

Integrating Business Process Context into Solidity-to-CPN Formal Verification

Ikram Garfatta
Institut Mines-Télécom,
Télécom SudParis,
SAMOVAR UMR 5157
Évry, France

Email: ikram_garfatta@telecom-sudparis.eu

Kaïs Klai
University Sorbonne Paris North,
LIPN UMR CNRS 7030
Villetaneuse, France
Email: kais.klai@lipn.univ-paris13.fr

Walid Gaaloul
Institut Mines-Télécom,
Télécom SudParis,
SAMOVAR UMR 5157
Évry, France
Email: walid.gaaloul@telecom-sudparis.eu

Abstract—Smart contracts, which are self-executing agreements, have a huge range of possible uses from finance to supply chain management. To avoid costly errors and vulnerabilities, it is crucial to guarantee the accuracy and reliability of these contracts. This paper explores the convergence of Blockchain technology, particularly Ethereum’s smart contracts, and Business Process Modeling (BPM), capitalizing on the synergies between these domains. We propose that viewing smart contracts as akin to business processes can significantly enhance the verification of Blockchain-based applications, addressing critical challenges in smart contract correctness and security. In this work we employ a formal verification approach based on Coloured Petri Nets and Linear Temporal Logic to detect potential vulnerabilities in Solidity smart contracts while considering their behavioral context as a business process model.

Index Terms—Solidity; Business Process Modeling; Formal Verification; Coloured Petri Nets; Linear Temporal Logic.

I. INTRODUCTION

Blockchain technology, with its security and transparency features, has revolutionized industries by providing a decentralized platform for secure transactions. Smart contracts, within this ecosystem, automate digital agreements, holding potential applications from finance to supply chain management. However, ensuring their reliability is crucial to avoid costly errors and vulnerabilities.

Numerous attacks on Blockchain platforms have exploited hidden vulnerabilities in deployed smart contracts, showcasing significant risks. The first major attack occurred in August 2010, generating 92 billion BTC due to an integer overflow in the Bitcoin Blockchain [1]. The DAO attack in June 2016 [2], caused by a reentrancy vulnerability, led to the theft of 3.6M ether and a hard fork in the Ethereum Blockchain. The Parity multisig wallet experienced two substantial attacks, resulting in the theft of more than 150K ETH in July 2017 and the locking of 513K ETH in November 2017 [3].

In parallel, the field of Business Process Modeling (BPM) has evolved as a critical discipline in streamlining and optimizing organizational operations. Business processes are fundamental to the core functions of any organization, guiding workflows and decision-making. Remarkably, there is a striking resemblance between the structure of smart contracts and that of BP models. In fact, the intended behaviour of smart

contracts can intuitively be represented using a BP representation that would characterize the *context* in which the smart contracts are supposed to be executed. Such a behavioural context can either come directly as a description of a business model or be derived from a script that would be used to invoke the smart contracts. In our work, we explore the convergence of the Blockchain technology, specifically through its smart contracts, and BPM, leveraging the synergies between these domains. We posit that viewing smart contracts as analogous to BPs can significantly enhance the verification of Blockchain-based applications and we aim to address critical challenges related to smart contract correctness and security.

In our work, we are interested in smart contracts written in Solidity [4], Ethereum’s primary language. We expand on our formal approach for the verification of Solidity smart contracts [5] and focus on its adaptation to detect the potential presence of vulnerabilities with the possibility of taking into account their *behavioural context* as a BP model.

The general approach is based on model checking as a formal verification method applied on Coloured Petri Net [6] as a representation formalism and using Linear Temporal Logic [7] to express the properties to be verified and vulnerabilities to be detected. Thanks to their ability to combine the analysis power of Petri nets with the expressive power of programming languages, Coloured Petri Nets [6] (CPNs) are suitable for the modeling and verification of complex systems, and therefore they are employed in our approach to model the contracts execution with respect to a behavioral context specification defining the workflow within which they are used.

To explain this approach in more detail, we depict its different stages in Figure 1. This approach comprises mainly five steps: (1) transformation of the smart contracts’ Solidity code into CPN sub-models corresponding to their functions [5], (2) transformation of the behavioural context specification into a CPN model, (3) expression of the properties/vulnerabilities to be verified in LTL, (4) generation of a Hierarchical CPN (HCPN) model, and (5) model checking of the generated HCPN model w.r.t to the specified LTL property. This is the final step that puts together all the pieces of the approach.

The paper is organized as follows: Section II reviews related studies on formal verification of Solidity smart contracts.

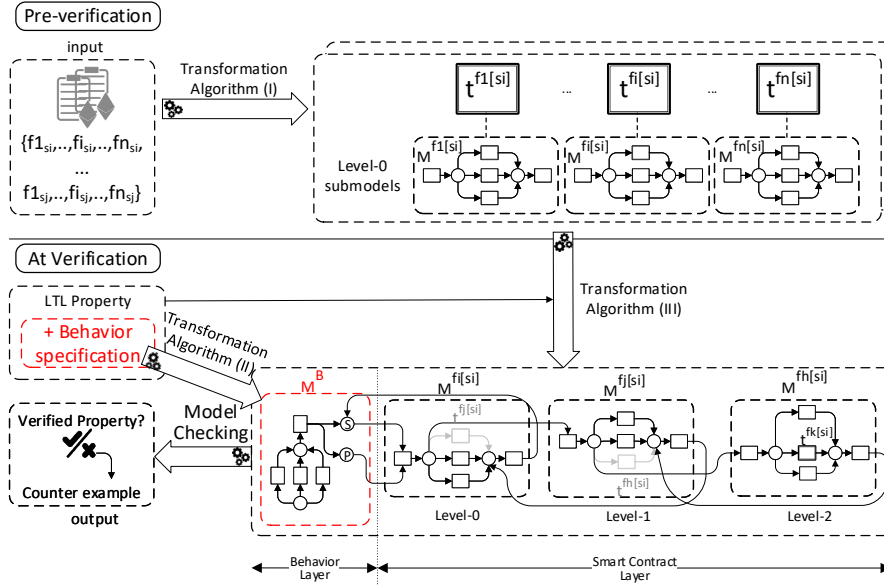


Fig. 1: Overview of the approach

Prerequisites on CPN, LTL, and BP model representation are provided in Section III. Section IV explains the transformation of smart contracts into CPN. Section V formalizes the context of smart contract representations. Vulnerabilities are specified in Section VI, perspectives are outlined in Section VII, and conclusions are drawn in Section VIII.

II. RELATED WORK

Efforts on the verification of smart contracts diverged between informal methods and formal verification approaches. Informal techniques provide quick checks within specific scenarios but lack guarantees of correctness. On the other hand, formal verification techniques offer more rigorous checks but may face scalability challenges.

Existing studies on formal verification of smart contracts follow two main streams [8]. The first group of studies is based on theorem proving [9], [10]. The general idea of such approaches 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. Such a 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. The first attempt was Oyente [11], a tool that operates at the EVM bytecode level, generating symbolic execution traces and analyzing them to detect the presence of four vulnerabilities. Numerous studies followed such as GASPER [12] which reuses Oyente's generated control flow graph for the detection of bytecode patterns with high gas costs, and Osiris [13] that extends it to support the detection of other vulnerabilities.

VeriSolid [14] aims at producing a correct-by-design contract

rather than detecting bugs by transforming a contract modeled as an FSM into a Solidity code and specifying intended behavior using templates for Computation Tree Logic (CTL) that are 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 [15] 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 automatically apply on any smart contract. Another CPN-based proposition was presented in [16] where Hoare's logic is used to generate a CPN model from a contract's bytecode which is then used for the security analysis of the contract. This approach, however, cannot be used for the verification of data-flow related properties as the generated model focuses only on the representation of the workflow of the contract.

To conclude, we note that approaches based on symbolic execution usually use under-approximation which means that critical violations can be overlooked. Moreover, most of the existing studies target specific vulnerabilities in smart contracts, as opposed to verifying customizable properties (more specifically, none of these studies target data-related properties). Besides, most of the proposed approaches operate on the EVM bytecode rather than on the Solidity code which results in loss of contextual information and 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 verify both behavioural and contract-specific properties that can depend on the data-flow and hence is not bound to a restricted set of vulnerabilities. Besides, we note that our approach relies on an explicit model checking technique and that our transformation algorithm operates on the source code, therefore avoiding the consequences of under-approximation and contextual information loss.

III. PRELIMINARIES

A. On Coloured Petri Nets

A Petri net [17] 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). A *Coloured Petri net* [6] is an extension of Petri net that equips the tokens with colours or types, hence allowing them to hold values. The formal definition of a CPN is given in Definition 3.1 and the main concepts needed to define its dynamics are given in Definition 3.2.

Definition 3.1 (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 $Type[v] \in \Sigma, \forall v \in V$.
- 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 with variables in V , is a *guard function* that assigns a guard to each transition t .
- 8) $E : A \rightarrow EXPR_V$ is an *arc expression function* that assigns an arc expression to each arc a such that $Type[E(a)] = C(p)_{MS}$.
- 9) $I : P \rightarrow EXPR_\emptyset$ is an *initialisation function* that assigns an initialisation expression to each place p such that $Type[I(p)] = C(p)_{MS}$.

Definition 3.2 (CPN concepts): For a CPN $(P, T, A, \Sigma, V, C, G, E, I)$, we note the following:

- 1) A *marking* is a function M that maps each place into a multiset of tokens.
- 2) The *initial marking* M_0 is defined by $M_0(p) = I(p)\langle \rangle$ for all $p \in P$.
- 3) $Var(t) \subseteq V$ denote the *variables of a transition* t .
- 4) A *binding* of a transition t is a function b that maps each variable $v \in Var(t)$ into a value $b(v) \in Type[v]$. It is written as $\langle var_1 = val_1, \dots, var_n = val_n \rangle$. The set of all bindings for a transition t is denoted $B(t)$.
- 5) A *binding element* is a pair (t, b) such that $t \in T$ and $b \in B(t)$. The set of all binding elements $BE(t)$ for a transition t is defined by $BE(t) = \{(t, b) | b \in B(t)\}$. The set of all binding elements is denoted BE .

For more details on CPN we refer readers to [6].

B. On Business Process Modeling Representations

When it comes to business process modeling languages, controversy arises as to whether imperative or declarative modeling approaches are better. An empirical investigation [18] states that while imperative languages (e.g., Business Process Model and Notation BPMN [19]) can be considered superior in terms of comprehensibility by end-users, this fact's accuracy can be influenced by the experimental subjects' familiarity with imperative modeling languages. On the other hand,

declarative modeling approaches (e.g. Dynamic Condition Response DCR Graphs [20]) are considered less rigid than their counterpart and therefore more suitable for rapidly evolving business processes. In fact, imperative models represent *how* a process is executed by explicitly defining its control flow while declarative models focus on *why* a process is executed in such a way by implicitly defining its control flow as a set of rules. Consequently, making changes to an imperative model is more time-consuming and complex than altering a declarative one, since the former would entail explicitly adding/deleting execution alternatives, which can call into question the correctness of the model, while the latter could be achieved by adding/deleting constraints from the model to discard/add execution alternatives. In our work, we do not support any claims for the supposed superiority of any parabusinessdigm over the other.

Definition 3.3: A DCR graph is a tuple $G = (E, M, Act, \rightarrow \bullet, \bullet \rightarrow, \rightarrow +, \rightarrow \%, \rightarrow \diamond, l)$ where $\mathcal{M}(G) =_{def} \mathcal{P}(E) \times \mathcal{P}(E) \times \mathcal{P}(E)$ is the set of all markings:

- 1) E is the set of events, ranged over by e .
- 2) $M \in \mathcal{M}(G)$ is the marking of the graph.
- 3) Act is the set of actions.
- 4) $\rightarrow \bullet, \bullet \rightarrow \subseteq E \times E$ are the condition and response relations, respectively.
- 5) $\rightarrow +, \rightarrow \% \subseteq E \times E$ are the dynamic include and exclude relations, respectively, satisfying that $\forall e \in E. e \rightarrow + \cap e \rightarrow \% = \emptyset$.
- 6) $\rightarrow \diamond \subseteq E \times E$ is the milestone relation.
- 7) $l : E \rightarrow Act$ maps every event to an action.

A marking $M = (Ex, Re, In) \in \mathcal{M}(G)$ is a triplet of event sets where Ex represents the previously executed events, Re the pending responses required to be executed or excluded, and In the currently included events.

For more details on DCR Graphs we refer readers to [20].

C. On Linear Temporal Logic

The approach presented in this paper is primarily based on model checking of CPN models w.r.t formulae expressed in Linear Temporal Logic (LTL). This logic was first introduced in [7] as a means to reason about concurrent programs.

An LTL formula is evaluated over an infinite sequence of indexed states ($i = 0, 1, 2, \dots$) where each point in time has a unique successor, starting from an i 'th state. It contains a finite set $Prop$ of atomic propositions, the usual Boolean operators (\neg, \wedge, \vee , and \rightarrow), in addition to temporal operators (Until: \mathcal{U} , Next: \mathcal{X} or \circ , Globally: \mathcal{G} or \square , Future: \mathcal{F} or \diamond).

Definition 3.4 (LTL formula): An LTL formula can be inductively defined as follows:

- $\forall p \in Prop, p$ is an LTL formula.
- If φ and ψ are two LTL formulae, then $\neg \varphi, \varphi \wedge \psi, \varphi \vee \psi, \varphi \rightarrow \psi, \varphi \mathcal{U} \psi, \mathcal{X} \varphi, \mathcal{G} \varphi$ and $\mathcal{F} \varphi$ are LTL formulae.

IV. SOLIDITY-TO-CPN TRANSFORMATION

The main and first step of our proposed approach consists in the transformation of Solidity smart contracts functions into CPN sub-models which are later used as building blocks for

the final HCPN model. This step has been first introduced in [21] and then presented in more detail in [5]. Here, we give a brief idea on this step and its role in our global approach.

A smart contract function is seen as a set of statements. To each one of the statement types a corresponding pattern in CPN is defined, according to which a snippet of a CPN model is generated. The general idea of the transformation process is to browse the body of each function, statement by statement, recursively, and to create snippets of a CPN model according to the type of the processed statement that interconnect to create the function's sub-model. The statement types considered in [5] are as follows: A statement $st \in \mathbb{S}$ can be either a compound statement $\{st[1]; st[2]; \dots; st[N]\}$ (where $\forall i \in [1..N], st[i] \in \mathbb{S}$), or a simple statement (st_{LHS}, st_{RHS}) (where st_{LHS} and st_{RHS} are expressions), or a control statement. A simple statement can be: (i) a function call statement, (ii) an assignment statement, (iii) a variable declaration statement, (iv) a sending statement, or (v) a returning statement. A control statement can be: (i) a requirement statement, (ii) a selection statement (single- or double-branching) or (iii) a looping statement (for or while).

We note that the Solidity supported components can be seen as exhaustive in that they can be used to rewrite smart contracts with more syntactic sugar, even though [5] did not consider all of the Solidity elements (e.g., do while loop).

V. CONTEXT TRANSFORMATION

We recall that the desired behavior of smart contracts can be intuitively depicted using a business process representation that would describe the context in which the smart contracts are intended to be used. Herein we are interested in the generation of the context's CPN sub-model for such a representation. We consider two types of behavioral context specifications:

- 1) *completely-free* if no information is provided on the execution context of a contract (Section V-A)
- 2) *constrained* if the context in which a smart contract is used is provided (e.g., as a DCR Graph) (Section V-B).

A. Completely Free Behavioural Context

If no behavioral context specification accompanies the smart contracts for verification, we define a CPN model to depict their execution freely. In this model (refer to Figure 2), a place S represents the blockchain environment's global state shared across all smart contract functions. Each function f_i is represented by a place P_i for its input parameters, with the initial marking encompassing all possible calling arguments for f_i . Here's the formal definition of CPN4Free, our proposed CPN model for this free context representation.

Definition 5.1 (CPN4Free): Given a set of smart contracts $SSC = \{SC_i, \forall i \in [1, n]\}$, such that n is the number of smart contracts to be verified and $\forall SC_i \in SSC, SC_i = (f_{ji}, v_{hi}), \forall j \in [1, n_i], \forall h \in [1, m_i]$ where n_i is the number of functions in SC_i and m_i is the number of global variables in SC_i , we denote by $Param_{ji} = \{param_{ji}^k, \forall k \in [1, n_{ji}]\}$, the set of input parameters of the function f_{ji} such that n_{ji} is the number of such parameters. A corresponding CPN

model $CPN4Free = (P, T, A, \Sigma, V, C, G, E, I)$ (depicted in figure 2) is defined as follows:

- $P = \{S\} \cup P_{param}$, where $P_{param} = \bigcup_{i \in [1, n]} P_i$, such that $\forall i \in [1, n], P_i = \{p_{ji}, \forall j \in [1, n_i]\}$
- $T = \bigcup_{i \in [1, n]} T_i$, such that $\forall i \in [1, n], T_i = \{t_{ji}, \forall j \in [1, n_i]\}$
- $A = \{(t_i, S), \forall t_i \in T\} \cup \{(S, t_i), \forall t_i \in T\} \cup \{(p_{ji}, t_{ji}), \forall p_{ji} \in P_i, \forall t_{ji} \in T_i, \forall i \in [1, n]\}$
- $\Sigma = \{C_S\} \cup \{C_{P_{ji}}, \forall i \in [1, n], \forall j \in [1, n_i]\}$, where $C_{P_{ji}} = [type_{param_{ji}^1} : param_{ji}^1, \dots, type_{param_{ji}^{n_{ji}}} : param_{ji}^{n_{ji}}]$ and $C_S = [uint : contractBalance, type_{v_{11}} : v_{11}, \dots, type_{v_{m_{nn}}} : v_{m_{nn}}] \times C_{P_{11}} \times \dots \times C_{P_{m_{nn}}}$
- $V = \{x, x'\} \cup \{vp_{ji}, \forall i \in [1, n], \forall j \in [1, n_i]\}$, with $Type[x] = Type[x'] = C_S$ and $Type[vp_{ji}] = C_{P_{ji}}, \forall i \in [1, n], \forall j \in [1, n_i]$
- $C = \{S \rightarrow C_S\} \cup \{p_{ji} \rightarrow C_{P_{ji}}, \forall p_{ji} \in P_i, \forall i \in [1, n]\}$
- $G = \emptyset$
- $E = \{a \rightarrow x, \forall a \in A \cap (\{S\} \times T)\} \cup \{a \rightarrow x', \forall a \in A \cap (T \times \{S\})\} \cap E_{param}$, where $E_{param} = \bigcup_{i \in [1, n]} \{a \rightarrow p_{ji}, \forall j \in [1, n_i]\}$
- $I = \{p \rightarrow init_p, \forall p \in P\}$, where $init_p$ is a predefined initialisation that depends on the type of the place.

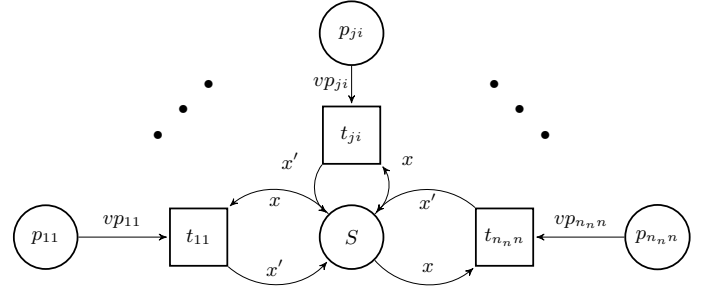


Fig. 2: CPN model for a completely-free behavioural context

B. Constrained Behavioural Context

The behavior of smart contracts can be represented either imperatively or declaratively. We'll focus on integrating declarative representations, namely DCR graphs, into our work, given that existing BPMN-to-CPN transformations [22], [23] for an imperative representation could also be leveraged. Our approach involves transforming DCR graphs into CPN models to verify smart contracts while ensuring semantic equivalence.

1) CPN4DCR - Initial Model: We give the formal definition of the initial CPN4DCR model that we propose and showcase its capabilities and limitations for LTL model checking.

Definition 5.2 (CPN4DCR): Given a DCR graph $G = (E, M, Act, \rightarrow, \bullet, \pm, l)$, a corresponding CPN model $CPN4DCR = (P, T, A, \Sigma, V, C, G, E, I)$ (depicted in figure 3) is defined as follows:

- $P = \{S\}$
- $T = \{t_i, \forall i \in [1, n]\}$, with $n = |E|$
- $A = \{(t_i, S), \forall t_i \in T\} \cup \{(S, t_i), \forall t_i \in T\}$

- $\Sigma = \{C_E, (C_E \times C_E \times C_E)\}$, where C_E is a colour defined as an integer type ($C_E = \text{range } \text{INT}$) where each event $e_i \in E$ is represented in C_E by its index.
- $V = \{Ex, Re, In, Ex', Re', In'\}$, with $\text{Type}[v] = C_E, \forall v \in V$
- $C = \{S \rightarrow (C_E \times C_E \times C_E)\}$
- $G = \{t_i \rightarrow \text{guard}_i, \forall i \in [1, n]\}$, with $n = |E|$
- $E = \{a \rightarrow \langle Ex, Re, In \rangle, \forall a \in A \cap (P \times T)\} \cup \{a \rightarrow \langle Ex', Re', In' \rangle, \forall a \in A \cap (T \times P)\}$ with
 - $Ex' = Ex \cup e_i$,
 - $Re' = (Re \setminus e_i) \cup e_i \bullet \rightarrow$ and
 - $In' = (In \cup e_i \rightarrow+) \setminus e_i \rightarrow\%$
- $I = \{S \rightarrow \langle S_1, S_2, S_3 \rangle\}$ with $\langle S_1, S_2, S_3 \rangle$ the initial marking M of G

For all $t_i \in T$ 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 :
 - 1) $i \in In$,
 - 2) $(\rightarrow \bullet i \cap In) \in Ex$ and
 - 3) $(\rightarrow \diamond i \cap In) \in E \setminus Re$
- the expression $\langle Ex', Re', In' \rangle$ on its output arc is defined s.t.:
 - 1) $Ex' = Ex \cup i$,
 - 2) $Re' = (Re \setminus i) \cup i \bullet \rightarrow$ and
 - 3) $In' = (In \cup i \rightarrow+) \setminus i \rightarrow\%$

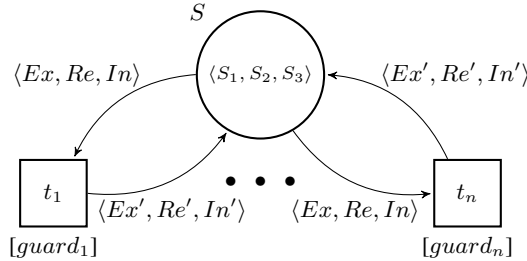


Fig. 3: Initial CPN4DCR

VI. FORMALIZING VULNERABILITIES

In line with Section I, our approach employs model checking to verify LTL properties that articulate the vulnerability susceptibility of contracts. We start by expressing each targeted vulnerability in LTL. For further insight into the vulnerabilities' definitions, we direct readers to [24].

A. Expressing Vulnerabilities in LTL

Here, $M_{s_i}^f$ represents the CPN submodel for function f in smart contract s_i . It's important to note that at times, we employ parameterized propositions, indicating their application to an unspecified aggregated transition. These propositions require explicit definition for each transition to be verified¹. For lack of space, we only include 2 vulnerabilities out of the 6 that we support in our work.

¹Not to be confused with first order predicates

1) *Integer Overflow/Underflow*: In our CPN model, we define correspondences between the types used in Solidity and those offered by *helena* so that they cover the same ranges. We propose to model integer overflows/underflows as a safety LTL property that can be verified on a specific variable x :

$$IUO_x = \Box \neg xIsOutOfRange$$

Where $xIsOutOfRange$ is a proposition that evaluates to true if the value of x is not included in the range of its type which we delimit by defining lower and higher thresholds (minThreshold and maxThreshold respectively).

$$xIsOutOfRange = (x < \text{minThreshold}) \vee (x > \text{maxThreshold})$$

2) *Reentrancy*: This vulnerability is related to functions that contain instructions responsible for Ether transfer, and therefore is applied w.r.t a function containing a *sending* statement. Given such a function, we propose two LTL properties. The first is a safety property defined as follows:

$$\text{Reentrancy}_{M_{s_i}^f} = \Box \neg \text{reentrant}_{M_{s_i}^f}$$

Where $\text{reentrant}_{M_{s_i}^f}$ is true if the necessary condition under which a reentrancy vulnerability can be detected in the function f in the smart contract s_i is valid. 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 :

$$\text{reentrant}_{M_{s_i}^f} = (\neg X\text{Assignment}) \mathcal{U} \text{Sending}$$

Where $X\text{Assignment}$ is true when a statement is an assignment to the variable x and Sending is true when a statement is a sending one. A vulnerability is detected when $\text{Reentrancy}(t_{s_i}^f)$ evaluates to false. This property is used when we only have the code of the smart contract to be verified (i.e., a totally free behaviour). If the code of the interacting contract s_j is available, we propose the following LTL property:

$$\text{Reentrancy}_{M_{s_i}^f} = \text{SendingTo}_{s_j} \rightarrow \Box((\neg \text{SendingTo}_{s_j}) \mathcal{U} \text{endOfFallback}_{s_j})$$

Using this we can verify that once the sending statement is executed (SendingTo_{s_j} is true), it cannot be executed again until the fallback function of the receiving contract has finished ($\text{endOfFallback}_{s_j}$ is true) i.e., no reentrancy can happen.

VII. PERSPECTIVES

Our work has introduced a novel approach for verifying Solidity smart contracts within a business-process-based behavioral context. However, there are exciting avenues for future research. Currently, we are focused on developing a dedicated software tool automating the transformation of Solidity contracts into CPNs and conducting LTL property verification. This tool aims to simplify the use of formal methods, making them accessible to a broader audience of developers and auditors. Figure 4 illustrates the workflow of our ongoing tool

[illegible]

Our next objective is to expand our work to accommodate other behavioral context representations, particularly those emerging from the integration of Blockchain technology with the Internet of Things (IoT). We aim to enhance our current approach for verifying Blockchain-based IoT applications by supporting behavioral context specifications delivered as Node-Red applications [25]. This involves converting Node-Red apps utilizing smart contracts to manage and manipulate data from IoT devices into CPN models. Subsequently, we will apply the same verification approach as in our current work to ensure the correctness of these applications.

Existing verification methods typically target specific vulnerabilities identified as the root cause of attacks or malfunctions. However, solely checking for vulnerabilities in a contract does not ensure its correctness, as flaws unique to that contract could lead to faulty behavior. Our approach addresses this issue by offering a formal verification method that not only checks for vulnerabilities but also allows for the expression of additional context-related properties to verify smart contracts. In this paper, we extend the approach in [5] to incorporate the execution context of the smart contracts as a behavioral specification, considering scenarios where such specification is not provided. We also formalize a set of vulnerabilities in LTL to support their detection using our approach.

- [1] “Overflow incident.” [Online]. Available: https://en.bitcoin.it/wiki/Value_overflow_incident
- [2] D. Siegel, F. Yue, B. Keoun, M. Shen, A. Engler, and B. Dale. “The dao attack: Understanding what happened,” Dec 2020. [Online]. Available: <https://www.coindesk.com/understanding-dao-hack-journalists>
- [3] S. Team, “Parity multi-sig wallets funds frozen (explained),” 2021. [Online]. Available: <https://www.springworks.in/blog/parity-multi-sig-wallets-funds-frozen-explained/>
- [4] “Solidity documentation,” <https://docs.soliditylang.org/en/latest/>.
- [5] I. Garfatta, K. Klai, M. Graët, and W. Gaaloul, “A solidity-to-cpn approach towards formal verification of smart contracts,” in *30th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2021, Bayonne, France, October 27-29, 2021*. IEEE, 2021, pp. 69–74.

- ²<https://soliditycpn.lipn.univ-paris13.fr/>