

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

Ikram Garfatta^{*†}, Kaïs Klai[†], Mohamed Graïet^{‡§} and Walid Gaaloul[¶]

^{*}University of Tunis El Manar, National Engineering School of Tunis, OASIS
Tunis, Tunisia

Email: ikram.garfatta@lipn.univ-paris13.fr

[†]University Sorbonne Paris North, LIPN UMR CNRS 7030
Villetaneuse, France

[‡]University of Monastir, Higher Institute for Computer Science and Mathematics
Monastir, Tunisia

[§]National School for Statistics and Information Analysis
Bruz, France

[¶]Institut Mines-Télécom, Télécom SudParis, SAMOVAR UMR 5157
Évry, France

Abstract—While Blockchains can open intriguing opportunities of research in many application contexts, they come with the risk of bringing new unconventional problems. In fact, because of the monetary value they hold, Blockchains have been subject to many attacks. Smart contracts, which are at the core of second-generation Blockchains, have been proven to be the origin of such attacks due to the exploitable vulnerabilities their code may hold. It is therefore an essential requirement to prove the correctness of the smart contracts to be deployed on a Blockchain to ensure its protection. The existing approaches have been focusing on targeting generic vulnerabilities like reentrancy, without offering the possibility to check temporal-based contract-specific properties. In this paper, we aim to address smart contracts verification while supporting such properties. We propose and implement a transformation of Solidity smart contracts into Coloured Petri nets and investigate the capability of existing model checking tools to check specific temporal properties of the formally modeled contract.

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

1. Introduction

The reach of the Blockchain technology has expanded to a myriad of application domains such as healthcare, insurance, Business Process Management, etc. Such an expansion is owed to Blockchain's inherent characteristics, namely its decentralized nature, ability to provide trust among trustless parties, immutability and financial transparency.

Blockchain is still considered an evolving technology whose extent has not been fully revealed. Working towards new ways of exploiting it, though can lead to valuable exploits, is undoubtedly a risky endeavor. In fact, smart contracts can turn into a weak spot in this context. As a

smart contract cannot be altered once it has been deployed on the Blockchain, it goes without saying that it cannot be corrected either, which makes verifying its correctness prior to its deployment an indispensable necessity. Many attacks on multiple Blockchain platforms have stemmed from smart contract vulnerabilities, like the attack exploiting an integer overflow vulnerability on the Bitcoin blockchain in August 2010 and the DAO attack exploiting a reentrancy vulnerability on Ethereum in June 2016.

Researchers have quickly started to address such problems by proposing informal as well as formal methods 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 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.

In this paper, we propose an approach based on Coloured Petri Nets (CPNs) [2], for the formal verification of Solidity contracts [3]. 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 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 existing tools, namely *CPN Tools* [4] and *Helena* [5] to verify system properties.

Our paper is structured as follows. Section 2 presents a use case, Section 3 provides essential prerequisites and Section 4 presents the related works. Our contribution is introduced in Section 5 and our verification results are presented in Section 6 followed by a conclusion and future perspectives in Section 7.

2. Use Case: Blind Auction

Our use case is adapted from an example in [1]. Participants in a blind auction have a bidding window during which they can place their bids. A participant can place more than one bid and the placed bid is blinded. The bidder has to make a deposit with the blinded bid, with a value that is supposedly greater than the real bid. Once the bidding window is closed, the revealing window is opened. Participants proceed to reveal their bids by sending the actual values of the bids along with the used keys. The system verifies whether the sent values correspond with the placed blinded bids and potentially updates the highest bid and bidder's values. If the revealed value of a bid does not correspond with its blinded value, or is greater than the deposit, the said bid is considered invalid. Once the revealing window is closed, participants can proceed to withdraw their deposits. A deposit made along a non-winning, invalid or unrevealed bid is wholly restored. In case of a winning bid, the difference between the deposit and the real bid is restored. The auction is terminated when all participants withdraw their deposits. Figure 1 describes the workflow of the blind auction system and Listing 1 represents an excerpt of the Solidity smart contract implementing it.

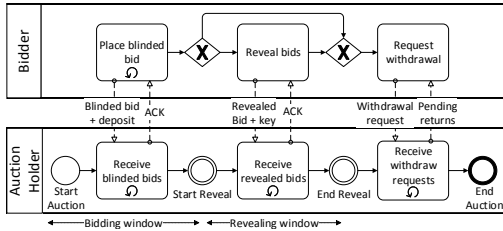


Figure 1. Blind Auction Workflow

```
contract BlindAuction {
    struct Bid {bytes32 blindedBid; uint deposit;}
    uint public biddingEnd, revealEnd;
    mapping(address => Bid[]) public bids;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;
    modifier onlyBefore(uint _time) {
        require(now<_time);;}
    modifier onlyAfter(uint _time) {
        require(now>_time);;}
    constructor(uint _biddingTime, uint
        _revealTime) public {...}
    function bid(bytes32 _blindedBid) public
        payable onlyBefore(biddingEnd) {...}
    function reveal(uint[] values, bytes32[]
        secrets) public onlyAfter(biddingEnd)
        onlyBefore(revealEnd) {...}
    function withdraw() public onlyAfter
        (revealEnd) {
        uint amount = pendingReturns[msg.sender];
```

```
if (amount > 0) {
    if (msg.sender != highestBidder)
        msg.sender.transfer(amount);
    else
        msg.sender.transfer(amount -
            highestBid);
    pendingReturns[msg.sender] = 0;}}
```

Listing 1. Excerpt of the Blind Auction smart contract in Solidity

3. Preliminaries on Coloured Petri Net

A Petri net [6] is a formal model with mathematics-based execution semantics. It is a directed bipartite graph with two types of nodes: places (drawn as circles) and transitions (drawn as rectangles). Despite its efficiency in modelling and analysing systems, a basic Petri net falls short when the system is too complex, especially when representation of data is required. To overcome such limitations, extensions to basic Petri nets were proposed, equipping the tokens with colours or types and hence allowing them to hold values. A large Petri net can therefore be represented in a much more compact manner using a *Coloured Petri net* [2].

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

- 1) P is a finite set of *places*.
- 2) T is a finite set of *transitions* such that $P \cap T = \emptyset$.
- 3) $A \subseteq (P \times T) \cup (T \times P)$ is a set of directed *arcs*.
- 4) Σ is a finite set of non-empty *colour sets*.
- 5) V is a finite set of *typed variables* such that $Type[v] \in \Sigma$ for all variables $v \in V$.
- 6) $C : P \rightarrow \Sigma$ is a *colour set function* that assigns a colour set to each place.
- 7) $G : T \rightarrow EXPR_V$, 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) $E : A \rightarrow EXPR_V$ 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) $I : P \rightarrow EXPR_{\emptyset}$ is an *initialisation function* that assigns an initialisation expression to each place p such that $Type[I(p)] = C(p)_{MS}$.

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 consuming (resp. adding), from each input place (resp. to each output place), the multiset of tokens corresponding to the input (resp. output) arc inscription (see [2] for more details).

4. Related Works

Existing studies on formal verification of smart contracts follow two main streams [7]: The first is based on theorem

proving [8], [9]. Approaches based on this technique cannot be fully automated as the user usually has to intervene to assist the prover. The second includes studies based on model checking, which is where our work can be situated. Most of the studies under this second category use symbolic model checking coupled with complementary techniques such as symbolic execution and abstraction. The first attempt was Oyente [10], a tool that targets 4 vulnerabilities, namely transaction order dependence, timestamp dependence, mis-handled 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 (e.g., GASPER [11]), while others extended it to support the detection of other vulnerabilities (e.g., Osiris [12]). VeriSolid [13] is an FSM-based approach that aims at producing a correct-by-design contract rather than detecting bugs. 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 contracts, and few are those that allow expressing customizable control flow-related properties while none target data-related properties.

More recently, attempts have been made to use CPN for the verification of smart contracts. The work in [14] shows an example of verification of behavioural properties applied manually on a CPN model for a case study of a crowdfunding smart contract. It does not, however, propose a complete approach with generic transformation rules that can be automated and applied to any smart contract. Another CPN-based proposition was presented in [15]. 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.

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

5. Solidity-to-CPN Transformation

We propose an algorithm that transforms a Solidity smart contract into a hierarchical CPN model over which CTL properties can be verified (cf. Section 6). The general idea is to start from a CPN model (referred to as level-0 model) representing the general workflow of the smart contract and then to build on it by embedding it with submodels (referred to as level-1 models) representing the execution of the smart

contract functions. We note that from a smart contract, only the functions' internal workflow can be deduced since the code defines the functions but not the way they are intended to be used. In this work, we use the *level-0 model* as a representation of this missing information and focus on the construction of the functions' corresponding submodels.

In a level-0 model, we distinguish 2 parts, namely the *user's behaviour* part which models the way users can interact with the system and the *smart contract's behaviour* part which represents the system. These two are linked via *communication places*. Figure 2 shows the level-0 model of the blind auction use case.

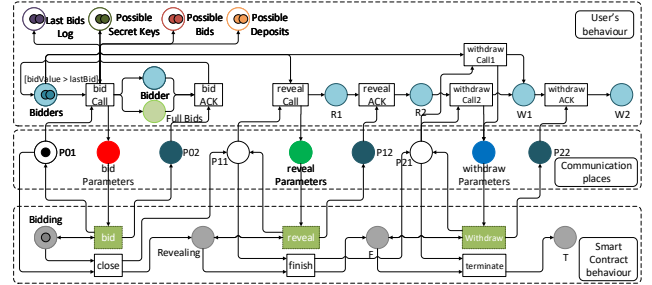


Figure 2. Blind Auction - Level-0 Model

5.1. Notations on the Model's Elements

5.1.1. Places P . In the Smart Contract's Behaviour part, we define *state places* P_S as places that hold the state of the smart contract, namely the contract's balance and the values of the state (global) variables. Their colour is as follows [*uint: contractBalance, type₁: stateVariable₁, ..., type_n: stateVariable_n*]

Among communication places, we distinguish:

- *parameter places* P_P : that convey potential inputs of function calls along with the caller's information and the transferred value. Their colour is as follows [*address: sender, uint: balance, uint: value, type₁: inputParameter₁, ..., type_n: inputParameter_n*]
- *return places* P_R : that communicate potential returned data from functions along with the caller's updated information. Their colour is as follows [*address: sender, uint: balance, type_R: returnParameter*]
- *interface places* P_I : that handle handovers between the function calls and their execution. They are uncoloured (basic) places.

On level 1 (a transition's detailed sub-model), we distinguish 2 types of places:

- *control flow places* P_{cf} . Their colour is a *metaColour* defined at each transition of level 0 as the concatenation of the colour of its input control flow place $\bullet t[cf] \in P_S$ and the colour of its input parameters place $\bullet t[input] \in P_P$.
- *data places* P_{data} where each place is of a colour corresponding to the represented variable's type.

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

- $t[cf] \in P_{cf} \cup P_S$, the input control flow place of t
- $t[input] \in P_P$, the input parameters place of t
- $t[com] \in P_C$, the input communication place of t
- $t[data] \subseteq P_{data}$, the input data places of t (in case of a submodel transition)
- $t[cf] \in P_{cf} \cup P_S$, the output cf place of t
- $t[output] \in P_R$, the output return place of t
- $t[data] \subseteq P_{data}$, the output data places of t (in case of a submodel transition)
- $t[com] \in P_C$, the output communication place

5.1.3. Statements S . We see a smart contract as a set of statements. A statement can be either a compound, a simple or a control one. A simple statement can be an assignment, a variable declaration, a sending or a returning statement. A control statement can be a requirement, a selection or a loop (a *for* or a *while* loop).

A complete list of the elements is provided on [github](https://github.com)¹.

5.2. Transformation Algorithm

Due to the lack of space, we only present an excerpt of the main algorithm responsible for the generation of submodels and the transformation algorithms for a compound and an assignment statements.

Algorithm 1: CREATESUBMODEL(t ; st ; p_{in} ; p_{out} ; P_{data} ; mc)

Input : t , statement st , cf input place p_{in} , cf output place p_{out} , set of internal data places P_{data} , meta colour mc

Output: submodel of transition t

```

1 set default place colour to  $mc$ 
2 switch  $st$  do
3   case compound  $st \{st[1]; st[2]; \dots; st[N]\}$  do
4     BUILDCOMPOUNDST( $t; st; p_{in}; p_{out}; P_{data}; mc$ )
5   end case
6   case simple  $st$  do
7     switch  $st$  do
8       case assignment  $st \{st_{LHS}, st_{RHS}\}$  do
9         BUILDASSIGNMENTST( $t; st; p_{in}; p_{out}; P_{data}; mc$ )
10      end case
11      ...
12    end switch
13  end case
14 end switch

```

CREATESUBMODEL browses the body of the transition's corresponding function recursively, statement by statement, and creates snippets of a CPN model, according to the type of the processed statement, that interconnect to create the transition's submodel. The full algorithms and descriptions can be found in the *Solidity2CPN* document available at

this repository¹ along with the algorithms responsible for connecting the submodels to the level-0 model.

Algorithm 2: BUILDCOMPOUNDST(t ; st ; p_{in} ; p_{out} ; P_{data} ; mc)

Input : t , compound $st \{st[1]; st[2]; \dots; st[N]\}$, cf input place p_{in} , cf output place p_{out} , set of internal data places P_{data} , meta colour mc

Output: snippet of statement st

```

1 for  $i = 1..N - 1$  do
2   create place  $p_i$ 
3 end for
4 CREATESUBMODEL( $t, st[1], p_{in}, p_1, P_{data}, mc$ )
5 for  $i = 2..N - 1$  do
6   CREATESUBMODEL( $t, st[i], p_{i-1}, p_i, P_{data}, mc$ )
7 end for
8 CREATESUBMODEL( $t, st[N], p_{N-1}, p_{out}, P_{data}, mc$ )

```

Compound statement $\{st[1]; st[2]; \dots; st[N]\}$: the algorithm is re-executed on each component statement $st[i]$, after creating $N - 1$ control flow places (of the *metaColour* colour) to interconnect the resulting CPN snippets while merging the entering point of the snippet of $st[1]$ with the entering point of the snippet of st and the exiting point of $st[N]$ to that of the snippet of st .

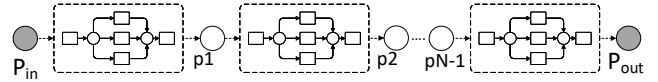


Figure 3. Compound Statement Pattern

Algorithm 3: BUILDASSIGNMENTST(t ; st ; p_{in} ; p_{out} ; P_{data} ; mc)

Input : t , assignment $st \{st_{LHS}, st_{RHS}\}$, cf input place p_{in} , cf output place p_{out} , set of internal data places P_{data} , meta colour mc

Output: submodel of transition t

```

1 create transition  $t'$  and arc from  $p_{in}$  to  $t'$  for
   $v \in st_{RHS}.vars \setminus \{st_{LHS}.var\}$  do
2   create arcs  $P_{data}[v] \rightarrow t'; t' \rightarrow P_{data}[v]$ 
3 end for
4 if  $st_{LHS}.var$  is a local variable then
5   create arcs  $P_{data}[st_{LHS}.var] \rightarrow t'; t' \rightarrow p_{out};$ 
   $t' \rightarrow P_{data}[st_{LHS}.var]$  with inscription  $st_{RHS}$ 
6 else
7   create arc  $t' \rightarrow p_{out}$  with inscription  $outInsc \leftarrow$ 
   $inInsc$  in which the variable corresponding to
   $st_{LHS}.var$  is replaced by  $st_{RHS}$ 
8 end if

```

Assignment statement (st_{LHS}, st_{RHS}) : a transition t' is created with input and output links to, respectively, the input

1. <https://github.com/Sol2CPN/WETICE2021>

(p_{in}) and output (p_{out}) places. t' is connected to the places in P_{data} that correspond to the variables used in the statement's RHS with input/output links (to read the data). In case of a local variable assignment (Figure 4), an input/output link is created with the place corresponding to the assigned variable in the statement's LHS with the new value (st_{RHS}) inscribed on the output link. In case of a state variable assignment, the new value (st_{RHS}) is given in the variable's corresponding placement on the link to the output (p_{out}) place.

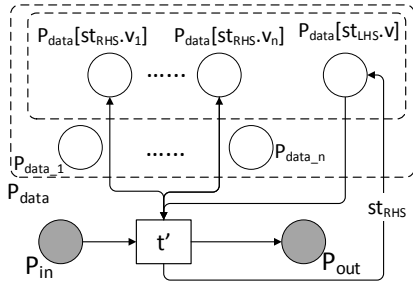


Figure 4. Local assignment statement pattern

5.3. Application on the Use Case

The application of the algorithm on the level-0 model of the Blind Auction use case (see Figure 2) yields a hierarchical CPN model whose level-1 submodels are created by the execution of `CREATESUBMODEL`. Figure 5 shows the submodel for transition *withdraw*.

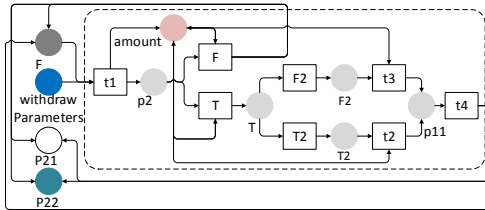


Figure 5. SubModel of transition *withdraw*

The light-grey-coloured places represent control flow places P_{cf} of a *metaColour* specific to the submodel, whereas places of other colours inside the dashed-line box are places of the relative P_{data} . Places outside the box are the input/output places of the corresponding level-0 transition, among which the dark grey places are state places in P_S , uncoloured places are interface places in P_I , dark blue places are return places in P_R and the others (blue for *withdraw*) are parameter places in P_P . The submodels inside the boxes are the product of `CREATESUBMODEL`.

The rest of the submodels can be found on [github](https://github.com)¹.

6. Smart Contract Verification

6.1. State Space Analysis Results

In Table 1, we present the results of the state space analysis of our designed model, rendered by both *CPN Tools*

and *Helena*, for different initial marking values.

We note that the state space generated for each scenario by both tools has the same size, both with and without considering hierarchy (noted *H* and *NH*), which validates the conformance of the model written in *Helena*'s language with that which was visually designed using *CPN Tools*. We can clearly see that *Helena* (the highlighted columns) outperforms *CPN Tools* (the non-highlighted columns) with a considerably shorter execution time.

6.2. Temporal Properties Verification

6.2.1. Using CPN Tools. Using the *ASK-CTL* library we were able to formulate a number of temporal properties. For instance, we defined a *termination* property to check that all the dead markings in the state space correspond to final markings of the model, and therefore check that the model is deadlock-free. To do so, we characterize the *termination* property as the model's capability to always reach a terminal state (a dead marking) where the following conditions are met: (C1) in the Smart Contract's Behaviour part, all places have to be empty except for *T* that must contain one token, (C2) all Communication places must be empty and (C3) in the User's Behaviour part: (C31) the markings of places *PossibleSecretKeys*, *PossibleBids* and *PossibleDeposits* must be the same as their initial markings, (C32) tokens in *W2* must correspond to the users who have placed bids, (C33) the union of the sets of users who have not placed bids and those who have must correspond to the set of initial bidders and (C34) every other place must be empty. The code for the *termination* property would look like this:

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

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

6.2.2. Using Helena. The same result could be obtained using *Helena* for the *termination* property:

```
ltl property termination: [] <> valid_termination;
```

with *valid_termination* representing the conjunction of the explained conditions. For lack of space, we include the full definitions of the used functions, along with definitions of other properties in the Solidity2CPN document¹.

6.2.3. Discussion. Considering that *Helena* showed better results in terms of execution times, we decided to implement our prototype¹ of the transformation algorithm targeting its specification language. While some temporal properties can be expressed using both logics (LTL and CTL), like the aforementioned termination property, we note that some

TABLE 1. STATE SPACE ANALYSIS RESULTS FOR DIFFERENT INITIAL MARKINGS

Possible Bidders		1		2		3		4		5		1		1	
Possible Bids		1		1		1		1		1		2		3	
Possible Secret Keys		1		1		1		1		1		2		3	
Possible Deposits		1		1		1		1		1		2		3	
State Space Generation	H	0s	0s	0s	0s	0s	2s	2s	622s	42s	-	0s	0s	1s	87s
	NH	0s	0s	0s	0s	0s	1s	0s	152s	11s	-	0s	0s	0s	16s
Number of Nodes	H	44		583		9166		156117		2714288		1288		79584	
	NH	24		235		3118		47621		766548		484		19984	
Number of Arcs	H	46		900		19784		446326		9948298		1411		97463	
	NH	26		378		7106		145062		3002038		555		22980	
Number of Dead Markings	H	3		10		35		124		437		65		3695	
	NH	3		10		35		124		437		65		3695	

other properties can only be expressed using one logic and not the other. This, however, does not imply that either logic is a subset of the other. It is also worth mentioning that *Helena*'s verification is on-the-fly whereas *CPN Tools* requires the generation of the whole state space, which can constitute a considerable overhead for the verification.

7. Conclusion

In this paper, we propose a CPN-based formal verification approach for Solidity smart contracts. We implement a prototype of a transformation algorithm that generates a hierarchical CPN model representing a given Solidity smart contract, including both its control-flow and data aspects. Temporal properties are then verified on the CPN model to check corresponding properties on the smart contract, unrestrictedly to certain predefined vulnerabilities. At the time being, the proposed transformation supposes the existence of the level-0 model to be extended. Our next step is to fully automate this process by generating this model from an additional artifact that can provide a description of the intended workflow (e.g., a BP model). Besides, though in this paper we apply our approach on a realistic example (and two other real examples that we include in the repository¹ for lack of space), we intend to evaluate our approach on a larger data-set² of smart contracts once the previously mentioned step is achieved. 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 [16] and applying it on CPNs.

References

- [1] "Solidity docs," <https://docs.soliditylang.org/en/latest/>, 2021.
- [2] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*, 1st ed. Springer Publishing Company, Incorporated, 2009.
- [3] I. Garfatta, K. Klai, M. Graïet, and W. Gaaloul, "Blockchain-based business processes: A solidity-to-cpn formal verification approach," in *Service-Oriented Computing - ICSOC 2020 Workshops, Dubai, UAE, December 14-17, 2020, Proceedings*, 2020, pp. 47–53.
- [4] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen, "CPN tools for editing, simulating, and analysing coloured petri nets," in *Applications and Theory of Petri Nets 2003, 24th ICATPN, The Netherlands, June 23-27, 2003, Proceedings*, 2003, pp. 450–462.
- [5] S. Evangelista, "High level petri nets analysis with helena," in *Applications and Theory of Petri Nets 2005, 26th ICATPN, Miami, USA, June 20-25, 2005, Proceedings*, 2005, pp. 455–464.
- [6] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
- [7] I. Garfatta, K. Klai, W. Gaaloul, and M. Graïet, "A survey on formal verification for solidity smart contracts," in *ACS'W '21: 2021 Australasian Computer Science Week Multiconference, Dmedin, New Zealand, 1-5 February, 2021*. ACM, 2021, pp. 3:1–3:10.
- [8] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Z. Béguelin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*. ACM, 2016, pp. 91–96.
- [9] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CA, USA, January 8-9, 2018*, pp. 66–77.
- [10] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 254–269.
- [11] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, 2017, pp. 442–446.
- [12] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, 2018, pp. 664–676.
- [13] A. Mavridou, A. Laszka, E. Stachtari, and A. Dubey, "Verisolid: Correct-by-design smart contracts for ethereum," in *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, St. Kitts and Nevis, February 18-22, 2019*, 2019, pp. 446–465.
- [14] Z. Liu and J. Liu, "Formal verification of blockchain smart contract based on colored petri net models," in *43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 2*. IEEE, 2019, pp. 555–560.
- [15] W. Duo, X. Huang, and X. Ma, "Formal analysis of smart contract based on colored petri nets," *IEEE Intell. Syst.*, vol. 35, no. 3, pp. 19–30, 2020.
- [16] K. Klai and D. Poitrenaud, "MC-SOG: an LTL model checker based on symbolic observation graphs," in *Applications and Theory of Petri Nets, 29th International Conference, PETRI NETS 2008, Xi'an, China, June 23-27, 2008. Proceedings*, 2008, pp. 288–306.

2. <https://github.com/smartbugs/SolidiFI-benchmark>