Model Checking of Vulnerabilities in Smart Contracts: A Solidity-to-CPN Approach Extended Content

This document contains complementary content for the paper "Model Checking of Vulnerabilities in Smart Contracts: A Solidity-to-CPN Approach" submitted to SAC 2022 (The 37th ACM Symposium on Applied Computing).

Abstract. Whether it's used as an additional protection layer or adopted as an underlying structure for development, the blockchain technology has been gaining more attention in the field of safety-critical systems. Despite the edge this new technology brings to this field, its adoption does not come without challenges. In fact, smart contracts, which are at the core of second-generation blockchains, can often be riddled with vulnerabilities that can be exploited to attack the platform and threaten its security. It is therefore crucial for the protection of the designed systems to prove the correctness of the smart contracts to be deployed on a blockchain. Existing approaches have been proposed to detect generic vulnerabilities like reentrancy, but the results would often include false positives where the detected vulnerability is either non existent or not exploitable. Besides, such approaches do not offer the possibility to check contract-specific properties.

In this paper we propose a formal approach based on the transformation of Solidity smart contracts into Coloured Petri nets, the expression of vulnerabilities in smart contracts using temporal logic formulae and their verification on the modeled contract. We leverage the capability of the *Helena* model checker to first, detect vulnerabilities in smart contracts while discerning their exploitability, and second, check temporal-based contract-specific properties.

Keywords: Blockchain · Model checker · Smart Contract · Solidity · Coloured Petri Nets · Temporal properties.

1 Introduction

First known as the supporting technology of the renowned Bitcoin cryptocurrency, the Blockchain has ever since known many advances that took it from being merely a database recording transactions between parties to being a computational platform on which smart contracts can be invoked as transactions. This leap significantly expands the power of blockchain systems, and increases their reach to many application fields. This can be particularly observed in the growing interest blockchains are gaining as part of safety-critical software systems, in domains such as health records, banking, voting, personal identity, etc [27]. In fact, some challenges for safety-critical systems could be overcome

when adopting key ideas behind blockchains [5]. The blockchain technology has multiple features (e.g., immutability, non-tampering, security encryption, distributed and decentralized storage...) that offer apparent applicability to safety-critical systems, especially in adding an extra layer of protection. As opposed to conventional centralised databases and computational platforms, blockchains can reduce some counter-party and operational risks by providing neutral ground between organisations, but on the other hand, they can create new risks and security challenges.

While blockchain technology itself has proved to be highly-tamper resistant, many attacks with significant consequences have been waged on several blockchain platforms, exploiting hidden vulnerabilities in deployed smart contracts and exposing the defectiveness of the targeted applications. In 2010, 92 billion BTC were generated out of thin air by exploiting an integer vulnerability on the Bitcoin blockchain. One of the most infamous attacks on Ethereum was the one exploiting a reentrancy vulnerability in the DAO (a Decentralized Autonomous Organization built upon Ethereum) and resulting in 3.6M of stolen Ether. A vulnerable blockchain-based application does not have to be the target of a malicious attack to malfunction. For instance, the Parity multisig wallet was subject to an unintentional accident caused by a self-destruct vulnerability in 2017 and resulting in freezing 500K of Ether.
Given the importance of the assets circulating in each blockchain, securing the applications deployed on such distributed ledgers is considered mission-critical, and seeing that most of the vulnerabilities allowing the breaches are rooted in the smart contracts implementing the applications, verification of such pieces of code before deployment is crucial to secure the designed applications.
Informal as well as formal methods have been proposed to enhance the security of smart contracts and ensure their correctness. While informal techniques can test a smart contract under certain scenarios, they cannot be relied on to verify specific properties defining its correctness (e.g., absence of integer overflow vulnerabilities, deadlock-freedom) which is where formal techniques prove to be efficient. We note that in our work, we are interested in Ethereum smart contracts as it is currently the second largest cryptocurrency platform after Bitcoin besides being the inaugurator of smart contracts, and more particularly those written in Solidity [1] as it is the most popular language used by Ethereum. We also note that while Ethereum allows smart contracts to be written in a 'Turing complete' language that facilitates semantically richer applications than Bitcoin which allows very simple forms of smart contracts, the former also enlarges the threat surface, as evidenced by the many high-profile attacks.

In this paper, we propose an approach based on Coloured Petri Nets (CPNs) [14], for the formal verification of Solidity contracts. Our choice of this formalism is driven by its ability to combine the analysis power of Petri nets with the expressive power of programming languages, which makes it a suitable candidate for the modeling and verification of large and complex systems. The main idea is to transform a Solidity smart contract into a hierarchical CPN model depicting the functionality of the former. This model is then analyzed and the correctness

of the represented contract is proven via the verification of a set of temporal properties which are able to express vulnerabilities as well as contract-specific properties relevant to its data- and control-flows. We propose an algorithm that automates such a transformation, implement a prototype to prove its feasibility and leverage the *Helena* [11] tool to verify system properties.

2   Background

2.1   Solidity Smart Contracts

Solidity [1] is the most commonly-used high-level programming language for smart contracts under Ethereum, one of the leading public Blockchain platforms nowadays. The faithful execution of a smart contract is governed and guaranteed by the semantics of a *stack* machine called Ethereum Virtual Machine (EVM) and its immutable nature gives it a sense of finality.

A Solidity smart contract is a collection of code and data, residing at a specific address on the Ethereum Blockchain, which can be invoked either by an internal account (i.e., a smart contract) or directly by an external account (i.e., user). Every account is characterized by a persistent *storage* (null in case of an external account) and a balance in Ether which is adjusted by transactions. A transaction is a message used to send ether from one account to another and/or invoke a smart contract's function if the message includes a payload and the targeted account is an internal one. Every smart contract features a *memory* and can also access certain properties of the current block.

A Solidity smart contract may look like a JavaScript or C program syntax-wise, but they are actually dissimilar since the underlying semantics of Solidity functions differently from traditional programs. This naturally calls on more vigilance from the programmers who might be faced by unconventional security issues as vulnerabilities in smart contracts seem to often stem from this gap between the semantics of Solidity and the intentions of the programmer [4]. In the following, we present two use cases that we use throughout this paper and explain the vulnerabilities that we treat in this work to provide a better understanding of how we verify them later.

Use Cases
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 1.1) is based on a published contract[1] implementing a lottery game. It has been tweaked to illustrate more vulnerabilities without altering its purpose. A player participates in this game by sending an amount of ether equal to the *TICKET_AMOUNT* through *playTicket()*, which is then added to the game's *pot*. The winner is determined based on a

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

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 *getPot()* and the game's host (*bank*) can start a new round of lotto using *RestartLotto()*. This contract may seem fair to inexperienced Solidity developers, but it actually presents multiple vulnerabilities as we will later explain.

```
1   contract EtherLotto {
2       address public bank;
3       struct GameRecord {
4           address winner;
5           uint amount;
6       }
7       uint8 gameNum;
8       GameRecord[] LottoLog;
9       bool won;
10      uint constant TICKET_AMOUNT = 10;
11      uint constant FEE_AMOUNT = 1;
12      uint public pot;
13      function EtherLotto() {
14          bank = msg.sender;
15          won = false;
16          gameNum = 0;
17      }
18      function RestartLotto() {
19          require(msg.sender == bank);
20          require(won == true);
21          require(pot == 0);
22          won = false;
23          gameNum += 1;
24      }
25      function playTicket() payable {
26          require(msg.value == TICKET_AMOUNT);
27          require(won == false)
28          pot += msg.value;
29          uint random = uint(sha3(block.timestamp)) % 2;
30          if (random == 0) {
31              bank.call.value(FEE_AMOUNT)("");
32              won = true;
33              GameRecord gr;
34              gr.winner = msg.sender;
35              gr.amount = pot - FEE_AMOUNT;
36              LottoLog[gameNum] = gr;
37          }
38      }
39      function getPot() {
40          require(won == true);
41          if(msg.sender == LottoLog[gameNum].winner){
42              msg.sender.call.value(LottoLog[gameNum].amount)("
                    ");
43              pot = 0;
```

```
44          }
45      }
46  }
```

Listing 1.1. Solidity example: EtherLotto.sol

```
1   contract MaliciousContract {
2       uint ticket;
3       EtherLotto el = EtherLotto(0xbf0061dc...);
4       EtherMilestone em = EtherMilestone(0xc50164dfa...);
5       function playLotto() {
6           ticket = msg.value;
7           el.playTicket.value(ticket)();
8           el.getPot();
9       }
10      function playMilestone() {
11          em.play.value(1)();
12      }
13      function getRevenge ( ) {
14          selfdestruct(em);
15      }
16      function () payable {
17          el.getPot();
18      }
19    }
```

Listing 1.2. A malicious smart contract in Solidity

We consider a second Solidity example[2] (Listing 1.3) to emphasize on the harmful effect the self-destruction vulnerability (see next subsection) can have on a contract. It implements another gambling game whereby a player sends 1 ether to the contract by calling *play()* 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 *claimReward()*.

```
1   contract EtherMilestone {
2       uint public payoutMileStone1 = 6 ether;
3       uint public mileStone1Reward = 4 ether;
4       uint public payoutMileStone2 = 10 ether;
5       uint public mileStone2Reward = 6 ether;
6       uint public finalMileStone = 20 ether;
7       uint public finalReward = 10 ether;
8       mapping(address => uint) redeemableEther;
9       function play() public payable {
10          require(msg.value == 1 ether);
11          uint currentBalance = this.balance + msg.value;
12          require(currentBalance <= finalMileStone);
13          if (currentBalance == payoutMileStone1) {
14              redeemableEther[msg.sender] += mileStone1Reward;
```

---

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

```
15            }
16            else if (currentBalance == payoutMileStone2) {
17                 redeemableEther[msg.sender] += mileStone2Reward;
18            }
19            else if (currentBalance == finalMileStone ) {
20                 redeemableEther[msg.sender] += finalReward;
21            }
22            return;
23        }
24        function claimReward() public {
25            require(this.balance == finalMileStone);
26            require(redeemableEther[msg.sender] > 0);
27            redeemableEther[msg.sender] = 0;
28            msg.sender.call.value(redeemableEther[msg.sender])(""
                 );
29        }
30   }
```

Listing 1.3. Solidity example: EtherMilestone.sol

Vulnerabilities in Smart Contracts

In this section, we present the vulnerabilities [9] that we will be treating in our paper.

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 257th 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 1.2) 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 17 in Listing 1.1), the *EtherLotto*

contract 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 8 and 16. 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 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: 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 12 in Listing 1.1), 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.

---

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

2.2 Coloured Petri Nets

A Petri net [24] 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 [13], [29] 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) [14] 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 [23], 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. $\forall p \in P : E(p, t)\langle b \rangle \ll= M(p)$.
   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. $\forall p \in P :_{MS}^{++} \sum_{(t,b) \in Y} E(p, t)\langle b \rangle \ll= M(p)$
   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

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

3 Related Works

Existing studies on formal verification of smart contracts follow two main streams [12]. The first group of studies is based on theorem proving [6, 2]. The general idea here is to transform the contract's code (often its corresponding EVM bytecode) into a code processable by a theorem prover and use the latter to discharge proofs on the correctness of the generated code. In this case, the verification is not automated and requires the user's expertise to manipulate the prover and manually intervene in discharging proofs.

The second group of studies is based on model checking. Most of these studies use symbolic model checking coupled with complementary techniques such as symbolic execution [16] and abstraction [3]. The first attempt falling under this category is Oyente [19], a tool that targets 4 vulnerabilities, namely transaction order dependence, timestamp dependence, mishandled exceptions and reentrancy. It operates at the EVM bytecode level of the contract, generating symbolic execution traces and analyzing them to detect the satisfaction of certain conditions on the paths which indicates the presence of corresponding vulnerabilities. Numerous studies followed in the footsteps of this work, some of which exploited some of its components in their implementations, as is the case with GASPER [7] which reuses Oyente's generated control flow graph for the detection of bytecode patterns with high gas costs, and some others extended it with the aim of supporting the detection of other vulnerabilities, as is the case with Osiris [28]).

Also based on symbolic model checking, we find Zeus [15] which operates on the source code of the smart contract. The user needs to specify the criteria to be checked as a CHC-based policy written in XACML, according to which the code is instrumented before its translation into a low-level intermediate representation (LLVM bitcode) which is then fed to an existing verification engine.

VeriSolid [21], initially introduced as FSolidM in [20], is an FSM-based approach that unlike the previous ones, aims at producing a correct-by-design contract rather than detecting bugs. The authors propose a transformation of a contract modeled as an FSM into a Solidity code and provide the user with the ability to specify intended behavior in the form of liveness, deadlock freedom and safety

properties that can be expressed using templates for CTL properties and checked by a backend symbolic model checker.

More recently, attempts have been made to use Coloured petri net for the verification of smart contracts. The work in [18] shows an example of verification of behavioural properties on a CPN model for a crowdfunding smart contract. It does not, however, propose a complete and generic approach to apply on any smart contract as the work merely presents case study of a manual passage from the use case's contract to the CPN model without defining general rules for the transformation. Another CPN-based proposition was presented in [10]. The authors start from the contract's bytecode and use Hoare's logic to generate the corresponding CPN model which is then used for the security analysis of the contract. This approach, despite being based on CPN, cannot be used for the verification of data-flow related properties as the generated model focuses on the representation of the workflow extracted from the contract's CFG.

In order to use symbolic execution to generate the traces that would be used for the verification, the proposed approaches usually use under-approximation (e.g., in the form of loop bounds) which means that critical violations can be overlooked. This explains the presence of false negatives and/or positives in their reported results. We also note that most of the existing studies target specific vulnerabilities in smart contracts, and few are those that allow expressing customizable properties, in which case they are control flow-related properties. In fact, none of these studies target data-related properties. It is worth mentioning that most of the proposed approaches operate on the EVM bytecode rather than on the Solidity code because of the latter's lack of formal semantics. This, however, results in loss of contextual information, and consequently limits the range of properties that can be verified on the contract.

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

4 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

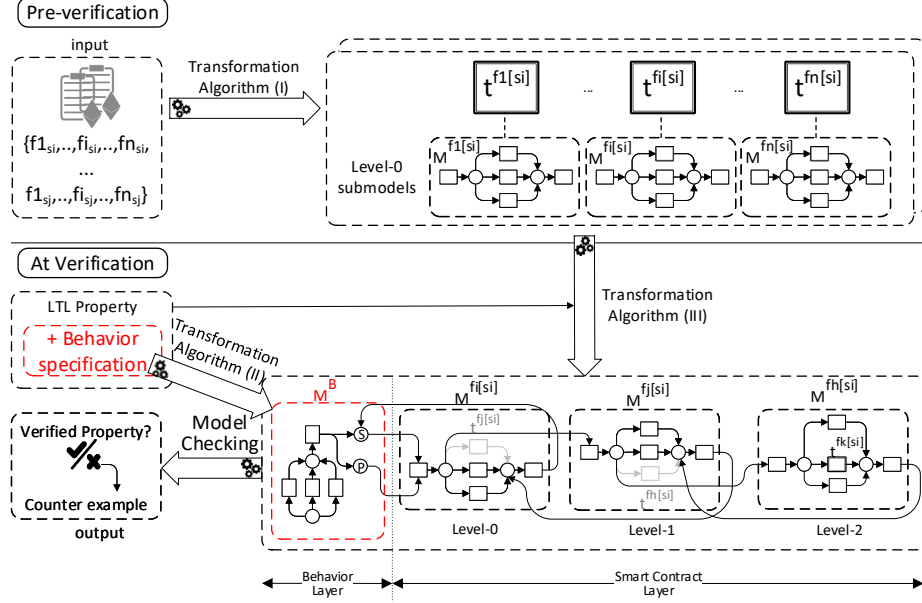provided behavior to be considered, and feeding it the model checking to verify the targeted property.

Fig. 1. Overview of the approach

More precisely, we opt for a hierarchical CPN model to represent the considered smart contracts' execution and interaction with respect to the provided behavior specification. As shown in Figure 1, 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 1) 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.

5   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.

5.1   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, $type_{v_1}$: stateVariable₁, ..., $type_{v_n}$: stateVariableₙ, $type_{p_1}$: inputParameter₁, ... , $type_{p_n}$: inputParameterₙ]* (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 *[$type_{p_1}$: inputParameter₁, ..., $type_{p_n}$: inputParameterₙ]*.
- *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, $type_{v_1}$: stateVariable₁, ..., $type_{v_n}$: stateVariableₙ]*
- a *parameters place* $p_p \in P_P$ representing the input parameters of the function in question.

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

Expressions E

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

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

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

Statements S

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

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

A control statement can be:

- a requirement statement of the form $require(c)$
- a selection statement which can have:
  - a single-branching form: $if(c) \; then \; st_T$
  - a double-branching form: $if(c) \; then \; st_T \; else \; st_F$
- a looping statement which can be:
  - a for loop: $for(init; c; inc) \; st_T$
  - a while loop: $while(c) \; st_T$

  – where:
  - c is a boolean expression
  - $c.vars$ designates the set of variables used in the condition (if $c \in E_V$)
  - $c.fctCalls$ designates the set of function calls used in the condition (if $c \in E_F$)
  - st_T, st_F, $init$ and inc are statements

5.2 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

```
 1: procedure GENERATEAGGREGATIONS(SC)
 2:     Input: a Solidity smart contract SC
 3:     Output: the aggregated transitions the CPN model of SC
 4:     metaColour ← [uint : contractBalance]
 5:     for v ∈ SC.vars do
 6:         add (v.type : v.name) to metacolour
 7:     end for
 8:     for f ∈ SC.fcts do
 9:         create aggregated transition tᵃ
10:         tᵃ.name ← f.name
11:         tᵃ.statement ← f.body
12:         newColour ← metaColour
13:         for p ∈ f.params do
14:             add (p.type : p.name) to newColour
15:         end for
16:         tᵃ.metaColour ← newColour
17:     end for
18: end procedure
```

GENERATELEVEL0 Algorithm

```
 1: procedure GENERATELEVEL0(tᵃ)
 2:     Input: an aggregated transition tᵃ
 3:     Output: the level-0 CPN submodel of tᵃ
 4:     P_data ← ∅
 5:     GETLOCALVARIABLES(tᵃ.statement; P_data)
 6:     tᵃ ← P_data
 7:     tᵃ.submodel ← CREATESUBMODEL(tᵃ, ∅, ∅)
 8: end procedure
```

GETLOCALVARIABLES creates a set of places to be used in the submodel of a transition t^a, corresponding to the local variables used in its function. To do so,

the statements in the function's body are recursively investigated in search for variable declaration statements. For each variable declaration statement found, a place bearing the name of the variable and its type as its name and colour is created and added to the set P_{data}. In addition to standalone variable declarations, we note that we can also find variables declared in the initialization of a For loop.

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

GETLOCALVARIABLES

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

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

CREATESUBMODEL Algorithm

1: **procedure** CREATESUBMODEL(t; st; p_{in}; p_{out})
2:     **Input:** transition t, statement st, control flow input place p_{in}, control flow output place p_{out}
3:     **Output:** submodel of transition t
4:     **switch** st **do**
5:         **case** compound statement $\{st[1]; st[2]; \ldots; st[N]\}$
6:             BUILDCOMPOUNDSTATEMENT$(t;st;p_{in};p_{out})$
7:     **case** simple statement
8:         **switch** st **do**
9:             **case** assignment statement
10:                 BUILDASSIGNMENTSTATEMENT$(t;st;p_{in};p_{out})$
11:             **case** variable declaration statement
12:                 BUILDVARIABLEDECLARATIONSTATEMENT$(tst;;p_{in};p_{out})$
13:             **case** sending statement
14:                 BUILDSENDINGSTATEMENT$(t;st;p_{in};p_{out})$
15:             **case** returning statement
16:                 BUILDRETURNINGSTATEMENT$(t;st;p_{in};p_{out})$
17:             **case** function call statement
18:                 BUILDFUNCTIONCALLSTATEMENT$(t;st;p_{in};p_{out})$
19:         **end switch**
20:     **case** control statement
21:         **switch** st **do**
22:             **case** requirement statement
23:                 BUILDREQUIREMENTSTATEMENT$(t;st;p_{in};p_{out})$
24:             **case** selection statement
25:                 BUILDSELECTIONSTATEMENT$(t;st;p_{in};p_{out})$
26:             **case** looping statement
27:                 **switch** st **do**
28:                     **case** for statement
29:                         BUILDFORLOOPSTATEMENT$(t;st;p_{in};p_{out})$
30:                     **case** while statement
31:                         BUILDWHILELOOPSTATEMENT$(t;st;p_{in};p_{out})$
32:                 **end switch**
33:         **end switch**

---

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

34:      **end switch**
35: **end procedure**

BUILDCOMPOUNDSTATEMENT Algorithm

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

BUILDASSIGNMENTSTATEMENT Algorithm

1: **procedure** BUILDASSIGNMENTSTATEMENT(t; st; p_{in}; p_{out})
2:      **Input:** transition t, an assignment statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
3:      **Output:** submodel for statement st
4:      create transition t'
5:      create arc from p_{in} to t'
6:      CONNECTLOCALVARIABLES$(st_{RHS}.vars\backslash\{st_{LHS}.vars\};t;t')$
7:      CONNECTFUNCTIONCALLS$(st_{RHS}.fctCalls;t)$
8:      **if** $st_{LHS}.vars$ is a local variable **then**
9:          create arc from $t.data[st_{LHS}.vars]$ to t'
10:          create arc from t' to $t.data[st_{LHS}.vars]$ with inscription st_{RHS}
11:          create arc from t' to p_{out}
12:      **else**
13:          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}
14:      **end if**
15: **end procedure**

BUILDVARIABLEDECLARATIONSTATEMENT Algorithm

1: **procedure** BUILDVARIABLEDECLARATIONSTATEMENT(t; st; p_{in}; p_{out})
2:      **Input:** transition t, a variable declaration statement $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in}, control flow output place p_{out}
3:      **Output:** submodel for statement st
4:      create transition t'
5:      create arc from p_{in} to t'

6:      CONNECTLOCALVARIABLES$(st_{RHS}.vars;t;t')$

7:      CONNECTFUNCTIONCALLS$(st_{RHS}.fctCalls;t)$

8:      create arc from t' to $t.data[st_{LHS}.vars]$ with inscription st_{RHS}

9:      create arc from t' to p_{out}

10: **end procedure**

BUILDSENDINGSTATEMENT Algorithm

1: **procedure** BUILDSENDINGSTATEMENT$(t; st; p_{in}; p_{out})$

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

3:      **Output:** submodel for statement st

4:      create transition t'

5:      create arc from p_{in} to t'

6:      CONNECTLOCALVARIABLES$(st_{RHS}.vars;t;t')$

7:      CONNECTFUNCTIONCALLS$(st_{RHS}.fctCalls;t)$

8:      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}

9: **end procedure**

BUILDRETURNINGSTATEMENT Algorithm

1: **procedure** BUILDRETURNINGSTATEMENT$(t; st; p_{in}; p_{out})$

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

3:      **Output:** submodel for statement st

4:      create transition t'

5:      create arc from p_{in} to t'

6:      CONNECTLOCALVARIABLES$(st_{RHS}.vars;t;t')$

7:      CONNECTFUNCTIONCALLS$(st_{RHS}.fctCalls;t)$

8:      create arc from t' to $t \bullet [cf]$

9:      create arc from t' to $t\bullet[output]$ with inscription $outInsc \leftarrow [inInsc.sender, inInsc.balance, st_{RHS}]$

10: **end procedure**

BUILDFUNCTIONCALLSTATEMENT Algorithm

1: **procedure** BUILDFUNCTIONCALLSTATEMENT$(t; st; p_{in}; p_{out})$

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

3:      **Output:** submodel for statement st

4:      create transition t^f

5:      create place p_{param_f}

6:      create arc from p_{in} to t^f

7:      create arc from p_{param_f} to t^f

8:      CONNECTLOCALVARIABLES$(f_{RHS}.vars;t;t^f)$

9:      CONNECTFUNCTIONCALLS$(f_{RHS}.fctCalls;t)$
10:     create arc from t^f to p_{out} with a placeholder inscription
11: **end procedure**

BUILDREQUIREMENTSTATEMENT Algorithm

1: **procedure** BUILDREQUIREMENTSTATEMENT(t; st; p_{in}; p_{out})
2:      **Input:** transition t, a requirement statement $st = require(c)$, control flow input place p_{in}, control flow output place p_{out}
3:      **Output:** submodel for statement st
4:      create transition t_{revert}
5:      $t_{revert}.guard \leftarrow !c$
6:      create arc from p_{in} to t_{revert}
7:      create arc from t_{revert} to $\bullet t[cf]$
8:      CONNECTLOCALVARIABLES$(c.vars;t;t_{revert})$
9:      CONNECTFUNCTIONCALLS$(c.fctCalls;t_{revert})$
10:     create transition $t_{!revert}$
11:     $t_{!revert}.guard \leftarrow c$
12:     create arc from p_{in} to $t_{!revert}$
13:     create arc from $t_{!revert}$ to p_{out}
14:     CONNECTLOCALVARIABLES$(c.vars;t;t_{!revert})$
15:     CONNECTFUNCTIONCALLS$(c.fctCalls;t_{!revert})$
16: **end procedure**

BUILDSELECTIONSTATEMENT Algorithm

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

21:        create arc from t_F to p_F

22:        CREATESUBMODEL$(t;st_F;p_F;p_{out})$

23:     **end if**

24: **end procedure**

BUILDFORLOOPSTATEMENT Algorithm

1: **procedure** BUILDFORLOOPSTATEMENT(t; st; p_{in}; p_{out})

2:     **Input:** transition t, a *for* looping statement $st = for(init; c; inc)\ st_T$, control flow input place p_{in}, control flow output place p_{out}

3:     **Output:** submodel for statement st

4:     create place p_{init}

5:     create place p_c

6:     create place p_T

7:     CREATESUBMODEL$(t;init;p_{in};p_{init})$

8:     create transition t_T

9:     $t_T.guard \leftarrow c$

10:    create arc from p_{init} to t_T

11:    CONNECTLOCALVARIABLES$(c.vars;t;t_T)$

12:    CONNECTFUNCTIONCALLS$(c.fctCalls;t_T)$

13:    create arc from t_T to p_c

14:    create transition t_F

15:    $t_F.guard \leftarrow !c$

16:    create arc from p_{init} to t_F

17:    CONNECTLOCALVARIABLES$(c.vars;t;t_F)$

18:    CONNECTFUNCTIONCALLS$(c.fctCalls;t_F)$

19:    create arc from t_F to p_{out}

20:    CREATESUBMODEL$(t;st_T;p_c;p_T)$

21:    CREATESUBMODEL$(t;inc;p_T;p_{init})$

22: **end procedure**

BUILDWHILELOOPSTATEMENT Algorithm

1: **procedure** BUILDWHILELOOPSTATEMENT(t; st; p_{in}; p_{out})

2:     **Input:** transition t, a *while* looping statement $st = while(c)\ st_T\ st_T$, control flow input place p_{in}, control flow output place p_{out}

3:     **Output:** submodel for statement st

4:     create place p_T

5:     create transition t_T

6:     $t_T.guard \leftarrow c$

7:     create arc from p_{in} to t_T

8:     CONNECTLOCALVARIABLES$(c.vars;t;t_T)$

9:     CONNECTFUNCTIONCALLS$(c.fctCalls;t_T)$

10:    create arc from t_T to p_T

11:    create transition t_F

12:    $t_F.guard \leftarrow !c$

13:    create arc from p_{in} to t_F

14:      CONNECTLOCALVARIABLES$(c.vars;t;t_F)$
15:      CONNECTFUNCTIONCALLS$(c.fctCalls;t_F)$
16:      create arc from t_F to p_{out}
17:      CREATESUBMODEL$(t;st_T;p_T;p_{in})$
18: **end procedure**

CONNECTLOCALVARIABLES Algorithm

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

CONNECTFUNCTIONCALLS Algorithm

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

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

UNFOLDTRANSITION Algorithm

 1: **procedure** UNFOLDTRANSITION(t^a;p_{in};p_{out})
 2:      **Input:** aggregated transition t^a, input place p_{in}, output place p_{out}
 3:      **Output:** submodel replacement of transition t^a
 4:      **for** $t' \in t^a.submodel.inTransition$ **do**
 5:          replicate (arc from p_{in} to t^a) to t'

6:           replicate (arc from $\bullet t[input]$ to t^a) to t'

7:         **for** $p \in \bullet t^a[data] \cup \bullet t^a[output]$ **do**

8:            replicate (arc from p to t^a) to t'

9:         **end for**

10:      **end for**

11:      **for** $t' \in t^a.submodel.outTransition$ **do**

12:         replicate (arc from t^a to p_{out}) to t' with the placeholder inscription replaced by values from $\bullet t'[cf]$

13:      **end for**

14:      hide transition t^a and all arcs linked to it

15: **end procedure**

5.3   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 [8, 22, 25] 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 2) 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.

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 6-10) and DCR graphs
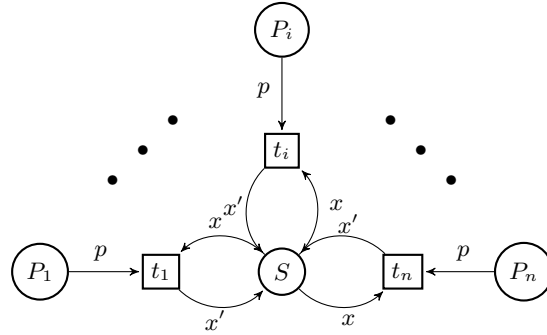
Fig. 2. CPN model for a completely-free behavior

(Definitions 11-14), then we define our proposed model (Definition 15). We then elaborate a theorem regarding the semantic equivalence between our model and DCR graphs (Theorem 3 and Proof 5.3).

   Coloured Petri Net

A Petri net [24] 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 [13], [29] 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) [14] 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 [23], to define data types.

Definition 6 (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 7 (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 8 (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. $\forall p \in P : E(p, t)\langle b \rangle \ll= M(p)$.
   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 9 (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. $\forall p \in P :_{MS}^{++} \sum_{(t,b) \in Y} E(p,t)\langle b \rangle \ll = M(p)$
   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 10 (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 2. *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

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

Dynamic Condition Response Graphs

Definition 11. *A dynamic condition response graph is a tuple $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond, l)$ 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. *$\rightarrow\bullet \subseteq E \times E$ is the condition relation*
5. *$\bullet\rightarrow \subseteq E \times E$ is the response relation*
6. *$\rightarrow+, \rightarrow\% \subseteq E \times E$ is the dynamic include relation and exclude relation satisfying that $\forall e \in E.e \rightarrow+ \cap e \rightarrow\% = \emptyset$*
7. *$\rightarrow\diamond \subset E \times E$ is the milestone relation*
8. *$l : E \rightarrow Act$ 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) $\rightarrow+$ and $\rightarrow\%$ 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 $e \rightarrow+ e'$ means that, whenever e is executed, e' becomes included in the graph if it is not already. The exclusion relation $e \rightarrow\%e'$ 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 $(b \rightarrow\diamond a)$, 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 12 (Enabled event). *For a dynamic condition response graph $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond, l)$ 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. *$(\rightarrow\bullet e \cap In) \in Ex$*
3. *$(\rightarrow\diamond e \cap In) \in E \setminus Re$*

Definition 13 (Event execution effect). *For a dynamic condition response graph $G = (E, M, Act, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond, l)$ 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 14 (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*
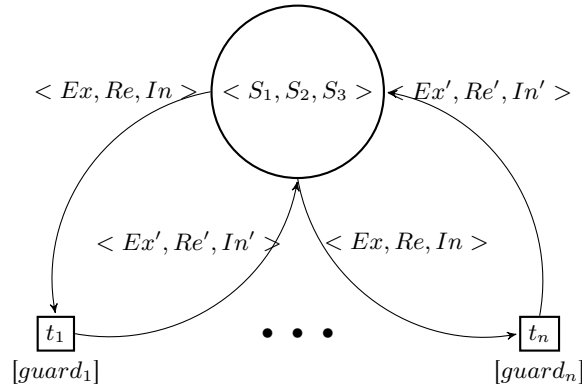
1. *$M = M_0$*
2. *$\forall i \in [k].a_i \in l(e_i)$*
3. *$\forall i \in [k].M_i \vdash_G e_i$*

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 $\forall i \in [k].(\forall e \in In_i \cap Re_i.\exists j \geq i.e_j = e \lor e \notin In'_j)$, 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 $\mathcal{M}_{M \to^*}(G)$ denote the set of all reachable markings from M.

CPN4DCR Model

Fig. 3. CPN model for a DCR Graph

Definition 15 (CPN4DCR). *Given a dynamic condition response graph $G = (E, M, Act, \to\bullet, \bullet\to, \pm, l)$, 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* $Type[v] = C_E, \forall v \in V$
- $C = \{S \to (C_E \times C_E \times C_E)\}$
- $G = \{t_i \to guard_i, \forall i \in [1, n]\}$, *with* $n = |E|$
- $E = \{a \to< Ex, Re, In >, \forall a \in A \cap (P \cup T)\} \cup \{a \to< Ex', Re', In' >, \forall a \in A \cap (T \cup P)\}$ *with*
    - $Ex' = Ex \cup e_i$
    - $Re' = (Re \backslash e_i) \cup e\bullet\to$

- $In' = (In \cup e_i \rightarrow +) \backslash e \rightarrow \%$
  - $I = \{S \rightarrow < S_1, S_2, S_3 >\}$ 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 16 (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 17 (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 3. *Let G be a DCR graph and C the corresponding CPN model generated by following definition 15, then G and C are semantically equivalent.*

Proof. Let G be a DCR graph and C the corresponding CPN model generated by following definition 15. 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 15. 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:

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

And therefore:

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

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

6   Formal Verification of Smart Contracts

In our proposed verification approach, we rely on *Helena* in the model checking phase to verify LTL properties [26] that express the susceptibility of contracts to vulnerabilities. To this aim, we start by expressing each targeted vulnerability in LTL (see Section 2.1 for the explanation of these vulnerabilities).

6.1   Expressing Vulnerabilities in LTL

In the following, $t_{s_i}^f$ 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:

$$IUO(t_{s_i}^f) = \Box \neg outOfRange(x)$$

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:

$$Reentrancy(t_{s_i}^f) = containsSending(t_{s_i}^f) \rightarrow \Box \neg reentrant$$

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

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

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

$$NoReentrancy(t_{s_i}^f) = containsSending(t_{s_i}^f) \rightarrow (sendingTo(s_j) \rightarrow$$
$$\bigcirc \Box((\neg sendingTo(s_j)) \cup end(t_{s_j}^{fallback})))$$

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:

$$selfDestruction(t_{s_i}^f) = \neg testOnBalance(t_{s_i}^f)$$

This detection process can be further enhanced when the code of the interacting smart contract is available. In that case, a function f in s_i is considered safe against this vulnerability if it does not contain a test on *this.balance* or if the interacting contract s_j does not contain a self destruction instruction or if

the latter cannot be executed prior to the function under inspection, which is expressed by the following LTL property:

$$selfDestruction(t_{s_i}^f) = \neg(testOnBalance(t_{s_i}^f) \wedge containsSelfDestruct(t_{s_j}^g, s_i))$$
$$\vee \, (\neg selfDestruct(t_{s_j}^g, s_i) \cup start(t_{s_i}^f))$$

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*:

$$TSD(t_{s_i}^f) = \Box \neg isTimestampDependant$$

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

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

$$SkipEmpty(t_{s_i}^f) = containsFunctionCall(t_{s_i}^f) \rightarrow (\Box \neg (isFunctionCall$$
$$\wedge \, \exists arg \in functionCall \backslash (arg = \text{""} \wedge \neg isLast(arg)))$$

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

$$UninitializedVariable(t_{s_i}^f) = isOfComplexType(x) \rightarrow (\neg read(x) \cup write(x))$$

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

6.2   Model Checking of Smart Contracts

The application of our approach on the Blind Auction use case presented in Section **??** yields a hierarchical CPN model whose level-0 submodels are shown in Figures 4-6. These submodels correspond to the transitions representing functions in the Blind Auction smart contract (Listing **??**), 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}.

Figure 7 represents the resulting submodel obtained by applying our transformation algorithm on the function *play()* of the *EtherMilestone* smart contract.
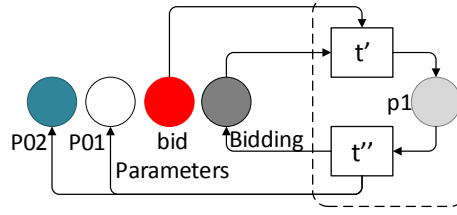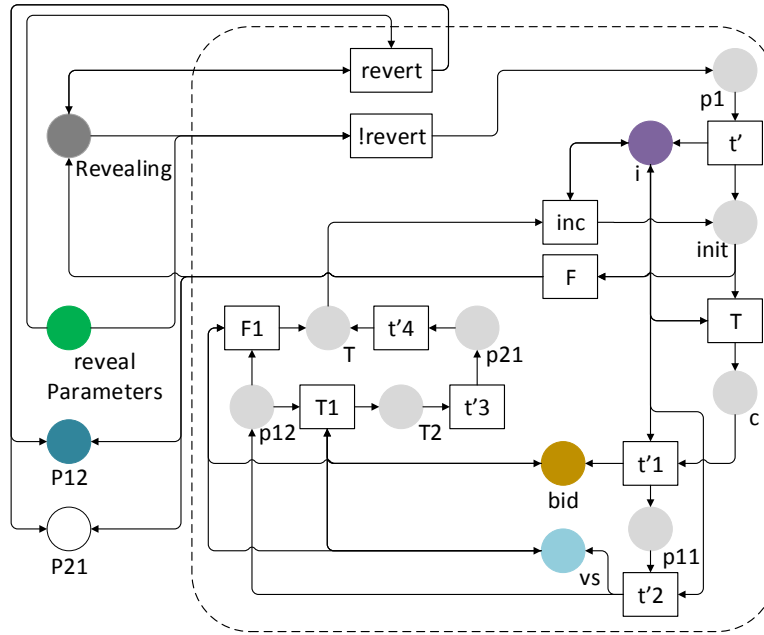
Fig. 4. SubModel of transition *bid*

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* [11] 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.

We have generated the CPN models of our use cases for *Helena* using our prototype for the transformation algorithm (see Figure 7 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 2.1. 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 8 shows the corresponding property written in *Helena* for the *IUO* LTL property applied on the variable *gameNum* in *EtherLotto* and Figure 9 is a snippet of the result of the model checker showing the detection of the vulnerability and the indication of a counter example.

7 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
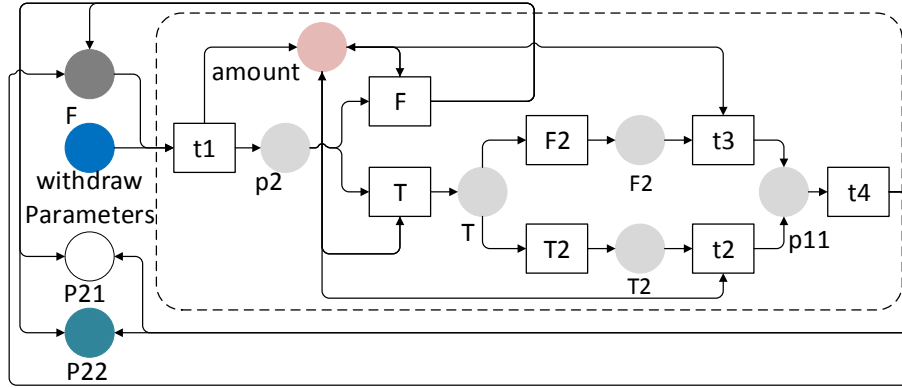
Fig. 5. SubModel of transition *reveal*

corresponding properties on the smart contract, unrestrictedly to certain pre-
defined vulnerabilities. Besides, our approach takes into account the context in
which the smart contracts to be verified are executed as a behavior specifica-
tion, 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 [17]
and applying it on CPNs.

References

1. Solidity 0.6.11 documentation. https://solidity.readthedocs.io/en/v0.6.11/index.html
2. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum
   smart contract bytecode in isabelle/hol. In: Proceedings of the 7th ACM SIG-
   PLAN International Conference on Certified Programs and Proofs. p. 66–77.
   CPP 2018, Association for Computing Machinery, New York, NY, USA (2018).
   https://doi.org/10.1145/3167084
3. Anand, S., Pasareanu, C.S., Visser, W.: Symbolic execution with abstraction. Int. J.
   Softw. Tools Technol. Transf. **11**(1), 53–67 (2009). https://doi.org/10.1007/s10009-
   008-0090-1
4. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart con-
   tracts (sok). In: Principles of Security and Trust - 6th International Conference,

Fig. 6. SubModel of transition *withdraw*
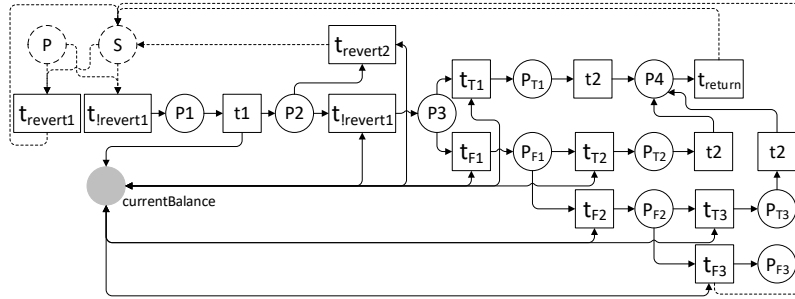
Fig. 7. CPN submodel of the *play()* function in *EtherMilestone*

POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings

5. Berger, C., Penzenstadler, B., Drögehorn, O.: On using blockchains for safety-critical systems. In: Bures, T., Fitzgerald, J.S., Schmerl, B.R., Weyns, D. (eds.) Proceedings of the 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems, ICSE 2018, Gothenburg, Sweden, May 27, 2018. pp. 30–36. ACM (2018)

6. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Béguelin, S.Z.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Secu-

Fig. 8. The integer overflow/underflow LTL property in *Helena*

Fig. 9. Model checking result

rity, PLAS@CCS 2016, Vienna, Austria, October 24, 2016. pp. 91–96 (2016). https://doi.org/10.1145/2993600.2993611

7. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017. pp. 442–446 (2017). https://doi.org/10.1109/SANER.2017.7884650

8. 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)

9. 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)

10. Duo, W., Huang, X., Ma, X.: Formal analysis of smart contract based on colored petri nets. IEEE Intell. Syst. **35**(3), 19–30 (2020)

11. 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)

12. Garfatta, I., Klai, K., Gaaloul, W., Graiet, M.: A survey on formal verification for solidity smart contracts. In: ACSW '21: 2021 Australasian Computer Science Week Multiconference, Dunedin, New Zealand, 1-5 February, 2021. pp. 3:1–3:10. ACM (2021)

13. 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)

14. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer Publishing Company, Incorporated, 1st edn. (2009)

15. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018 (2018)

16. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as

Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings. pp. 553–568 (2003). https://doi.org/10.1007/3-540-36577-X_40

17. 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

18. Liu, Z., Liu, J.: Formal verification of blockchain smart contract based on colored petri net models. In: Getov, V., Gaudiot, J., Yamai, N., Cimato, S., Chang, J.M., Teranishi, Y., Yang, J., Leong, H.V., Shahriar, H., Takemoto, M., Towey, D., Takakura, H., Elçi, A., Takeuchi, S., Puri, S. (eds.) 43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 2. pp. 555–560. IEEE (2019)

19. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 254–269 (2016). https://doi.org/10.1145/2976749.2978309

20. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: A finite state machine based approach. In: Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers. pp. 523–540 (2018). https://doi.org/10.1007/978-3-662-58387-6_28

21. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. In: Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers. pp. 446–465 (2019). https://doi.org/10.1007/978-3-030-32101-7_27

22. 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)

23. Milner, R., Tofte, M., Harper, R.: Definition of standard ML. MIT Press (1990)

24. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)

25. 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 FO-CLASA, Oslo, Norway, September 16-20, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12226, pp. 85–100. Springer (2019)

26. Pnueli, A.: The temporal semantics of concurrent programs. Theor. Comput. Sci. **13** (1981)

27. Staples, M., Chen, S., Falamaki, S., Ponomarev, A., Rimba, P., Tran, A., Weber, I., Xu, S., Zhu, J.: Risks and opportunities for systems using blockchain and smart contracts. In: Data61 (CSIRO), Sydney (2017)

28. Torres, C.F., Schütte, J., State, R.: Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 664–676 (2018). https://doi.org/10.1145/3274694.3274737

29. 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)