to the represented variable's type.
- For a place $p \in P$, we can access its identifier ($p.name$) and its colour ($p.colour$)

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
- $t^{action} \in S$: action of t (the function's body)

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

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

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: $if(c)\ then\ st_T$
- 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 in which we can access the set of used variables $c.variables$.

2.2 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.

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 sub-models
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 var-iables in the function of transition t *(lines 6 and 7)*
- inserting the submodel of transition t into the level-0 model model *(line 8)*

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}.

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.

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};meta-Colour)*
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 ∈ 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 ∈ 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 ∈ 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 ∈ 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.

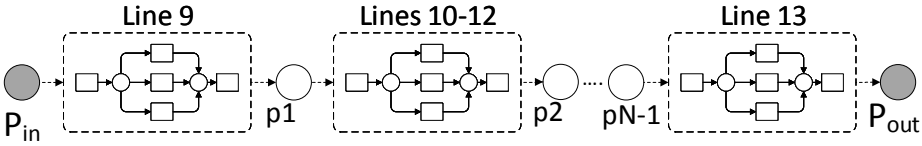
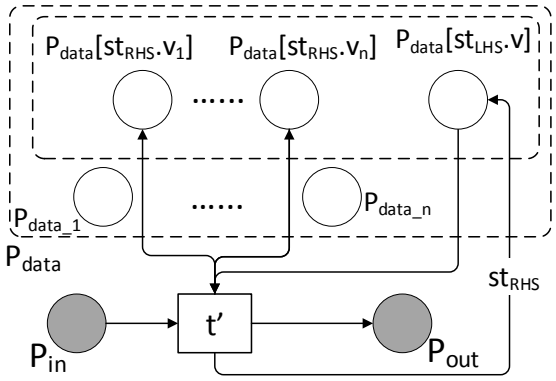


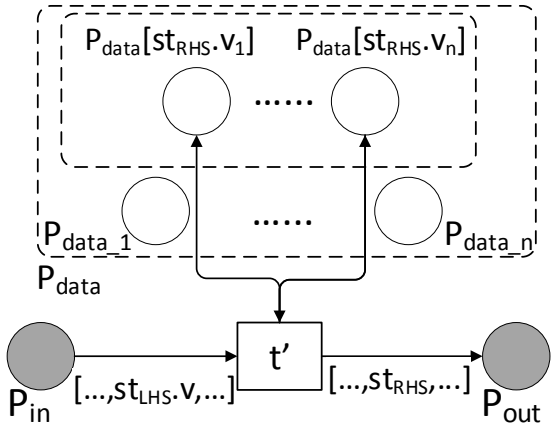
Fig. 3: Compound Statement Pattern

– 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.



(a) Local variable assignment



(b) Global variable assignment

Fig. 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.

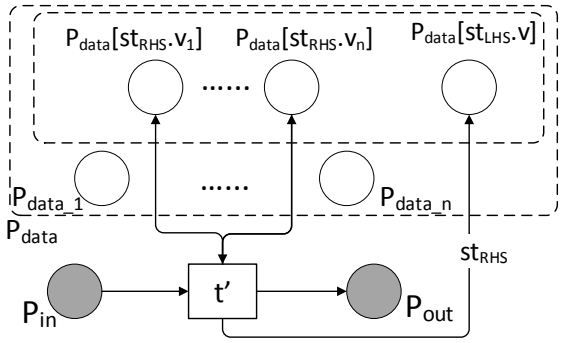


Fig. 5: Variable Declaration Statement Pattern

– 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}.

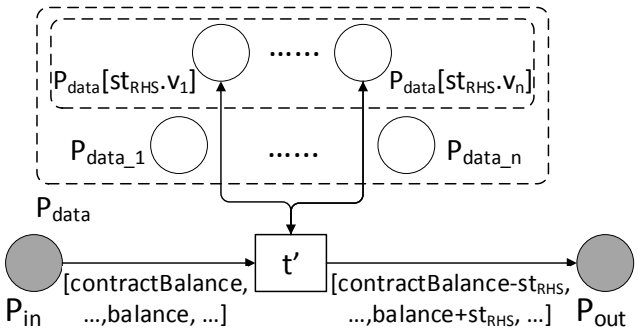


Fig. 6: Sending Statement Pattern

– 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.

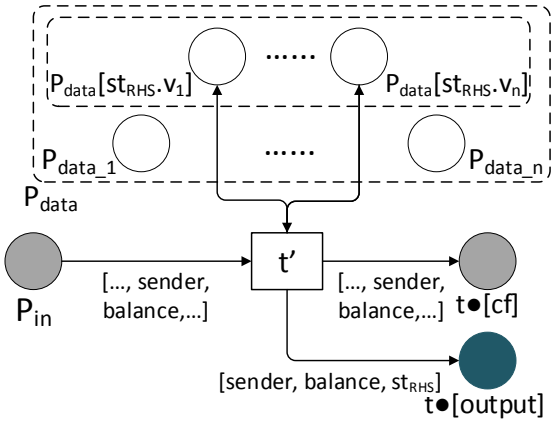


Fig. 7: Returning Statement Pattern

– 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.

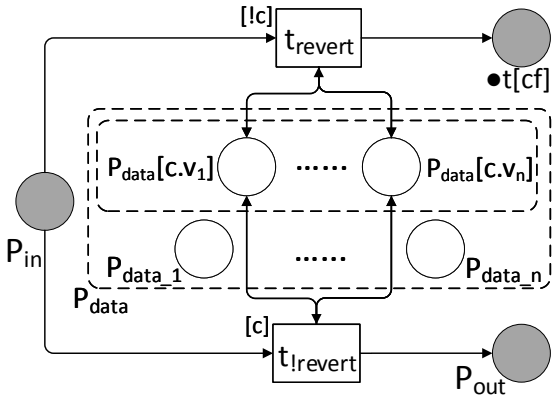
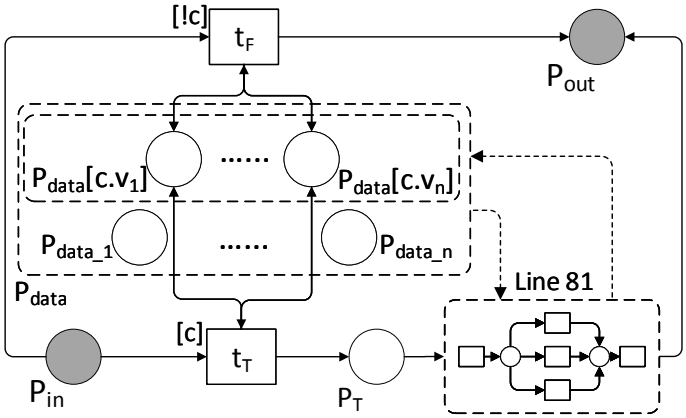
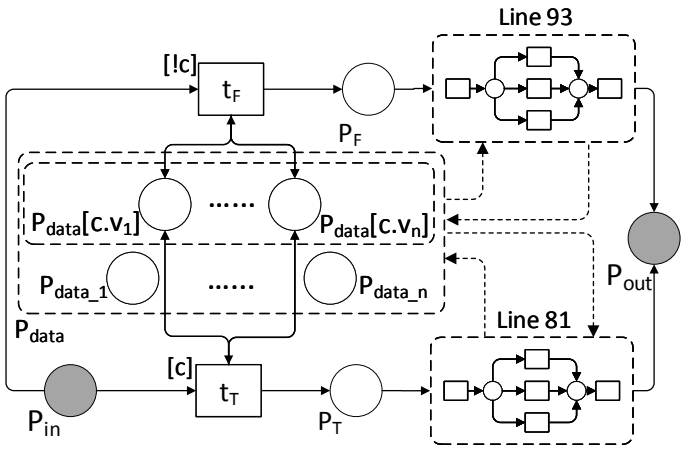


Fig. 8: Requirement Statement Pattern

– 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.



(a) One-branch selection



(b) Double-branch selection

Fig. 9: Selection Statement Pattern

– 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.

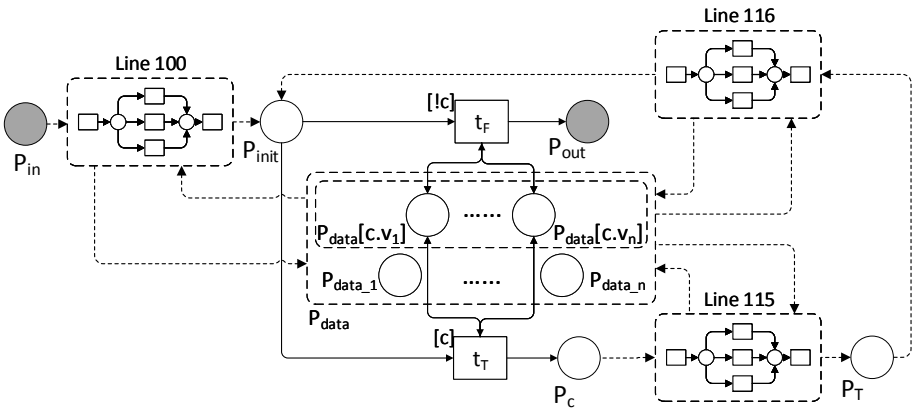


Fig. 10: For Looping Statement Pattern

– 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.

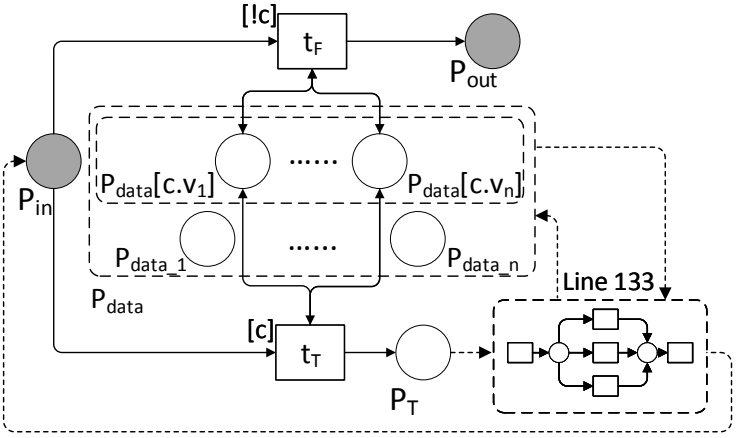


Fig. 11: While Looping Statement Pattern

2.3 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 1 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 1.1), 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.

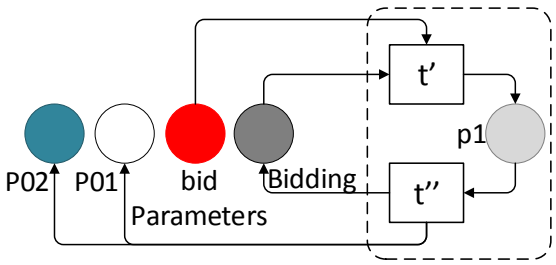


Fig. 12: SubModel of transition *bid*

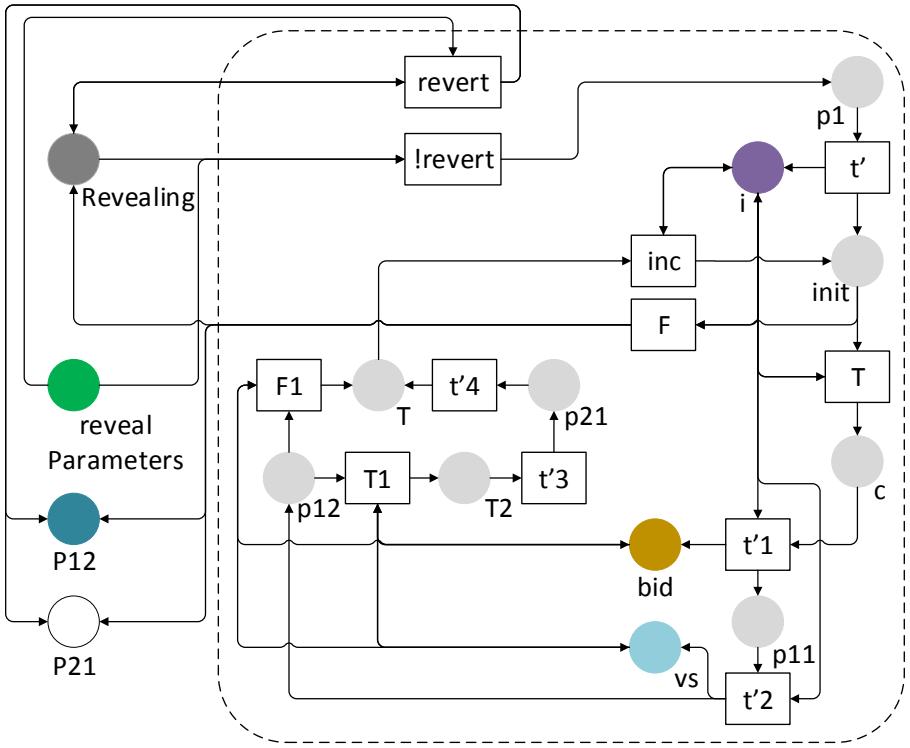


Fig. 13: SubModel of transition *reveal*

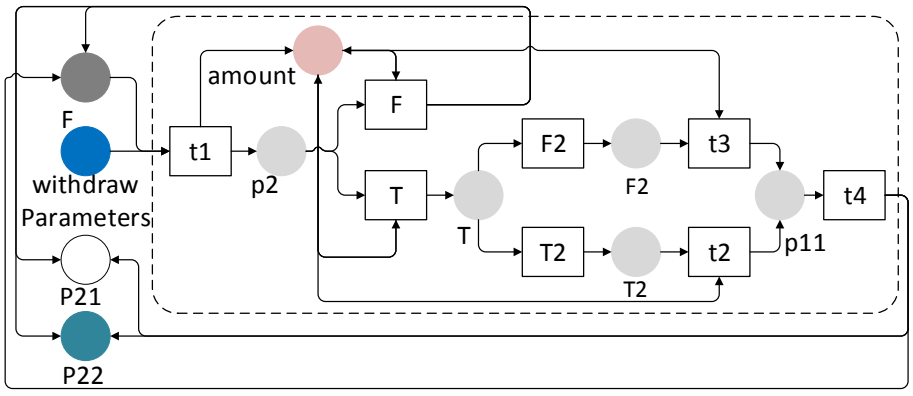


Fig. 14: SubModel of transition *withdraw*

3 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 [2]. 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.

3.1 State Space Analysis Results

In the following, we present state space analysis statistics for different initial marking values. Table 1 shows the evolution of the state space generation time, 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. Every line is subdivided into two sub-lines, the upper one representing the analysis results for the level-0 model (i.e., without hierarchy), and the bottom one representing the model extended with the level-1 submodels (i.e., with hierarchy). 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. In fact, as the number of state variables in the system increases, the size of the system state space grows exponentially. The last line of the table indicates the number of the state space report generated in each tested case. The generated reports can be found in this repository[1].

3.2 State Space and ASK-CTL for Verification

The state space report generated by CPN Tools allows the deduction of several general behavioural properties.

- Boundedness: The state space reports show that all of the places of our model are bounded. A place is bounded when the number of tokens it contains does not exceed some finite number for any marking reachable from the initial marking.
- Reversibility: In order to derive this property in our case, we modify the model as to obtain a short-circuited version. To do so, we added a transition that restores the model to its initial state once it reaches a final one. The state space report of the modified version indicates that all of the markings are home markings, hence the model is reversible.

[1] https://github.com/garfatta/solidity2cpn

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	-	5s	74s	-
Nodes	24	235	3118	47621	-	484	19984	-
	44	583	9166	156117	-	1424	65513	-
Arcs	26	378	7106	145062	-	555	22980	-
	46	900	19784	446326	-	1545	70704	-
Dead mark.	3	10	35	124	-	49	1999	-
	3	10	35	124	-	49	1999	-
Report Number	R01	R11	R21	R31	-	R41	R51	-
	R02	R12	R22	R32	-	R42	R52	-

– Liveness/Deadlock-freedom: The concept of liveness is closely related to the complete absence of deadlocks in operating systems. In fact, a live Petri net guarantees a deadlock-free operation, regardless of the firing sequence. The resulting state space analysis report for our modified model (the short-circuited version) indicates that all of our model's transitions are live, which means that every transition remains fireable for any firing sequence.

More specific properties can be verified by elaborating CTL properties. For instance, we can formulate a *termination* property to check that all the dead markings in the state space actually correspond to final markings of the model, and therefore checks that the model is deadlock-free. To do so, we give a characterization of the *termination* property as the model's capability to always reach a terminal state (a dead marking) where the following conditions are met:

– (C1) in the Smart Contract's Behaviour part, all places have to be empty except for the place T that must contain one token

```
fun C1  n = size (Mark.System'T 1 n) = 1
    andalso size (Mark.System'Bidding 1 n) = 0
    andalso size (Mark.System'Revealing 1 n) = 0
    andalso size (Mark.System'F 1 n) = 0
```

– (C2) all Communication places must be empty

```
fun C2  n = size (Mark.System'P01 1 n) = 0
    andalso size (Mark.System'P11 1 n) = 0
    andalso size (Mark.System'P21 1 n) = 0
    andalso size (Mark.System'bidParameters 1 n) = 0
    andalso size (Mark.System'revealParameters 1 n) = 0
    andalso size (Mark.System'withdrawParameters 1 n) = 0
    andalso size (Mark.System'P02 1 n) = 0
    andalso size (Mark.System'P12 1 n) = 0
    andalso size (Mark.System'P22 1 n) = 0
```

– (C3) in the User's Behaviour part:
  - (C31) the markings of places *PossibleSecretKeys*, *PossibleBids* and *PossibleDeposits* must be the same as their initial markings

```
fun C31 n = Mark.System'PossibleSecretKeys 1 n = Mark.
    System'PossibleSecretKeys 1 InitNode
    andalso Mark.System'PossibleBids 1 n = Mark.System'
    PossibleBids 1 InitNode
    andalso Mark.System'PossibleDeposits 1 n = Mark.
    System'PossibleDeposits 1 InitNode
```

  - (C32) tokens in $W2$ must correspond to the users who have actually placed bids

```
(* biddersWithBids: returns the list of tokens in
    LastBidsLog with bidValue<>0 *)
fun biddersWithBids n = List.filter(fn {bidder,
    bidValue} => bidValue <> 0)
    (Mark.System'LastBidsLog 1 n)

(* sameNumberOfBidders: checks that the number of
    tokens in W2 is equal to the number of bidders with
     placed bids *)
fun sameNumberOfBidders n = size (biddersWithBids n)
    = size (Mark.System'W2 1 n)

(* bidderExists: checks that a token with a bidder's
    address (adr) exists in W2 *)
fun bidderExists adr n = List.find(fn {bidder, ...} =>
    adr = bidder) (Mark.System'W2 1 n) <> NONE

(* allBiddersExist: checks that all bidders in a list (
    l) exist in W2 *)
fun allBiddersExist l n = case l of {bidder,bidValue}::
    t =>
    bidderExists bidder n andalso allBiddersExist t n
    | [] => true

(* C32: checks that the tokens in W2 correspond to the
    users who have actually placed bids (i.e., bidders
    with a bidding value different from 0 in
    LastBidsLog*)
fun C32 n = sameNumberOfBidders n
    andalso allBiddersExist (biddersWithBids n) n
```

  - (C33) the union of the sets of users who have not placed bids and those who have must correspond to the set of initial bidders

```
(* getBiddersBidder: returns the list of bidder
    addresses in Bidders *)
fun getBiddersInBidders n = remdupl(List.map(fn e:
    FULL_BIDDER => #bidder e)(Mark.System'Bidders 1 n))
```

```
(* getBiddersInW2: returns the list of bidder addresses
     in W2 *)
fun getBiddersInW2 n = remdupl(List.map(fn e:
    FULL_BIDDER => #bidder e) (Mark.System'W2 1 n))

(* C33: checks that the union of users who have not
    placed bids and those who have corresponds to the
    set of initial bidders *)
fun C33 n = size(Mark.System'Bidders 1 n) + size(Mark.
    System'W2 1 n)=size(Mark.System'Bidders 1 InitNode)
     andalso contains (remdupl((getBiddersInBidders n)
     ^^(getBiddersInW2 n))) (remdupl(getBiddersInBidders
     InitNode)) andalso contains (remdupl(
     getBiddersInBidders InitNode)) (remdupl((
     getBiddersInBidders n)^^(getBiddersInW2 n)))
```

- (C34) every other place must be empty

```
fun C34 n = size (Mark.System'Bidders2 1 n) = 0
    andalso size (Mark.System'B1 1 n) = 0
    andalso size (Mark.System'R1 1 n) = 0
    andalso size (Mark.System'R2 1 n) = 0
    andalso size (Mark.System'W1 1 n) = 0
```

The code for the *termination* property would look like this:

```
fun Termination n = Terminal n andalso C1 n andalso C2 n
    andalso C31 n andalso C32 n andalso C33 n andalso C34 n
val terminationFormula = INV(EV(NF("termination", Termination
    )));
eval_node terminationFormula InitNode;
```

where $C1$, $C2$, $C31$, $C32$, $C33$ and $C34$ are functions that check the corresponding aforementioned conditions on a node n. The evaluation of this property on our model returns *true* which confirms that all dead markings in the state space do correspond to $final$ markings and that consequently the model contains no deadlocks.

4 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 we present, 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 [3] 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 [4] and applying it on CPNs.

References

1. Solidity 0.6.11 documentation. https://solidity.readthedocs.io/en/v0.6.11/index.html
2. Cheng, A., Christensen, S., Mortensen, K.H.: Model checking coloured petri nets-exploiting strongly connected components. DAIMI report series (519) (1997)
3. Evangelista, S.: High level petri nets analysis with helena. In: Ciardo, G., Darondeau, P. (eds.) Applications and Theory of Petri Nets 2005. pp. 455–464. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
4. 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). https://doi.org/10.1007/978-3-540-68746-7_20, https://doi.org/10.1007/978-3-540-68746-7_20