Reasoning about Smart Contracts via LTL Encoding

Valeria Fionda[1], Gianluigi Greco[1] and Marco Antonio Mastratisi[1]

[1]DeMaCS, University of Calabria, Italy

Abstract

Smart contracts are programs that automatically execute on blockchains when some conditions are verified. They encode the legally relevant events needed to satisfy the terms of an agreement and are usually defined by using procedural languages, even if some recent proposals investigated the possibility of using logic formalisms. However, existing logic-based initiatives are rather limited since they only enable the formal checking of properties of a single smart contract, totally neglecting its possible interactions with other smart contracts running on the same blockchain. This paper proposes a framework, called SCRea, that works on a set of smart contracts specified as Linear Temporal Logic (LTL) formulas. SCRea enables to reason about smart contracts considered individually or by considering them as running cooperatively or competitively on the same blockchain.

Keywords

Smart Contracts, Linear Temporal Logic, Reasoning

1. Introduction

Distributed Ledger Technologies (DLT) [1] offers an infrastructure for the synchronized and shared management of data regulated via consensus algorithms. The blockchain is a particular type of DLT in which values of data are stored in a "chain of blocks". Examples of implementations of blockchain are Bitcoin [5] and Ethereum [6] (exchange of cryptocurrency). DLTs allow for the deployment and execution of smart contracts [7]: programs stored within the distributed register, that are automatically executed when certain conditions occur. The effect of the execution of a smart contract may be the alteration of the state of the register or the activation of other contracts. Some examples of smart contract application include banking functions (e.g. Savings), decentralized markets (e.g. EtherMarket), prediction markets (e.g., Augur,), distribution of music royalties (e.g., Ujo) and encoding of virtual property (e.g., a'Ascribe).

Smart contracts in blockchains are typically programmed in a procedural language specific of the host DLT platform, such as Solidity[8] for Ethereum, or general purpose such as Javascript, Java, Golang, and Python. Some initiatives exist in the literature for the formalization of smart contracts using a logic declarative language (e.g., [2, 3, 4]). However, they only enable the formal checking of the correctness and some other properties of a single smart contract independently from the other smart contracts running on the same blockchain and the state of the blockchain itself.

We try to fill this gap by focusing on the application of linear temporal logic on finite traces (LTL_f) [17], with past operators, as a high-level logical tool for the formalization, validation and resolution of any conflicts in the execution of smart contracts running on the same blockchain infrastructure. In our framework, called SCRea, each smart contract is characterized by a set of preconditions and an effect, where effects are visible on the blockchain when the smart contract is activated or, at most, at

SEBD 2022: The 30th Italian Symposium on Advanced Database Systems, June 19-22, 2022, Tirrenia (PI), Italy
fionda@mat.unical.it (V. Fionda); greco@mat.unical.it (G. Greco); mastratisi@mat.unical.it (M. A. Mastratisi)
0000-0002-7018-1324 (V. Fionda); 0000-0002-5799-6828 (G. Greco)

the subsequent time instant. SCREA will enable, given a set of smart contracts, to verify whether the execution of each of them is compatible with the execution of the others. If so, the framework also allows to establish a particular order of execution that ensures that the preconditions of each smart contract are verified before its execution.

We also studied the applicability of SCREA for the identification of conflict situations at runtime by dealing with one execution trace. In the domain of smart contracts, the term trace denotes a sequence of events executed by a blockchain or a sequence of function or event invocations issued by one or more smart contracts. The availability of information at runtime helps dynamic verification techniques to mitigate one of the main obstacles in the analysis of smart contracts: the need to model a complex blockchain execution environment.

The rest of the paper is organized as follows. In Section 2 we introduce some preliminary definitions. The encoding of smart contracts in linear temporal logic is discussed in Section 3 while Section 4 reports about reasoning problems that are enabled by our encoding. The SCREA framework is described in Section 5 and Section 6 will conclude the paper.

2. Linear Temporal Logic on Finite Traces with Past Operators

Linear temporal logic, denoted by LTL, is a modal logic introduced in the 1970s [9, 10]. Since then, it has found applications in several fields of artificial intelligence and computer science, including planning [11], robotics and control theory [12], the management of business processes [13] and temporal querying of data [14]. LTL is a modal logic whose modalities are temporal operators related to events that occur at particular time instants on an ordered timeline. Classically, LTL formulas are interpreted on infinite traces, but there are some applications where is more appropriate to focus on interpretations on finite traces [16]. Indeed, a liner temporal logic endowed with this semantics, denoted by LTL_f, has been recently studied [15, 17, 18].

Syntax. Let's V be a set of propositional variables. An LTL_f formula φ is built on variables in V using boolean connectives $\wedge$, $\vee$, $\neg$ as well as a number of time operators of two categories: future and past temporal operators. For our purposes we will focus on a specific fragment of LTL_f which includes the future operator X (next) and the past operators Y (previous), H (always in the past), P (in the past), S (since). Moreover, we assume that the negation is atomic in φ in the sense that it is applied directly only to propositional variables. More formally, the formula φ is constructed according to the following grammar, where x is any variable in V:

$$\varphi ::= x \mid \neg x \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid X(\varphi) \mid Y(\varphi) \mid H(\varphi) \mid P(\varphi) \mid (\varphi \; S \; \varphi)$$

Semantics. A finite trace defined on the variables in V is a sequence $\pi = \pi_0, \pi_1, ...\pi_{n-1}$ that associates with each $i \in \{0, 1, ...n-1\}$ a state $\pi_i \subseteq V$ consisting of all propositional variables that are assumed to be true in the time instant i. The length of π (its number of time instants) is denoted by $len(\pi)$. Given a finite trace π, we define that an LTL_f formula φ evaluate true in π at the time instant $i \in \{0, 1, ...len(\pi)\text{-}1\}$ inductively as follows:

- $\pi, i| = x$ if and only if $x \in \pi_i$;
- $\pi, i| = \neg x$ if and only if $x \notin \pi_i$;
- $\pi, i| = (\varphi_1 \wedge \varphi_2)$ if and only if $\pi, i| = \varphi_1$ and $\pi, i| = \varphi_2$;
- $\pi, i| = (\varphi_1 \vee \varphi_2)$ if and only if $\pi, i| = \varphi_1$ or $\pi, i| = \varphi_2$;
- $\pi, i| = X(\varphi')$ if and only if $i < len(\pi) - 1$ and $\pi, i + 1| = \varphi'$;
- $\pi, i| = Y(\varphi')$ if and only if $i > 0$ and $\pi, i - 1| = \varphi'$;
- $\pi, i| = H(\varphi')$ if and only if for each j such that $0 \leq j \leq i$ it's true that $\pi, j| = \varphi'$;
- $\pi, i| = P(\varphi')$ if and only if exists j such that $0 \leq j \leq i$ and $\pi, j| = \varphi'$;

Table 1
Interpretation of temporal operators in the context of smart contracts and blockchains.

$x \in V$	the condition/action x occurs
$\neg x \mid x \in V$	the condition/action x does not occurs
$(x_1 \wedge x_2)$	the conditions/actions x_1 and x_2 occurs
$(x_1 \vee x_2)$	the conditions/actions x_1 or x_2 occurs
$X(x')$	in the next step the condition/action x' must apply/be executed
$Y(x')$	in the previous step the condition/action x' must be valid/have been executed
$P(x')$	in the past there must be an instant in which the condition/action x' was valid/was performed
$H(x')$	in the past the condition/action x' has been verified/has always been performed
$(x_1\ S\ x_2)$	in the past there is an instant in which x_2 has been verified/ executed, and from that moment the condition/action x_1 has always been verified/executed

- $\pi, i| = (\varphi_1\ S\ \varphi_2)$ if and only if exists j such that $0 \leq j \leq i$ and $\pi, j| = \varphi_2$ and for any k such that $j \leq k \leq i$ it's true that $\pi, k| = \varphi_1$.

When π, n-1 $\models \varphi$ we say that π is a model of φ and φ is satisfiable.

3. Encoding Smart Contracts in Linear Temporal Logic

Our goal is to use the fragment of linear time logic on finite traces with past operators described in the Section 2 to reason on smart contracts. From this perspective, the time operators are interpreted as reported in Table 1. In our encoding, each smart contract is a pair $\langle p, e\rangle$, where: *(i)* p are the preconditions, i.e. an LTL$_f$ formula using past operators only; *(ii)* e is the effect, i.e. a propositional formula or an LTL$_f$ formula using future operators only. Since p are the preconditions that activate the smart contract and e the effect of its execution, the smart contract will be expressed by the LTL$_f$ formula $\varphi = H(p \rightarrow e)$, i.e., every time the preconditions p occurs then the smart contract must be executed and, thus, its effect e must be visible. In the smart contracts domain, a trace π denotes a sequence of events executed by a blockchain platform or a sequence of function or event invocations issued by one or more smart contracts. If a trace π satisfies a smart contract $\langle p, e\rangle$ it means that it satisfies the LTL$_f$ formula $\varphi = H(p \rightarrow e)$ encoding it.

Example 1. Let's consider a simple smart contract within the sharing economy context to access multimedia contents. The contract consists of the following clause: if a user requests a content for which he has previously paid, that content must be transferred to him. The propositional variables encoding events are: $requireContent$ (he user requests the content), $payContent$ (the user pays for the content) and $transmitContent$ (the content is transmitted to the user). The precondition of the smart contract is $p = requireContent \wedge P(payContent)$, that is when the user requires a content it must be checked that he already payed for it. The effect is the transmission of the content to the user which is considered to be made in the time instant immediately following the request: $e = X(transmitContent)$. The smart contract will then be codified by the following formula: $\varphi = H((requireContent \wedge P(payContent)) \rightarrow X(transmitContent))$.

4. Reasoning about Smart Contracts

Using LTL$_f$ to encode smart contracts allows to reason over (group of) smart contracts by exploiting its reasoning capabilities. In this section we will discuss several reasoning problems on smart contracts.

4.1. Reasoning Problems on a Single Smart Contract

In this section we discuss three reasoning problems that can be defined if considering a single smart contract.

Model checking. A major issue when dealing with a smart contract is model verification or model checking. The goal is to verify if a given trace π, encoding the events registered by a blockchain, is a model of a given smart contract $\langle p, e\rangle$ encoded by the formula LTL$_f$ $\varphi = H(p \rightarrow e)$. This corresponds to check whether the smart contract has been executed (i.e., the effects of its execution are visible on the blockchain) every time its preconditions have been satisfyied. Checking if a trace is a model of a smart contract can be done in polynomial time w.r.t the length of the formula φ [15].

Example 2. Let's consider the smart contract discussed in Example 1 consisting in the LTL$_f$ formula $\varphi = H((requireContent \wedge P(payContent)) \rightarrow X(transmitContent))$ and the trace $\pi_1 = \{payContent\}, \{requireContent\}, \{transmitContent\}$. It's easy to see that π_1 is a model of φ. Instead, the trace $\pi_2 = \{payContent, requireContent\}$ is not a model of φ since the precondition of the smart contract is satisfied but the content is not transmitted to the user.

Satisfiability. Satisfiability is the problem of establishing whether a smart contract $\langle p, e\rangle$ encoded by the LTL$_f$ formula $\varphi = H(p \rightarrow e)$ admits a satisfying trace π in which it has been executed at least once. This corresponds to check whether the LTL$_f$ formula $P(p) \wedge H(p \rightarrow e)$ admits a model.

Example 3. The smart contract discussed in Example 1 is satisfiable, and examples of traces that satisfy it are:
π_1=$\{payContent\}, \{requireContent\}, \{transmitContent\}$
π_2=$\{payContent, requireContent\}, \{transmitContent\}$
π_3=$\{payContent\}, \{requireContent\}, \{transmitContent\}, \{requireContent\}, \{transmitContent\}$.

Satisfiability at runtime. Satisfiability at runtime is the problem of verifying whether a smart contract $\langle p, e\rangle$ (encoded by the LTL$_f$ formula $\varphi = H(p \rightarrow e)$) is satisfiable given a partial trace π_p. In particular, it is the problem of verifying whether there exists a completion $\pi = \pi_p\pi_t$ of π_p (obtained by adding to π_p a subtrace π_t consisting of a certain number of states) such that the trace π obtained is a model of the formula $P(p) \wedge H(p \rightarrow e)$.

Example 4. Consider again the smart contract discussed in Example 1 and the partial trace $\pi_p = \{payContent\}, \{requireContent\}$. This smart contract can be satisfied at runtime starting from π_p, and examples of completions of π_p are:
π_1=$\{payContent\}, \{requireContent\}, \{transmitContent\}$
π_2=$\{payContent\}, \{requireContent\}, \{transmitContent\}, \{requireContent\}, \{transmitContent\}$

4.2. Reasoning Problems on a Set of Smart Contracts

The previous section considered reasoning problems defined on individual smart contracts. It is also interesting in the context of blockchains to consider problems defined on a set of smart contracts, which are supposed to operate and interact in the same blockchain infrastructure.

Model checking. Given a set of smart contracts $\{\langle p_1, e_1\rangle, \langle p_2, e_2\rangle, ..., \langle p_n, e_n\rangle\}$ encoded by the LTL$_f$ formulas $\varphi_1 = H(p_1 \rightarrow e_1)$, $\varphi_2 = H(p_2 \rightarrow e_2)$,... , $\varphi_n = H(p_n \rightarrow e_n)$, the goal is to verify if a given trace π is a model of $\varphi_1, \varphi_2, ..., \varphi_n$. To this end, we can use the same approach used for the model checking of a single smart contract, but defining appropriately the LTL$_f$ formula encoding the

set. In particular, since no smart contracts must be violated by the trace, we can consider the formula $\varphi = \varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_n$.

Example 5. Consider the smart contract consisting of the following clause: if a user shares a content that has been previously transmitted to him, thus violating contractual terms, that user will be deleted from the system. The propositional variables used for the encoding are: $shareContent$ (the user shares the content), $transmitContent$ (the content is transmitted to the user) and $deleteUser$ (the user is deleted from the system). The precondition of the smart contract is $p = shareContent \wedge P(transmitContent)$. The effect is the deletion of the user from the system which is considered to be made in the time instant immediately following the sharing $e = X(deleteUser)$. The smart contract is encoded by the following formula: $\varphi_2 = H((shareContent \wedge P(transmitContent)) \rightarrow X(deleteUser))$. If we consider that φ_2 and the smart contract discussed in Example 1 (consisting of the LTL$_f$ formula $\varphi_1 = H((requireContent \wedge P(payContent)) \rightarrow X(transmitContent))$) are running on the same blockchain; we can perform model checking by considering the LTL$_f$ formula $\varphi = \varphi_1 \wedge \varphi_2$. Let's consider the following trace: $\pi = \{payContent\}, \{requireContent\}, \{transmitContent\}, \{requireContent\}, \{transmitContent, shareContent\}, \{deleteUser\}$. It is easy to verify that π satisfies φ and it's compliant with the two smart contracts formalized by it.

Consistency of a set of smart contracts. If we consider a set of smart contracts running on the same blockchain, another relevant problem is to verify their consistency, i.e. if it is possible that all smart contracts can sooner or later be executed or if they are incompatible with each other, meaning that the execution of one of them makes another smart contract never executable. The goal is to check if there is a trace π in which all the smart contracts in the set have been executed at least once. Let's consider a set of smart contracts $\{\langle p_1, e_1\rangle, ..., \langle p_n, e_n\rangle\}$ encoded by the LTL$_f$ formulas φ_1=$H(p_1 \rightarrow e_1), \ldots, \varphi_n$=$H(p_n \rightarrow e_n)$. Then, the LTL$_f$ formula for consistency checking is $\varphi = \bigwedge_{i\in\{1,\ldots,n\}}(H(p_i \rightarrow e_i) \wedge P(p_i))$, that is, the conjunction of the formulas encoding the individual smart contracts plus the check that the preconditions of each smart contract have been satisfied at least once (using the $P(p_i)$ subformulas).

Example 6. Consider the set of smart contracts discussed in Example 5. This set of smart contracts is consistent, in fact we found a trace satisfying this set. Let: φ_1''=$((requiresContent \wedge P(paysContent)) \rightarrow X(transmitsContent)$ and φ_2''=$((sharesContent \wedge P(transmitContent)) \rightarrow X(deleteUser)$. The LTL$_f$ formula allowing to check the consistency of the set is: $\varphi = H(\varphi_1'') \wedge H(\varphi_2'') \wedge P(requiresContent \wedge P(paysContent)) \wedge P(sharesContent \wedge P(transmitContent))$. A trace that satisfies φ is the following: $\pi = \{payContent\}, \{requestContent\}, \{transmitContent\}, \{requestContent\}, \{transmitContent, shareContent\}\ \{deleteUser\}$.

Example 7. Consider the two smart contracts within the sharing economy environment $\langle p_1$=$(requestDeletion \wedge H(\neg notDeletable)), e_1$=$deleteContent\rangle$ and $\langle p_2$= $(requestPermanence \wedge H(\neg deleteContent)), e_2$=$notDeletable)$. The smart contract $\langle p_1, e_1\rangle$ establishes that if the owner of a content ask for its deletion ($requestDeletion$) and previously such content has not been declared not deletable ($H(\neg notDeletable)$), the content is deleted from the system. Instead, the smart contract $\langle p_2, e_2\rangle$ states that if a content is requested to be made permanent in the system ($requestPermanence$) and has not been previously deleted ($H(\neg deleteContent)$) then it is marked as not deletable. It is easy to see that this set of smart contracts is not consistent since the execution of one smart contract prevents the precondition of the other to be ever satisfied.

Consistency at runtime. Consistency at runtime is the problem of checking if a set of smart contracts $\{\langle p_1, e_1\rangle, ..., \langle p_n, e_n\rangle\}$ can be satisfied starting from a partial trace π_p, that is to verify whether there exists a completion π=$\pi_p\pi_t$ of π_p (obtained by adding to π_p a subtrace π_t) such that all smart contracts in the set are executed at least once. To solve such a problem the encoding of the set of smart contracts in LTL$_f$ is the same as discussed for the standard consistency checking. However, the partial trace π_p

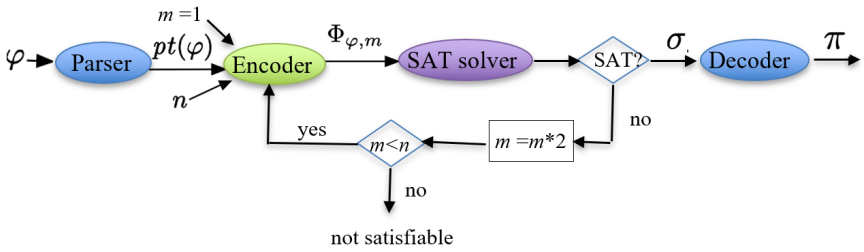

Figure 1: SCREA conceptual architecture

must be completed by adding indicator variables f_i to register the satisfaction of preconditions.

Example 8. Consider again the set of smart contracts discussed in Example 6 and the partial trace π_p=$\{payContent\}, \{requireContent\}$. The completion of π_p corresponds to the addition of the indicator variable f_1 in the last state, i.e., π_p=$\{payContent\}\{requireContent, f_1\}$. The set of smart contracts is consistent at runtime starting from π_p, and examples of completions of π_p are:
π_1=$\{payContent\}, \{requireContent, f_1\}, \{transmitContent, shareContent, f_2\}, \{deleteUser\}$
π_2=$\{payContent\}, \{requireContent, f_1\}, \{transmitContent\}, \{requireContent, , shareContent, f_1, f_2\}, \{transmitContent, deleteUser\}$

5. SCREA: Smart Contracts Reasoner

In this section we introduce the Smart Contracts Reasoner (SCREA) that works on smart contracts encoded in LTL$_f$ and exploits boolean satisfiability for solving the reasoning tasks introduced in the previous section. The proposed reasoner is based on a SAT (boolean satisfiability) rewrite technique that recalls the approach followed by (incremental) model checking methods for bounded LTL [19]. The conceptual architecture of SCREA is shown in Figure 1. The input to the system is the LTL$_f$ formula encoding a single smart contract or a group of smart contracts (see, Section 4).

The first component is the Parser which constructs the parse tree $pt(\varphi)$ associated with the input formula φ. Starting from the parse tree, the SAT rewriting technique rewrites φ into an equivalent propositional formula $\Phi_{\varphi,m}$ which is satisfiable if, and only if, φ can be satisfied by a model of maximum length m. To verify if $\Phi_{\varphi,m}$ is satisfiable, the reasoner can use any existing SAT solver (e.g., Glucose [20]). In particular, $pt(\varphi)$ is given as input to the Encoder module which produces a SAT translation of the formula by initially considering a model length m=1. Then, whenever no model of Φ_φ is found, m is doubled and the process is repeated until m exceeds the upper limit n given as input to the Encoder. If the SAT solver finds a model σ for a SAT formula $\Phi_{\varphi,m}$, the Decoder module translates σ into a trace π (having length at most m) which satisfies φ. Note that n is set by the user and could be less than the minimum length of any model of φ, so the reasoner acts as a correct, although not complete, solver.

5.1. Encoder and Decoder Modules

Given an LTL$_f$ formula representing one or a set of smart contracts, for its SAT encoding a crucial role is played by the parse tree of a formula $pt(\varphi) = (V, E, \lambda)$ that is a tree (V, E) with a node labeling function $\lambda : V \to \{\wedge, \vee, X, Y, P, H, S\} \cup \{x, \neg x | x \in V_\varphi\}$ inductively defined as follows:

- For any literal $l \in \{x, \neg x\}$ with $x \in V_\varphi$, $pt(l) = (\{r\}, \emptyset, \lambda)$ and $\lambda(r) = l$.
- If $pt(\varphi_i)$=(V_i, E_i, λ_i), with $i \in \{1, 2\}$, it is an analysis tree having as root the vertex $r_i \in V_i$ and if $O \in \wedge, \vee$ is a boolean connective or $O = S$ then $pt(\varphi_1 O \varphi_2) = (\{r\} \cup V_1 \cup V_2, \{(r, r_1), (r, r_2)\} \cup$

$E_1 \cup E_2, \lambda)$ is the analysis tree which has as roots the new node r, with r_1 and r_2 as its two children, and where λ is such that $\lambda(r) = O$ and its restriction on V_i coincides with λ_i.

- If $pt(\varphi')$=(V', E', λ') is an analysis tree having the root r' and if $O \in \{X, Y, P, H\}$ is a time operator, then $pt(O(\varphi'))$=$(\{r\} \cup V', \{(r, r')\} \cup E', \lambda)$ is the tree rooted in the new node r whose only child is r' and where λ is such that $\lambda(r)$=O and its restriction on V' coincides with λ.

To explain the encoding approach used by the reasoner to construct $\Phi_{\varphi,m}$, let us first note that, given a trace π, each node $v \in V$ of the parse tree $pt(\varphi) = (V, E, \lambda)$ can easily be equipped with a set $sat(v, \pi)$ of all time instants in which the sub-formula φ_v associated with vertex v holds in π. In particular, such sets can be computed by structural induction on the parse tree according to the semantics of the different temporal operators. Then, the basic idea we use to construct the formula $\Phi_{\varphi,m}$ is to imitate the construction of the sets $sat(v, \pi)$. Formally, we first define the variables $l[0], ..., l[m-1]$ that will be used to encode the last time instant of the model. Intuitively, $\Phi_{\varphi,m}$ will be defined in such a way that there exists an index $i \in \{0, ..., m-1\}$, such that $l[j]$ evaluates true (respectively, false) whenever $j \geq i$ (respectively, $j < i$). The instant i indicates the last time instant of a model of φ – more formally, whenever $\Phi_{\varphi,m}$ is satisfiable, there exists a model of φ of length $i \leq m$. Then, we associate the variables $sv[0], ..., sv[m-1]$ to each node v of $pt(\varphi)$. Intuitively, $sv[i]$ is meant to check if the formula encoded in the parse tree rooted at node v is satisfied at instant i. In the following expressions, we use two additional variables $sv[-1]$ and $sv[m]$, whose value will actually be a constant that will be fixed depending on the type of node v.

Then, the SAT encoding is the formula $\Phi_{\varphi,m} = \bigwedge_{v \in V}(\Phi_v \wedge sroot[m\text{-}1] \wedge l[m\text{-}1]) \wedge \bigwedge_{i=0}^{m-2}(l[i] \rightarrow l[i+1]) \wedge \bigwedge_{i=0}^{m-2}(l[i] \rightarrow \bigwedge_v(sv[i] \leftrightarrow sv[i+1]))$ and, for all $v \in V$, $\Phi_v = sv[0] \leftrightarrow \Phi_v^0 \bigwedge_{i=1}^{m-1}(\neg l[i\text{-}1] \rightarrow (sv[i] \leftrightarrow \Phi_v^i))$, where:

- if $\lambda(v) \in \{x, \neg x\}$, then $\Phi_v^i = \bigwedge_{v' \in V, \lambda(v') \equiv \neg\lambda(v)} \neg sv'[i] \wedge \bigwedge_{v' \in V, \lambda(v') \equiv \lambda(v)} sv'[i]$;
- if $\lambda(v)$=$\wedge$ or $\lambda(v)$=$\vee$, then Φ_v^i=$sv_1[i]\lambda(v)sv_2[i]$, where v_1 and v_2 are the right/left children of v;
- if $\lambda(v)$=X, then Φ_v^i=$sv'[i+1] \wedge \neg l[i]$, where v' is the child of v and $sv[m]$ is the constant false;
- if $\lambda(v)$=Y, then Φ_v^i=$sv'[i-1]$, where v' is the child of v and $sv[-1]$ is the constant false;
- if $\lambda(v)$=P, then Φ_v^i=$sv'[i] \vee sv[i-1]$, where v' is the child of v and $sv[-1]$ is the constant false;
- if $\lambda(v)$=H, then Φ_v^i=$sv'[i] \wedge sv[i-1]$, where v' is the child of v and $sv[-1]$ is the constant true;
- if $\lambda(v)$=S, then Φ_v^i=$(sv_2[i] \wedge sv_1[i]) \vee (sv_1[i] \wedge sv[i-1])$, where v_1 and v_2 are the left and right children of v and $sv[-1]$ is the constant false.

We note, for the sake of completeness, that $\Phi_{\varphi,m}$ is rewritten (in polynomial time) in conjunctive normal form, before being passed to the solver. Moreover, by construction, it is possible to establish that $\Phi_{\varphi,m}$ is satisfiable if, and only if, φ can be satisfied by a model π with $len(\pi) \leq m$. When SCREA finds a model σ for a SAT formula $\Phi_{\varphi,m}$, the decoder module is responsible for the building of the trace π that satisfies the smart contract φ. In particular, starting from σ it is possible to construct the trace π satisfying φ by implementing the following steps: *(i)* If $l[i] \in \sigma$ and there is no $l[j] \in \sigma$ with $j < i$ then the decoder constructs a trace π with i states; *(ii)* For every v such that $\lambda(v) = x$ and every i such that $sv[i] \in \sigma$ then the decoder adds x to $\pi[i]$.

5.2. SCREA at runtime

In this section we will discuss how the reasoner can be applied to solve reasoning problems at runtime. To work at runtime it is necessary to slightly modify the architecture of SCREA. In particular, the Encoder module must also receive in input the partial trace π_p which contains the events recorded on the blockchain up to the current time instant. In this case, the parameter m concerns the additional number of states (i.e., sets of events) that can happen after those already present in π_p. Moreover, if SCREA is meant to work on a set of smart contracts π_p must be preprocessed to add indicator variables.

6. Concluding remarks

In this paper we proposed a framework, based on the encoding of smart contracts in Linear Temporal Logic on finite traces, that enables various reasoning tasks on a single and on a set of smart contracts running on the same blockchain. The proposed framework, called SCRea, is the first tool developed with the aim of checking the consistency of a set of smart contracts that are supposed to cooperate in same blockchain infrastructure and it is able to work at runtime on a running blockchain system. As a first line for future research, we are planning to implement our framework on an existing blockchain infrastructure to test its effectiveness on a real system. As a limitation of the proposed framework we want to point out that LTL is based on a propositional logic and, thus, has some limitations in expressiveness. Therefor, a second line for future research regards the extension of the proposed framework with a more powerful logic such as first-order temporal logic that will increase SCRea's expressive power.

References

[1] Douglis, F., Stavrou, A. (2020). Distributed Ledger Technologies. IEEE Internet Comput., 24(3).

[2] Idelberger, F., Governatori, G., Riveret, R., Sartor, G. (2016). Evaluation of Logic-Based Smart Contracts for Blockchain Systems. In Proc. of RuleML, pp. 167–183.

[3] Hu, J., Zhong, Y. (2018). A Method of Logic-Based Smart Contracts for Blockchain System. In Proc. of ICDPA, pp. 58–61.

[4] Stancu, A., Dragan, M. (2020). Logic-Based Smart Contracts. In Proc. of WorldCIST, pp. 387–394.

[5] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

[6] Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper, 151(2014), 1-32.

[7] Szabo, N. (1997). The idea of smart contracts. Nick Szabo's Papers and Concise Tutorials, 6.

[8] Dannen, C. (2017). Introducing Ethereum and Solidity (p. 185). Berkeley: Apress.

[9] Pnueli, A. (1977). The Temporal Logic of Programs. In Proc. of FOCS, pp. 46–57.

[10] Pnueli, A. (1981). The Temporal Semantics of Concurrent Programs. Theoretical Computer Science, 13, 45–60.

[11] Calvanese, D., De Giacomo, G., Vardi, M. Y. (2002). Reasoning about actions and planning in ltl action theories. In Proc. of KR, pp. 593–602.

[12] Ding, X., Smith, S., Belta, C., Rus, D. (2014). Optimal Control of Markov Decision Processes With Linear Temporal Logic Constraints. IEEE Transactions on Automatic Control, 59(5), 1244–1257.

[13] Fionda, V., Guzzo, A. (2020). Control-Flow Modeling with Declare: Behavioral Properties, Computational Complexity, and Tools. IEEE Trans. Knowl. Data Eng., 32(5), pp. 898–911.

[14] Chekol, M.W., Fionda, V., Pirrò, G. (2017). Time Travel Queries in RDF Archives. In Proc. of MEPDaW, pp. 28–42.

[15] Fionda, V., Greco, G. (2018). LTL on Finite and Process Traces: Complexity Results and a Practical Reasoner. J. Artif. Intell. Res., 63, pp. 557–623.

[16] Pesic, M., Bosnacki, D., Van der Aalst, W. (2010). Enacting Declarative Languages Using LTL: Avoiding Errors and Improving Performance. In Proc. of SPIN, pp. 146–161.

[17] De Giacomo, G., Vardi, M. Y. (2013). Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In Proc. of IJCAI, pp. 854–860.

[18] Li, J., Zhang, L., Pu, G., Vardi, M. Y., He, J. (2014). LTLf satisfiability checking. In Proc. of ECAI, pp. 513–518.

[19] Biere, A., Heljanko, K., Junttila, T., Latvala, T., and Schuppan, V. (2006). Linear Encodings of Bounded LTL Model Checking. Logical Methods in Computer Science, 2(5), 1–64.

[20] Audemard, G., and Simon, L. (2009). Predicting Learnt Clauses Quality in Modern SAT Solvers. In Proc. of IJCAI, pp. 399–404.