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

This document contains complementary content for the published papers on our “Solidity-to-CPN Approach Towards Formal Verification of Smart Contracts". In section 1, we detail the different use cases we used throughout the papers. Section 2 gives an overview of our proposed approach, followed by details on the generated HCPN model considered for the verification and the algorithm used for its generation in Section 3. Explanation of how the model is used for the model checking of smart contracts is developed in Section 4 and a conclusion seals the document in Section 5.

1 Use Cases

Three use cases will be detailed in this section. The first one is the Blind Auction use case presented in Section 1.1, through which we focus on the design of the smart contract as a business process. The second and third use cases presented in Section 1.2 are two gambling games that we use to explain different vulnerabilities that smart contracts can have.

1.1 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. We propose a design for such a blind auction system using a BPMN choreography diagram (Figure 1) as well as a DCR graph (Figure 2). Listing 1.1 represents the Solidity smart contract implementing it.

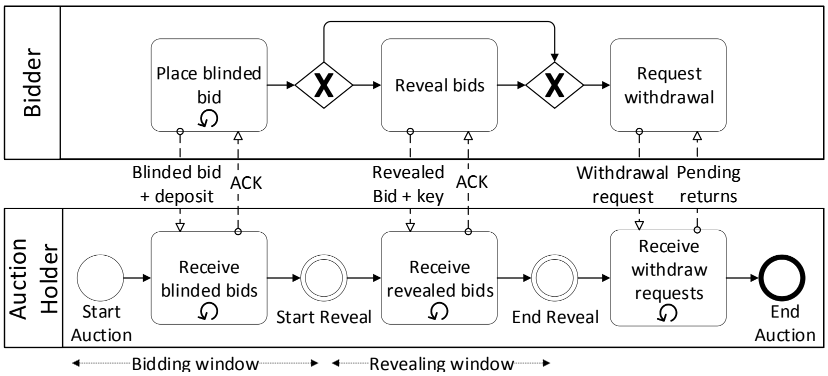


Fig. 1. Blind Auction Workflow as a BPMN choreography

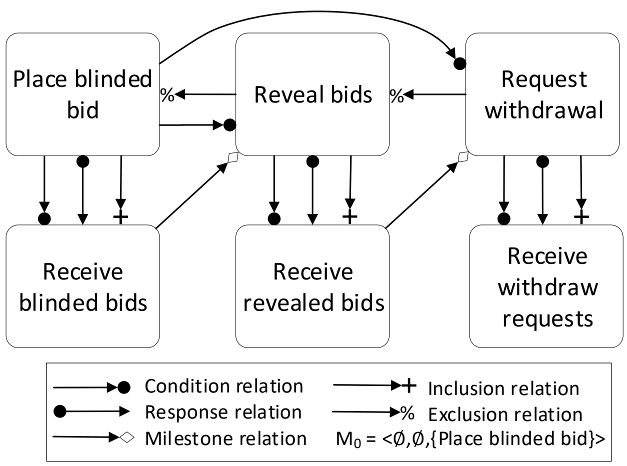


Fig. 2. Blind Auction Workflow as a DCR graph

```
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

1.2 Gambling Games

One of the most widespread smart contract applications is delivering gambling services. In fact, thanks to Blockchain's decentralized nature and the transparency of transactions within it, players can have a clear view of the behaviour of the game and therefore are led and incentivized to put their trust in the system which is determined by the rules implemented by its smart contracts.

Our first Solidity example (Listing 1.2) is based on a published smart contract[1] implementing a lottery game. It has been tweaked to illustrate more vulnerabilities that can be present in a smart contract without altering its purpose.

[1] https://etherscan.io/address/0xa11e4ed59dc94e69612f3111942626ed513cb172

A player participates in this game by sending an amount of ether equal to the *TICKET_AMOUNT* through the function *playTicket()*, which is then added to the game's *pot*. The winner is determined based on a *random* value calculated using the block's timestamp and the *LottoLog* is updated accordingly to keep track of the winners. The winner then gets paid by calling the *getPot()* function and the game's host (*bank*) can start a new round of lotto using the *RestartLotto()* function. This contract may seem fair to inexperienced Solidity developers, but it actually presents multiple vulnerabilities as we will be later explaining.

```
contract EtherLotto {
    address public bank;
    struct GameRecord {
      address winner;
      uint amount;
    }
    uint8 gameNum;
    GameRecord[] LottoLog;
    bool won;
    uint constant TICKET_AMOUNT = 10;
    uint constant FEE_AMOUNT = 1;
    uint public pot;
    function EtherLotto() {
        bank = msg.sender;
        won = false;
        gameNum = 0;
    }
    function RestartLotto() {
      require(msg.sender == bank);
      require(won == true);
      require(pot == 0);
      won = false;
      gameNum += 1;
    }
    function playTicket() payable {
        require(msg.value == TICKET_AMOUNT);
        require(won == false)
        pot += msg.value;
        uint random = uint(sha3(block.timestamp)) % 2;
        if (random == 0) {
            bank.call.value(FEE_AMOUNT)("");
            won = true;
            GameRecord gr;
            gr.winner = msg.sender;
            gr.amount = pot - FEE_AMOUNT;
            LottoLog[gameNum] = gr;
        }
    }
    function getPot() {
```

```
        require(won == true);
        if(msg.sender == LottoLog[gameNum].winner){
            msg.sender.call.value(LottoLog[gameNum].amount)("");
            pot = 0;
        }
    }
}
```

Listing 1.2. Solidity example: EtherLotto.sol

```
contract MaliciousContract {
    uint ticket;
    EtherLotto el = EtherLotto(0xbf0061dc...);
    EtherMilestone em = EtherMilestone(0xc50164dfa...);
    function playLotto() {
        ticket = msg.value;
        el.playTicket.value(ticket)();
        el.getPot();
    }
    function playMilestone() {
        em.play.value(1)();
    }
    function getRevenge ( ) {
        selfdestruct(em);
    }
    function () payable {
        el.getPot();
    }
  }
```

Listing 1.3. A malicious smart contract in Solidity

We consider a second Solidity example[2] (Listing 1.4) to emphasize on the harmful effect that the self-destruction vulnerability (see next subsection) can have on the execution of a contract. It implements another gambling game whereby a player sends 1 ether to the contract by calling the *play()* function in hopes to be the one to hit a milestone. Once the game is over (i.e., the *finalMileStone* is reached) winners can claim their rewards through the *claimReward()* function.

```
contract EtherMilestone {
    uint public payoutMileStone1 = 6 ether;
    uint public mileStone1Reward = 4 ether;
    uint public payoutMileStone2 = 10 ether;
    uint public mileStone2Reward = 6 ether;
    uint public finalMileStone = 20 ether;
    uint public finalReward = 10 ether;
    mapping(address => uint) redeemableEther;
```

[2] https://gist.github.com/vasa-develop/415a17c709d804a4d351485cd1b7c981

```
    function play() public payable {
        require(msg.value == 1 ether);
        uint currentBalance = this.balance + msg.value;
        require(currentBalance <= finalMileStone);
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }
    function claimReward() public {
        require(this.balance == finalMileStone);
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.call.value(redeemableEther[msg.sender])(""
            );
    }
}
```

Listing 1.4. Solidity example: EtherMilestone.sol

Vulnerabilities in Smart Contracts

Integer Overflow/Underflow: due to Solidity's lack of safeguards on mathematical operators, errors such as overflows and underflows may occur as a result of violation of value limitations of integer data types. For instance, the *uint8 gameNum* variable in the *EtherLotto* contract can be the source of such a vulnerability when the game exceeds 256 rounds. In fact, at the 257^{th} round, and due to Solidity's wrapping in two's complement representation for integers, *gameNum* will be set back to 0, causing data errors/overwriting into the critical *LottoLog* variable.

Reentrancy: this is by far the most notorious vulnerability since it led to the infamous DAO attack. An attack of this type can take several forms (e.g, we can talk about a single function reentrancy attack or a cross-function reentrancy attack), but the main idea behind it is that a function can be interrupted in the middle of its execution and then be safely called again before its initial call completes. Once the second call completes, the initial one resumes correct execution. The simplest example is when a smart contract uses a variable to keep track of balances and offers a withdraw function. A vulnerable contract would make a transfer of funds prior to updating the corresponding balance which an attacker can take advantage of by recursively calling this function and eventually draining the contract. This can be illustrated by a call to the function *playLotto()* with a value of 10 in the *MaliciousContract* (Listing 1.3) which would start by playing a ticket in the *EtherLotto* contract by invoking its *playTicket()* function and then attempting its *getPot()* function. In the instance where attacker's ticket is a winning one and the contract holds more than twice the amount of the *pot* in that round, a reentrancy attack can happen. In fact, by sending the jackpot to the winner (line 42 in Listing 1.2), the *EtherLotto* contracts invokes the *fallback* function of the *MaliciousContract*, which is an unnamed function used to receive data or Ether. This is where the control flow is handed over to the latter contract whose *fallback* function recursively calls *getPot()*, which is allowed since the conditions on its execution are still valid, until the *EtherLotto* contract's balance is less than the current *pot*'s amount.

Self-Destruction: the *selfdestruct(address)* function, when implemented in a contract, removes all bytecode from the contract's address to render it inaccessible and sends all its ether to the specified *address*. The latter can be another contract's address, in which case, the ether transfer happens forcibly, regardless of the recipient's code (i.e., without invoking its *fallback* function). Getting back to our second example *EtherMilestone*, we note the use of *this.balance* in lines 11 and 25. A player who missed a milestone, could vengefully send an amount of ether using *selfdestruct()* (e.g., function *getRevenge()* in *MaliciousContract*) as to push the contract's balance above the *finalMileStone*, locking all of the contract's ether and denying the winners who had already reached some milestones their rewards since *claimReward()* would always revert.

Timestamp dependence: since the execution on a Blockchain needs to be deterministic for all the miners to get the same results and reach a consensus, users usually resort to block-related variables such as timestamp as a source of entropy. Sharing the same view on the Blockchain, miners would generate the same result, albeit being unpredictable. Even though this seems to be safe, it gives the miners a small room for manipulation given that they can choose a timestamp within a certain range for the new block, which gives them the possibility to tamper with the results and put some bias towards a certain user for example. Such a vulnerability can be exploited by any contract relying on a time constraint to determine its course of action. In our *EtherLotto* example, the function *playTicket()* is timestamp-dependent.

Skip Empty Literal [3]: the source of this vulnerability is the way the encoder of the Solidity compiler treats the arguments in a function call. In fact, when a function call's argument is an empty string literal, it affects the following arguments which are shifted to the right by 32 bytes. This results in a function call with corrupted data.

Uninitialized Storage Variable: Solidity stores state variables sequentially. So in *EtherLotto*, the variable *bank* is stored in slot 0. Since Solidity uses storage for complex data types like structs by default when declared as local variables, they become pointers to storage. Because *gr* is uninitialized (line 33 in Listing 1.2), it would actually point to the same slot as *bank*. When setting *gr.winner* to the first winner's address, this is effectively changing the address stored in *bank* to the winner's, which results in an unexpected behaviour by this contract. In our example, we present this vulnerability as an error unintentionally introduced by the contract's owner and unintentionally exploited by the first winner. It can, however, be intentionally injected in a contract's code or intentionally exploited by a user, as is the case in the *OpenAddressLottery*[3] honeypot.

2 Overview of our Formal Verification Approach

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

1. A pre-verification phase: consists in transforming the smart contracts' Solidity code into CPN submodels corresponding to their functions.
2. A verification phase: consists in constructing a CPN model with regard to an LTL property that can express: *(i)* a vulnerability in the code or *(ii)* a contract-specific property, linking it to a CPN model representing the provided behavior to be considered, and feeding it the model checking to verify the targeted property.

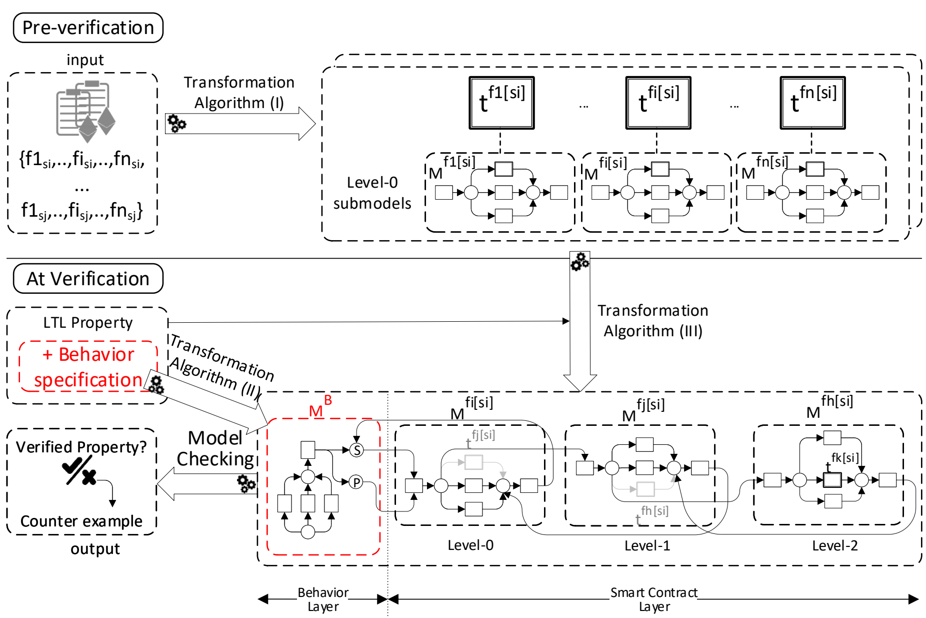


Fig. 3. Overview of the approach

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

[3] https://etherscan.io/address/0x741f1923974464efd0aa70e77800ba5d9ed18902

behavior specification. As shown in Figure 3, we represent each function of a smart contract by an *aggregated transition* that encapsulates a submodel corresponding to the internal workflow of the former. These submodels are initially represented disjointedly. In fact, our aim at this pre-verification phase is to get building blocks for the hierarchical model that will be fed to the model checker. Then, given a behavior specification and an LTL property to be verified, the final CPN model is built by *(1)* linking the aggregated transition representing the targeted function to the behavioral model and *(2)* building a hierarchy by explicitly representing function calls in the submodel in question (if the checked property requires it). In fact, function calls are initially abstracted and therefore represented by aggregated transitions in the model (e.g., $t^{fj[si]}$ in Figure 3) under the assumption that they do not present behavioral problems (deadlock-free and strong-livelock-free) which can be separately verified for each function. Depending on the property to be verified, an aggregated transition may need to be *unfolded* if any of its corresponding function's instructions or variables are involved in the property, hence the multi-level hierarchy in the model (e.g., $t^{fj[si]}$ in $M^{fi[si]}$ is hidden and replaced by its submodel $M^{fj[si]}$). It is kept *folded* otherwise (e.g., $t^{fk[si]}$ in $M^{fh[si]}$). This abstraction leads to a reduction in the size of the state space the model checker needs to explore.

3 Generation of the Hierarchical CPN Model

In order to implement our approach, we propose a transformation algorithm that automates the generation of the final hierarchical CPN model from the provided input artifacts.

3.1 Our HCPN Model: Defining its Elements

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

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

1. aggregated transitions (T^A): used at the level-0 model for the representation of functions, as well as at higher levels for the modular representation of function calls. They are transitions that can be substituted by submodels.
2. regular transitions (T^R): are simple unsubstitutable CPN transitions.

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

- $t.name$, the name of the transition t
- $t.statement$, the Solidity code associated to transition t
- $t.metaColour$, the metaColour associated to the control flow places of transition t (if $t \in T^A$)
- $t.data$, the set of data places associated to transition t (if $t \in T^A$)

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

Places *P*

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

- *Control flow places* P_{CF} are places created to implement the order of execution of the workflow. We also use them to carry data related to the state of the smart contract which can be defined by its balance and the values of its state variables. Such places have a $metaColour$ defined at each aggregated transition of level-0 as the concatenation of the *state* and the *input parameters*: *[uint: contractBalance, $type_{v_1}$: stateVariable$_1$, ..., $type_{v_n}$: stateVariable$_n$, $type_{p_1}$: inputParameter$_1$, ... , $type_{p_n}$: inputParameter$_n$]* (which corresponds to the concatenation of the colour of the input control flow place $\bullet t[cf] \in P_S$ and the colour of the input parameters place $\bullet t[input] \in P_P$ of the transition in question at the second step of our modelling approach).
- *Data places* P_{data} (for internal local variables) where each place is of a colour corresponding to the represented variable's type.
- *Parameter places* P_P that convey potential inputs of function calls. Each function call has an associated parameter place whose colour is as follows *[$type_{p_1}$: inputParameter$_1$, ..., $type_{p_n}$: inputParameter$_n$]*.
- *Return places* P_R that communicate potential functions' returned data. Each function call has an associated return place whose colour corresponds to the return type of the called function.

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

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

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

Expressions $\boldsymbol{E}$

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

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

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

Statements $\boldsymbol{S}$

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

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

  - $st_{RHS}.fctCalls$ designates the set of function calls in the expression of the value to be sent (if $st_{RHS} \in E_F$)
- a returning statement, where:
  - $st_{LHS} = \emptyset$
  - $st_{RHS}.vars$ designates the variables in the expression of the returned value (if $st_{RHS} \in E_V$)
  - $st_{RHS}.fctCalls$ designates the function calls in the expression of the returned value (if $st_{RHS} \in E_F$)

A control statement can be:

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

3.2 Solidity-to-CPN: Our Proposed Transformation Algorithm

The first step is to generate the aggregated transitions for the smart contracts' functions along with their level-0 submodels. To do so, we propose the following algorithms.

generateAggregations Algorithm

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

end for
t^a.*metaColour* ← *newColour*
end for
end procedure

generateLevel0 Algorithm

procedure generateLevel0(t^a)
Input: an aggregated transition t^a
Output: the level-0 CPN submodel of t^a
$P_{data} \leftarrow \emptyset$
getLocalVariables($t^a.statement; P_{data}$)
$t^a \leftarrow P_{data}$
$t^a.submodel$ ← createSubModel($t^a, \emptyset, \emptyset$)
end procedure

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

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

getLocalVariables

procedure getLocalVariables(st; P_{data})
Input: statement st, set of places P_{data} being created
Output: updated P_{data} with the set of places corresponding to local variables in the statement st
if st is a variable declaration statement **then**
create place p
$p.name \leftarrow st_{LHS}.vars.name$
$p.colour \leftarrow st_{LHS}.type$
add p to P_{data}
else if st is a selection statement **then**
getLocalVariables(st_T, P_{data})

if st is a double-branching selection statement **then**
getLocalVariables(st_F, P_{data})
end if
else if st is a looping statement **then**
if st is a for statement: $for(init; c; inc)st_T$ **then**
getLocalVariables($init, P_{data}$)
getLocalVariables(st_T, P_{data})
else if st is a while statement: $while(c)st_T$ **then**
getLocalVariables(st_T, P_{data})
end if
else if st is a compound statement $\{st[1]; st[2]; \ldots; st[N]\}$ **then**
for $i = 1..N$ **do**
getLocalVariables($st[i], P_{data}$)
end for
end if
end procedure

We see a smart contract function as a set of statements. To each one of the statement types we define a corresponding pattern in CPN, according to which a snippet of a CPN model is generated. The resulting snippets are linked according to the function's internal workflow. The *createSubModel* implements such correspondences[4].

createSubModel Algorithm

procedure createSubModel(t; st; $\mathrm{p_{in}}$; $\mathrm{p_{out}}$)
Input: transition t, statement st, control flow input place p_{in}, control flow output place p_{out}
Output: submodel of transition t
switch st **do**
case compound statement $\{st[1]; st[2]; \ldots; st[N]\}$
buildCompoundStatement(*t;st;p_{in};p_{out}*)
case simple statement
switch st **do**
case assignment statement
buildAssignmentStatement(*t;st;p_{in};p_{out}*)
case variable declaration statement
buildVariableDeclarationStatement(*tst;;p_{in};p_{out}*)
case sending statement
buildSendingStatement(*t;st;p_{in};p_{out}*)
case returning statement
buildReturningStatement(*t;st;p_{in};p_{out}*)
case function call statement

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

```
                buildFunctionCallStatement(t;st;p_in;p_out)
            end switch
        case control statement
            switch st do
                case requirement statement
                    buildRequirementStatement(t;st;p_in;p_out)
                case selection statement
                    buildSelectionStatement(t;st;p_in;p_out)
                case looping statement
                    switch st do
                        case for statement
                            buildForLoopStatement(t;st;p_in;p_out)
                        case while statement
                            buildWhileLoopStatement(t;st;p_in;p_out)
                    end switch
            end switch
    end switch
end procedure
```

buildCompoundStatement Algorithm

```
procedure buildCompoundStatement(t; st; p_in; p_out)
    Input: transition t, a compound statement st ={st[1]; st[2]; ...; st[N]},
control flow input place p_in, control flow output place p_out
    Output: submodel for statement st
    for i = 1..N − 1 do
        create place p_i
    end for
    createSubModel(t;st[1];p_in;p_1)
    for i = 2..N − 1 do
        createSubModel(t;st[i];p_i-1;p_i)
    end for
    createSubModel(t;st[N];p_N-1;p_out)
end procedure
```

buildAssignmentStatement Algorithm

```
procedure buildAssignmentStatement(t; st; p_in; p_out)
    Input: transition t, an assignment statement st = (st_LHS, st_RHS), control flow input place p_in, control flow output place p_out
    Output: submodel for statement st
    create transition t'
    create arc from p_in to t'
    connectLocalVariables(st_RHS.vars\{st_LHS.vars};t;t')
    connectFunctionCalls(st_RHS.fctCalls;t)
    if st_LHS.vars is a local variable then
```

create arc from $t.data[st_{LHS}.vars]$ to t'
create arc from t' to $t.data[st_{LHS}.vars]$ with inscription st_{RHS}
create arc from t' to p_{out}
else
create arc from t' to p_{out} with inscription *outInsc* $\leftarrow$ *inInsc* in which the variable corresponding to $st_{LHS}.vars$ is replaced by st_{RHS}
end if
end procedure

buildVariableDeclarationStatement Algorithm

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

buildSendingStatement Algorithm

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

buildReturningStatement Algorithm

procedure buildReturningStatement(t; st; $\mathrm{p_{in}}$; $\mathrm{p_{out}}$)
Input: transition t, a returning statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
Output: submodel for statement *st*
create transition t'
create arc from p_{in} to t'
connectLocalVariables*($st_{RHS}.vars$;t;t')*

CONNECTFUNCTIONCALLS($st_{RHS}.fctCalls$;t)
create arc from t' to $t\bullet[cf]$
create arc from t' to $t\bullet[output]$ with inscription $outInsc \leftarrow [inInsc.sender, inInsc.balance, st_{RHS}]$
end procedure

BUILDFUNCTIONCALLSTATEMENT Algorithm

procedure BUILDFUNCTIONCALLSTATEMENT(t; st; p_{in}; p_{out})
Input: transition t, a function call statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
Output: submodel for statement st
create transition t^f
create place p_{param_f}
create arc from p_{in} to t^f
create arc from p_{param_f} to t^f
CONNECTLOCALVARIABLES($f_{RHS}.vars$;t;t^f)
CONNECTFUNCTIONCALLS($f_{RHS}.fctCalls$;t)
create arc from t^f to p_{out} with a placeholder inscription
end procedure

BUILDREQUIREMENTSTATEMENT Algorithm

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

BUILDSELECTIONSTATEMENT Algorithm

procedure BUILDSELECTIONSTATEMENT(t; st; p_{in}; p_{out})
Input: transition t, a selection statement $st = if(c)\ then\ st_T\ [else\ st_F]$, control flow input place p_{in}, control flow output place p_{out}
Output: submodel for statement st

create place p_T
create transition t_T
$t_T.guard \leftarrow c$
create arc from p_{in} to t_T
create arc from t_T to p_T
connectLocalVariables*(c.vars;t;t_T)*
connectFunctionCalls*(c.fctCalls;t_T)*
createSubModel*(t;st_T;p_T;p_{out})*
create transition t_F
$t_F.guard \leftarrow !c$
create arc from p_{in} to t_F
connectLocalVariables*(c.vars;t;t_F)*
connectFunctionCalls*(c.fctCalls;t_F)*
if st is a selection statement: $if(c)$ *then* st_T **then**
    create arc from t_F to p_{out}
else if st is a selection statement: $if(c)$ *then* st_T *else* st_F **then**
    create place p_F
    create arc from t_F to p_F
    createSubModel*(t;st_F;p_F;p_{out})*
end if
end procedure

buildForLoopStatement Algorithm

procedure buildForLoopStatement(t; st; p_{in}; p_{out})
Input: transition t, a *for* looping statement $st = for(init; c; inc)\ st_T$, control flow input place p_{in}, control flow output place p_{out}
Output: submodel for statement st
create place p_{init}
create place p_c
create place p_T
createSubModel*(t;init;p_{in};p_{init})*
create transition t_T
$t_T.guard \leftarrow c$
create arc from p_{init} to t_T
connectLocalVariables*(c.vars;t;t_T)*
connectFunctionCalls*(c.fctCalls;t_T)*
create arc from t_T to p_c
create transition t_F
$t_F.guard \leftarrow !c$
create arc from p_{init} to t_F
connectLocalVariables*(c.vars;t;t_F)*
connectFunctionCalls*(c.fctCalls;t_F)*
create arc from t_F to p_{out}
createSubModel*(t;st_T;p_c;p_T)*
createSubModel*(t;inc;p_T;p_{init})*

end procedure

buildWhileLoopStatement Algorithm

procedure buildWhileLoopStatement(t; st; p_{in}; p_{out})
  Input: transition t, a *while* looping statement $st = while(c)\ st_T\ st_T$, control flow input place p_{in}, control flow output place p_{out}
  Output: submodel for statement st
  create place p_T
  create transition t_T
  $t_T.guard \leftarrow c$
  create arc from p_{in} to t_T
  connectLocalVariables*($c.vars$;t;t_T)*
  connectFunctionCalls*($c.fctCalls$;t_T)*
  create arc from t_T to p_T
  create transition t_F
  $t_F.guard \leftarrow !c$
  create arc from p_{in} to t_F
  connectLocalVariables*($c.vars$;t;t_F)*
  connectFunctionCalls*($c.fctCalls$;t_F)*
  create arc from t_F to p_{out}
  createSubModel*(t;st_T;p_T;p_{in})*
end procedure

connectLocalVariables Algorithm

procedure connectLocalVariables(V; t; t')
  Input: set of local variables V, transition t, transition t'
  Output: submodel with connections to local variables
  for $v \in V$ **do**
    create arc from $t.data[v]$ to t'
    create arc from t' to $t.data[v]$
  end for
end procedure

connectFunctionCalls Algorithm

procedure connectFunctionCalls(FC; t)
  Input: set of function calls FC, transition t
  Output: submodel with connections to function calls
  for $f \in FC$ **do**
    create transition t^f
    create place p_{return_f}
    create place p_{param_f}
    connectLocalVariables*($f_{RHS}.vars$;t;t^f)*
    create arc from p_{param_f} to t^f with inscription in which every element of f_{RHS} is replaced by its corresponding argument

create arc from t^f to p_{return_f} with a placeholder inscription
end for
end procedure

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

unfoldTransition Algorithm

procedure unfoldTransition($\mathrm{t^a;p_{in};p_{out}}$)
  Input: aggregated transition t^a, input place p_{in}, output place p_{out}
  Output: submodel replacement of transition t^a
  for $t' \in t^a.submodel.inTransition$ **do**
    replicate (arc from p_{in} to t^a) to t'
    replicate (arc from $\bullet t[input]$ to t^a) to t'
    for $p \in \bullet t^a[data] \cup \bullet t^a[output]$ **do**
      replicate (arc from p to t^a) to t'
    end for
  end for
  for $t' \in t^a.submodel.outTransition$ **do**
    replicate (arc from t^a to p_{out}) to t' with the placeholder inscription replaced by values from $\bullet t'[cf]$
  end for
  hide transition t^a and all arcs linked to it
end procedure

3.3 Behavior-to-CPN: Generation of the Behavioral Layer

In this paper, we focus on the second phase of our approach, and more particularly on the generation of the CPN model corresponding to the input behavior specification. We consider three types of behaviors for smart contracts:

- a completely-free behavior: if no information is provided on the context in which a smart contract is used
- a constrained behavior: if conditions on the way a smart contract is used are provided (e.g., as a DCR Graph)
- a fully-specified behavior: if the context in which a smart contract is used is provided (e.g., as a BPMN model)

Such a behavioral model is added to the smart contract's model as an additional layer which is then linked to the hierarchical model built using the previously generated CPN submodels. In the following subsections we will be detailing our proposed CPN models for the representation of the first and second types of behavior specifications. Existing studies on BPMN-to-CPN transformation [2, 8, 11] could be leveraged for the representation of the third type.

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

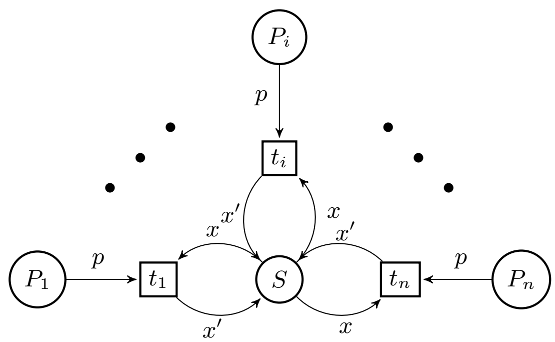


Fig. 4. CPN model for a completely-free behavior

Modeling a Constrained Behavior The user may want to define the behavior of smart contracts by specifying a set of constraints. This can be captured using a DCR Graph. In order to be able to integrate such a representation in our CPN hierarchical model, we propose a CPN model for DCR. We first recall the formal definitions of the CPN formalism (Definitions 1-5) and DCR graphs (Definitions 6-9), then we define our proposed model (Definition 10). We then elaborate a theorem regarding the semantic equivalence between our model and DCR graphs (Theorem 2 and Proof 3.3).

Coloured Petri Net

A Petri net [10] is a formal model with mathematics-based execution semantics. It is a directed bipartite graph with two types of nodes: places (drawn as circles) and transitions (drawn as rectangles). Despite its efficiency in modelling and analysing systems, a basic Petri net falls short when the system is too complex, especially when representation of data is required. To overcome such limitations, extensions to basic Petri nets were proposed, equipping the tokens with colours or types [5], [13] 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) [6] 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 [9], to define data types.

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

1. P *is a finite set of* places*.*
2. T *is a finite set of* transitions *such that* $P \cap T = \emptyset$*.*
3. $A \subseteq (P \times T) \cup (T \times P)$ *is a set of directed* arcs*.*
4. Σ *is a finite set of non-empty* colour sets*.*
5. V *is a finite set of* typed variables *such that* $Type[v] \in \Sigma$ *for all variables* $v \in V$*.*
6. $C : P \to \Sigma$ *is a* colour set function *that assigns a colour set to each place.*
7. $G : T \to EXPR_V$*, where* $EXPR_V$ *is the set of expressions provided by CPN ML with variables in* V*, is a* guard function *that assigns a guard to each transition* t *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 2 (CPN concepts). *For a Coloured Petri Net* $CPN = (P, T, A, \Sigma, V, C, G, E, I)$*, we define the following concepts:*

1. $\bullet p$ *and* $p\bullet$ *respectively denote the sets of input and output transitions of a place* p*.*
2. $\bullet t$ *and* $t\bullet$ *respectively denote the sets of input and output places of a transition* t*.*
3. *A* marking *is a function* M *that maps each place* $p \in P$ *into a multiset of tokens* $M(p) \in C(p)_{MS}$*.*
4. *The* initial marking M_0 *is defined by* $M_0(p) = I(p)\langle\rangle$ *for all* $p \in P$*.*
5. *The* variables of a transition t *are denoted by* $Var(t) \subseteq V$ *and consist of the free variables appearing in its guard and in the arc expressions of its connected arcs.*
6. *A* binding *of a transition* t *is a function* b *that maps each variable* $v \in Var(t)$ *into a value* $b(v) \in Type[v]$*. It is written as* $\langle var_1 = val_1, ..., var_n = val_n \rangle$*. The set of all bindings for a transition* t *is denoted* $B(t)$*.*
7. *A* binding element *is a pair* (t, b) *such that* $t \in T$ *and* $b \in B(t)$*. The set of all binding elements* $BE(t)$ *for a transition* t *is defined by* $BE(t) = \{(t, b) | b \in B(t)\}$*. The set of all binding elements in a CPN model is denoted* BE*.*
8. *A* step $Y \in BE_{MS}$ *is a non-empty, finite multiset of binding elements.*

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

Definition 3 (Enabling and occurrence of a binding element). *A binding element $(t, b) \in BE$ is enabled in a marking M if and only if the following two properties are satisfied:*

1. $G(t)\langle b\rangle$.
2. $\forall p \in P : E(p, t)\langle b\rangle \lll= M(p)$.
   When (t, b) is enabled in M, it may occur, leading to the marking M' defined by:
3. $\forall p \in P : M'(p) = (M(p) - -E(p, t)\langle b\rangle) + +E(t, p)\langle b\rangle$.

Definition 4 (Enabling and occurrence of steps). *A step $Y \in BE_{MS}$ is enabled in a marking M if and only if the following two properties are satisfied:*

1. $\forall (t, b) \in Y : G(t)\langle b\rangle$.
2. $\forall p \in P :{}^{++}_{MS} \sum_{(t,b)\in Y} E(p, t)\langle b\rangle \lll= M(p)$
   When Y is enabled in M, it may occur, leading to the marking M' defined by:
3. $\forall p \in P : M'(p) = (M(p) - -{}^{++}_{MS} \sum_{(t,b)\in Y} E(p, t)\langle b\rangle) + +{}^{++}_{MS} \sum_{(t,b)\in Y} E(t, p)\langle b\rangle$.

Definition 5 (Occurrence sequences and reachability). *A finite occurrence sequence of length $n \geq 0$ is an alternating sequence of markings and steps, written as*

$$M_1 \xrightarrow{Y_1} M_2 \xrightarrow{Y_2} M_3 ... M_n \xrightarrow{Y_n} M_{n+1}$$

such that $M_i \xrightarrow{Y_i} M_{i+1}$ for all $1 \leq i \leq n$. All markings in the sequence are said to be reachable from M_1. This implies that an arbitrary marking M is reachable from itself by the trivial occurrence sequence of length 0.

Analogously, an infinite occurrence sequence is a sequence of markings and steps

$$M_1 \xrightarrow{Y_1} M_2 \xrightarrow{Y_2} M_3 \xrightarrow{Y_3} ...$$

such that $M_i \xrightarrow{Y_i} M_{i+1}$ for all $i \geq 1$. The set of markings reachable from a marking M is denoted $\mathscr{R}(M)$. The set of reachable markings is $\mathscr{R}(M_0)$, i.e., the set of markings reachable from the initial marking M_0.

Theorem 1. *Let Y be a step and M and M' be markings such that $M \xrightarrow{Y} M'$. Let $Y1$ and $Y2$ be steps such that*

$$Y = Y1 + +Y2$$

Then there exists a marking M'' such that

$$M \xrightarrow{Y_1} M'' \xrightarrow{Y_2} M'$$

Definition 6. *A dynamic condition response graph is a tuple* $G = (E, M, Act, \to\bullet, \bullet\to, \to+, \to\%, \to\diamond, l)$ *where*

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

The marking (2) $M = (Ex, Re, In) \in \mathcal{M}(G)$ *is a triplet of event sets where the first component represents the set of events that have previously been executed (*Ex*), the second component represents the set of events that are pending responses required to be executed or excluded (*Re*), and third components represents the set of events that are currently included (*In*). The idea conveyed by the dynamic inclusion/exclusion relations (6)* $\to+$ *and* $\to\%$ *is that only the currently included events are considered in evaluating the constraints. In other words, if an event* b *is a condition for an event* a*, but it is excluded from the graph then it no longer restricts the execution of the event* a*. Moreover, if event* b *is the response for an event* a *but it is excluded from the graph, then it is no longer required to happen for the flow to be acceptable. The inclusion relation* $e \to+ e'$ *means that, whenever* e *is executed,* e' *becomes included in the graph if it is not already. The exclusion relation* $e \to\% e'$ *means that when* e *is executed,* e' *becomes excluded from the graph if it is not already. The milestone relation (7) is similar to the condition relation in that it is a blocking one. The difference is that it based on the events in the pending response set. In other words, if an event* b *is a milestone of an event* a *(*$b \to\diamond a$*), then the event* a *cannot be executed as long as the event* b *is in the set of pending responses (*Re*). It is worth mentioning that, like a condition relation, a milestone relation is only blocking when the event in question is included in the graph.*

Definition 7 (Enabled event). *For a dynamic condition response graph* $G = (E, M, Act, \to\bullet, \bullet\to, \to+, \to\%, \to\diamond, l)$ *with marking* $M = \{Ex; Re; In\}$*, we define that an event* $e \in E$ *is enabled, written as* $M \vdash_G e$ *if*

1. $e \in In$
2. $(\to\bullet e \cap In) \in Ex$
3. $(\to\diamond e \cap In) \in E \setminus Re$

Definition 8 (Event execution effect). *For a dynamic condition response graph* $G = (E, M, Act, \to\bullet, \bullet\to, \to+, \to\%, \to\diamond, l)$ *with marking* $M = \{Ex; Re; In\}$

and with an enabled event $M \vdash_G e$, the result of executing the event e will be a dynamic condition response graph $G = (E, M', Act, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond, l)$, where $M' = M \oplus_G e = \{Ex'; Re'; In'\}$ such that

1. $Ex' = Ex \cup \{e\}$
2. $Re' = (Re \setminus \{e\}) \cup e\bullet\rightarrow$
3. $In' = (In \cup e \rightarrow+) \setminus e \rightarrow\%$

Definition 9 (Graph execution). *For a Dynamic Condition Response Graph $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond, l)$ we define an execution of G to be a (finite or infinite) sequence of tuples $\{(M_i; e_i; a_i; M_i')\}_{i\in[k]}$ each consisting of a marking, an event, a label and another marking (the result of executing the event) such that*

1. $M = M_0$
2. $\forall i \in [k].a_i \in l(e_i)$
3. $\forall i \in [k].M_i \vdash_G e_i$
4. $\forall i \in [k].M_i' = M_i \oplus_G e_i$
5. $\forall i \in [k-1].M'i = M_{i+1}$.

Further, we say the execution (or a run) is accepting if $\forall i \in [k].(\forall e \in In_i \cap Re_i.\exists j \geq i.e_j = e \vee e \notin In_j')$, where $M_i = (Ex_i; In_i; Re_i)$ and $M_j' = (Ex_j'; In_j'; Re_j')$.

Finally we say that a marking M' is reachable in G (from the marking M) if there exists a finite execution ending in M' and let $\mathcal{M}_{M\rightarrow^*}(G)$ denote the set of all reachable markings from M.

CPN4DCR Model

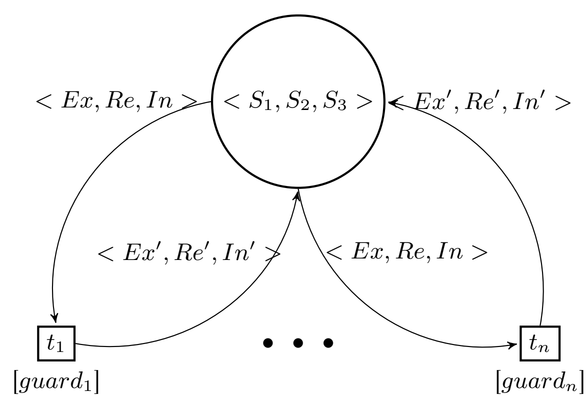


Fig. 5. CPN model for a DCR Graph

Definition 10 (CPN4DCR). *Given a dynamic condition response graph $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$, a corresponding CPN model $CPN = (P, T, A, \Sigma, V, C, G, E, I)$ is defined such that:*

- $P = \{S\}$
- $T = \{t_i, \forall i \in [1,n]\}$, *with* $n = |E|$ *the number of events in* G
- $A = \{(t_i, S), \forall i \in T\} \cup \{(S, t_i), \forall i \in T\}$
- $\Sigma = \{C_E, (C_E \times C_E \times C_E)\}$, *where* C_E *is a colour defined to represent a set of events. Here we define* C_E *as an integer type* ($C_E = rangeINT$) *where each event* $e_i \in E$ *is represented in* C_E *by its index.*
- $V = \{Ex, Re, In, Ex', Re', In'\}$, *with* $Type[v] = C_E, \forall v \in V$
- $C = \{S \rightarrow (C_E \times C_E \times C_E)\}$
- $G = \{t_i \rightarrow guard_i, \forall i \in [1,n]\}$, *with* $n = |E|$
- $E = \{a \rightarrow < Ex, Re, In >, \forall a \in A \cap (P \cup T)\} \cup \{a \rightarrow < Ex', Re', In' >, \forall a \in A \cap (T \cup P)\}$ *with*
  - $Ex' = Ex \cup e_i$
  - $Re' = (Re \backslash e_i) \cup e\bullet\!\!\rightarrow$
  - $In' = (In \cup e_i \rightarrow\!\!+) \backslash e \rightarrow\!\!\%$
- $I = \{S \rightarrow < S_1, S_2, S_3 >\}$ *with* $< S_1, S_2, S_3 >$ *corresponding to the initial marking* M *of* G

For each transition t_i *in the CPN model representing an event* e_i *in the DCR graph, we further precise that:*

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

Definition 11 (Marking Equivalence). *A marking* $M^G =< Ex, Re, In >$ *of a DCR graph* G *is said to be equivalent to a marking* $M^C =< S \rightarrow < S_1, S_2, S_3 >>$ *of a CPN model* C *iff*

- $\forall e_i \in Ex$ *(respectively* Re *and* In*)*, $\exists i \in S_1$ *(respectively* S_2 *and* S_3*), and*
- $\forall i \in S_1$ *(respectively* S_2 *and* S_3*)*, $\exists e_i \in Ex$ *(respectively* Re *and* In*)*

We note $M^G \equiv M^C$.

Definition 12 (Execution Sequence Equivalence). *An execution sequence of length* k, $\sigma_k^G =< e_i, ..., e_j >$ *of a DCR graph* G *is said to be equivalent to an execution sequence of length* k, $\sigma_k^C =< t_i, ..., t_j >$ *of a CPN model* C *iff*

- $(M_1^G \equiv M_1^C \wedge M_1^G \xrightarrow{\sigma_k^G} M_2^G \wedge M_1^C \xrightarrow{\sigma_k^C} M_2^C) \implies M_2^G \equiv M_2^C$

We note $\sigma_k^G \equiv \sigma_k^C$.

Theorem 2. *Let* G *be a DCR graph and* C *the corresponding CPN model generated by following definition 10, then* G *and* C *are semantically equivalent.*

Proof. Let G be a DCR graph and C the corresponding CPN model generated by following definition 10. In order to prove that G and C are semantically equivalent we need to prove that

1. $\forall \sigma_k^G =< e_1, ..., e_k >, \exists \sigma_k^C =< t_1, ..., t_k >$, and
2. $\forall \sigma_k^C =< t_1, ..., t_k >, \exists \sigma_k^G =< e_1, ..., e_k >$

such that $\sigma_k^G \equiv \sigma_k^C$, $\forall k \in [1, m]$ with m the length of the longest execution sequence.
We start by proving *(1)*:

- Let $P(n)$ be the statement: $\forall \sigma_n^G =< e_1, ..., e_n >, \exists \sigma_n^C =< t_1, ..., t_n >$ such that $\sigma_n^G \equiv \sigma_n^C$.
- $P(1) : \forall \sigma_1^G =< e_1 >, \exists \sigma_1^C =< t_1 >$ such that $\sigma_1^G \equiv \sigma_1^C$. This can be derived from Definition 10. In fact, the initial marking of C (M_0^C) being defined as equivalent to that of G (M_0^G), and the guard of each transition $t_i \in T$ being defined as to correspond to the enabling conditions of the relative event $e_i \in E$, we can deduce that the set of fireable transitions ($M_0^C \rightarrow$) corresponds to the set of enabled events ($M_0^G \rightarrow$). Additionally, the marking M_i^C obtained by firing t_i is equivalent to that obtained by executing e_i ($M_i^C \equiv M_i^G$) since the elements of M_i^C are defined as to correspond to the effect of the execution of e_i in G.
- Assume that $P(k) : \forall \sigma_k^G =< e_1, ..., e_k >, \exists \sigma_k^C =< t_1, ..., t_k >$ such that $\sigma_k^G \equiv \sigma_k^C$ is true for some $k \in [2, m-1]$. We will prove that $P(k+1) : \forall \sigma_{k+1}^G =< e_1, ..., e_{k+1} >, \exists \sigma_{k+1}^C =< t_1, ..., t_{k+1} >$ such that $\sigma_{k+1}^G \equiv \sigma_{k+1}^C$ is true.

$$\sigma_{k+1}^G \equiv \sigma_{k+1}^C \implies \exists e_{k+1} \in E, t_{k+1} \in T \text{ such that } \sigma_k^G \cdot e_{k+1} \equiv \sigma_k^C \cdot t_{k+1} \quad (1)$$

$$\sigma_k^G \equiv \sigma_k^C \iff (M_0^G \xrightarrow{\sigma_k^G} M_k^G \wedge M_0^C \xrightarrow{\sigma_k^C} M_k^C \wedge M_k^G \equiv M_k^C) \quad (2)$$

Analogously to the reasoning in the previous point, we can deduce that:

$$\begin{aligned} &\forall e_{k+1} \in E \text{ such that } M_k^G \xrightarrow{e_{k+1}} M_{k+1}^G, \\ &\exists t_{k+1} \in T \text{ such that } (M_k^C \xrightarrow{t_{k+1}} M_{k+1}^C \wedge M_{k+1}^G \equiv M_{k+1}^C) \end{aligned} \quad (3)$$

And therefore:

$$\forall \sigma_{k+1}^G =< e_1, ..., e_{k+1} >, \exists \sigma_{k+1}^C =< t_1, ..., t_{k+1} > \text{ such that } \sigma_{k+1}^G \equiv \sigma_{k+1}^C \quad (4)$$

The second part *(2)* is provable following a similar reasoning.

4 Formal Verification of Smart Contracts

In our proposed verification approach, we rely on *Helena* in the model checking phase to verify LTL properties [12] that express the susceptibility of contracts to vulnerabilities. To this aim, we start by expressing each targeted vulnerability in LTL (see Section 1.2 for the explanation of these vulnerabilities).

4.1 Expressing Vulnerabilities in LTL

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned} selfDestruction(t^f_{s_i}) =& \neg(testOnBalance(t^f_{s_i}) \land containsSelfDestruct(t^g_{s_j}, s_i)) \\ & \lor (\neg selfDestruct(t^g_{s_j}, s_i) \cup start(t^f_{s_i})) \end{aligned}$$

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

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

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

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

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

$$\begin{aligned} SkipEmpty(t^f_{s_i}) =& containsFunctionCall(t^f_{s_i}) \rightarrow (\Box\neg(isFunctionCall \\ & \land \exists arg \in functionCall \backslash (arg = \text{``''} \land \neg isLast(arg))) \end{aligned}$$

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

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

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

4.2 Model Checking of Smart Contracts

The application of our approach on the Blind Auction use case presented in Section 1.1 yields a hierarchical CPN model whose level-0 submodels are shown in Figures 6-8. 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}.

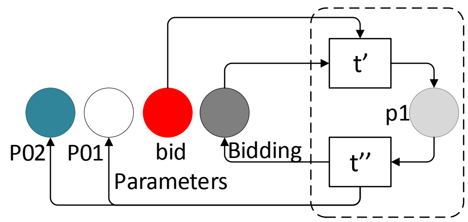


Fig. 6. SubModel of transition *bid*

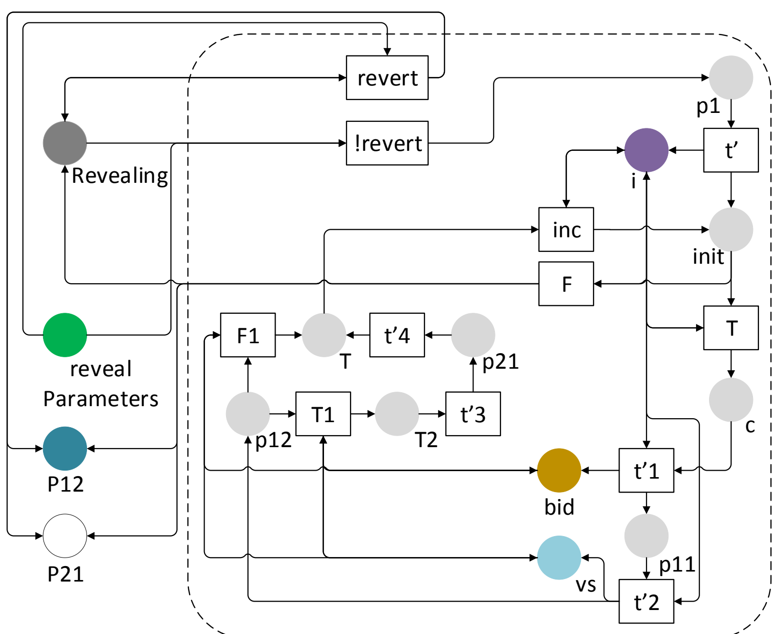


Fig. 7. SubModel of transition *reveal*

Figure 9 represents the resulting submodel obtained by applying our transformation algorithm on the function *play()* of the *EtherMilestone* smart contract.

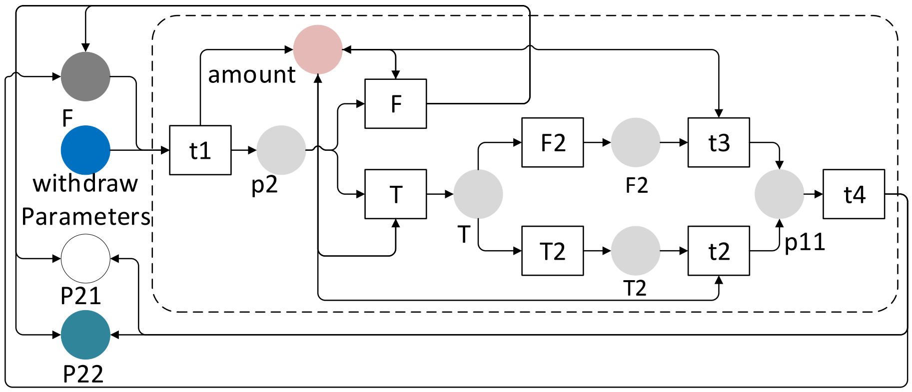


Fig. 8. SubModel of transition *withdraw*

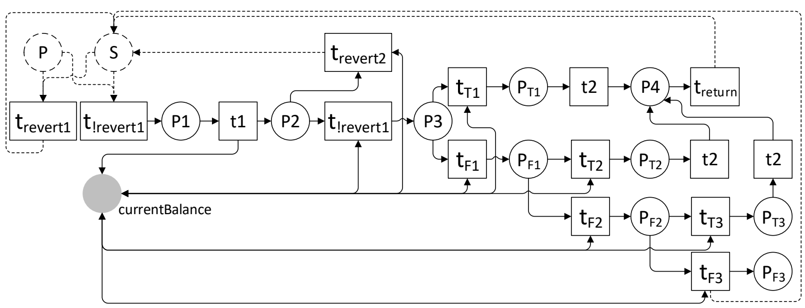


Fig. 9. CPN submodel of the *play()* function in *EtherMilestone*

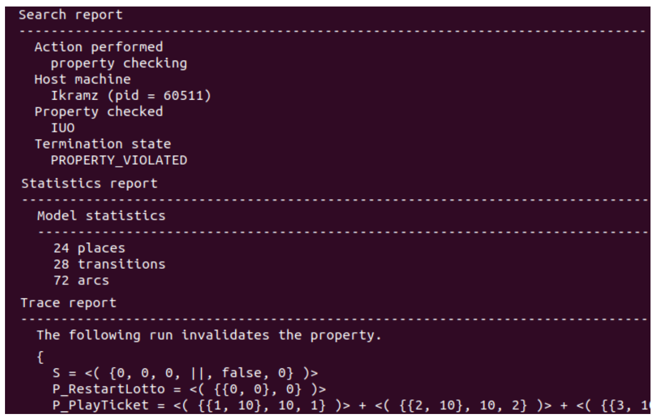
Once we have applied our transformation algorithm to first generate the submodels of a smart contract's functions, verifying properties of the contract would come down to verifying properties on the corresponding CPN model. For model checking, we chose *Helena* [4] which is a High LEvel Nets Analyzer available as a command line tool. It offers explicit model checking support for an on-the-fly verification of state and LTL properties over CPN models described in *Helena*'s specification language.

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

```
ltl property IUO:
   [] not outOfRange;
```

Fig. 10. The integer overflow/underflow LTL property in *Helena*

We have generated the CPN models of our use cases for *Helena* using our prototype for the transformation algorithm (see Figure 9 for an example of a visual representation of the CPN submodel of the function *play()* in *EtherMilestone*). We have then written the corresponding properties in *Helena*'s language for the vulnerabilities in Section 1.2. Thanks to our approach, we were able to detect these vulnerabilities as well as contract-specific properties that we have established for our examples. Figure 10 shows the corresponding property written in *Helena* for the *IUO* LTL property applied on the variable *gameNum* in *EtherLotto* and Figure 11 is a snippet of the result of the model checker showing the detection of the vulnerability and the indication of a counter example.



```
Search report
--------------------------------------------------------------------------------
  Action performed
    property checking
  Host machine
    Ikramz (pid = 60511)
  Property checked
    IUO
  Termination state
    PROPERTY_VIOLATED
Statistics report
--------------------------------------------------------------------------------
  Model statistics
  ------------------------------------------------------------------------------
    24 places
    28 transitions
    72 arcs
Trace report
--------------------------------------------------------------------------------
  The following run invalidates the property.
  {
    S = <( {0, 0, 0, ||, false, 0} )>
    P_RestartLotto = <( {{0, 0}, 0} )>
    P_PlayTicket = <( {{1, 10}, 10, 1} )> + <( {{2, 10}, 10, 2} )> + <( {{3, 1
```

Fig. 11. Model checking result

5 Conclusion

Being an important pillar for Blockchain technology, smart contracts need to provide certain guarantees in terms of correctness 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 (e.g., reentrancy) which have been reported to be the root of some attacks or malfunctions. Checking the absence of such vulnerabilities in a smart contract does not guarantee its correctness as a faulty behaviour may stem from a flaw specific to that contract. Moreover, 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 CPN-based formal verification approach for Solidity smart contracts. We implement a prototype of a transformation algorithm that generates a hierarchical CPN model representing a given Solidity smart contract, including both its control-flow and data aspects. Temporal properties are then verified on the CPN model to check corresponding properties on the smart contract, unrestrictedly to certain predefined vulnerabilities. Besides, our approach takes into account the context in which the smart contracts to be verified are executed as a behavior specification, while also considering the case where no such specification is provided. To further improve the tool's performance, we intend to work on *Helena*'s model checker by embedding it with an extension to an existing technique previously developed to deal with the state space explosion problem in regular PNs [7] and applying it on CPNs.

References

1. Solidity 0.6.11 documentation. https://solidity.readthedocs.io/en/v0.6.11/index.html
2. Dechsupa, C., Vatanawood, W., Thongtak, A.: Transformation of the BPMN design model into a colored petri net using the partitioning approach. IEEE Access **6**, 38421–38436 (2018)
3. Dingman, W., Cohen, A., Ferrara, N., Lynch, A., Jasinski, P., Black, P.E., Deng, L.: Defects and vulnerabilities in smart contracts, a classification using the NIST bugs framework. IJNDC **7**(3), 121–132 (2019)
4. 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)
5. Jensen, K.: Coloured petri nets: A high level language for system design and analysis. In: International Conference on Application and Theory of Petri Nets. pp. 342–416. Springer (1989)
6. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer Publishing Company, Incorporated, 1st edn. (2009)
7. 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
8. Meghzili, S., Chaoui, A., Strecker, M., Kerkouche, E.: An approach for the transformation and verification of BPMN models to colored petri nets models. Int. J. Softw. Innov. **8**(1), 17–49 (2020)
9. Milner, R., Tofte, M., Harper, R.: Definition of standard ML. MIT Press (1990)
10. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
11. Najem, T., Perucci, A.: Mapping BPMN2 service choreographies to colored petri nets. In: Cámara, J., Steffen, M. (eds.) Software Engineering and Formal Methods - SEFM 2019 Collocated Workshops: CoSim-CPS, ASYDE, CIFMA, and FOCLASA, Oslo, Norway, September 16-20, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12226, pp. 85–100. Springer (2019)
12. Pnueli, A.: The temporal semantics of concurrent programs. Theor. Comput. Sci. **13** (1981)
13. Van Hee, K., Verkoulen, P.: 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. pp. 410–431 (1991)