Towards FATEful Smart Contracts

Luigi Bellomarini[1], Marco Favorito[1], Eleonora Laurenza[1], Markus Nissl[2] and Emanuel Sallinger[2,3]

[1]*Bank of Italy, Italy*
[2]*TU Wien, Austria*
[3]*University of Oxford, UK*

Abstract

Achieving fair, accountable, transparent, and ethical decentralized finance requires activating enabling properties at the level of smart contracts, the executable scripts at its basis. In this vision paper, a joint effort of the Central Bank of Italy, TU Wien, and the University of Oxford, we leverage the vast amount of experience in this sense from the database community and propose a logic-based reasoning framework that captures smart contracts as a set of rules in DatalogMTL, a temporal language for querying databases. We show how the theoretical underpinnings of the reasoning of DatalogMTL convey important properties to the approach and show it in action on industrial cases of relevance to a central bank

Keywords

Smart contracts, DatalogMTL, FATE principles

1. Introduction

The Artificial Intelligence and database communities are experiencing a growing infusion of the *FATE* (*Fairness*, *Accountability*, *Transparency*, *Ethics*) [1]. These principles are gaining prominence, drawing attention to the non-functional requirements of everyday AI-assisted and data-driven decision-making and catalyzing the discussion around regulatory bodies. Unfortunately, the same level of attention to these high-level concerns is not mirrored in developer circles, and recent studies underscore the scant regard machine learning developers have shown for FATE concerns in machine learning applications [2, 3].

FATE and DeFi. We see similar patterns emerging when assessing developers' awareness of FATE concerns within the industrial realm of *Decentralized Finance* (DeFi). DeFi entails financial transactions devoid of intermediaries, instead relying on software modules executed on a decentralized public ledger [4]. At the core of DeFI is the notion of *smart contracts* [5], which are machine-readable and executable agreements that establish and enforce the binding terms for the parties involved.

Supporting FATE. In the AI and data world, social forces have been effective in supporting FATE, for example, by means of third-party audits of the algorithms, either conducted by experts

6th Distributed Ledger Technology Workshop, May 14–15, 2024, Turin, Italy
luigi.bellomarini@bancaditalia.it (L. Bellomarini); marco.favorito@bancaditalia.it (M. Favorito); eleonora.laurenza@bancaditalia.it (E. Laurenza); nissl@dbai.tuwien.ac.at (M. Nissl); sallinger@dbai.tuwien.ac.at (E. Sallinger)
0000-0001-6863-0162 (L. Bellomarini); 0000-0001-9566-3576 (M. Favorito); 0000-0002-2786-8163 (E. Laurenza); 0000-0001-8196-5688 (M. Nissl); 0000-0001-7441-129X (E. Sallinger)

or by everyday users. As recently highlighted by Hong in CACM [1], prominent examples can be found in the fights against the racial bias of face-recognition systems [6], and commercial gender disparities in photo cropping or credit card algorithms [7]. These audits, spurred by social forces and the scientific community, have provided regulators with positive guidance.

In contrast to AI, DeFi boasts a substantial theoretical transparency advantage, thanks to its open-access code and the ability for anyone to inspect smart contract data on a public ledger. However, enforcing policies in a decentralized context remains an incredibly challenging task. The praiseworthy goal of establishing standards, taxonomies, compliance measures, quality controls, and upholding ethical principles [8] can benefit from robust support from the social forces in the monitoring and enforcement of such policies. However, this must align with the specific technical attributes of these smart contracts. Yet, smart contracts have been criticised within the community due to their overly complex business logic, and limited explainability, resulting in a lack of transparency. Furthermore, they often prove challenging to describe and communicate, rendering them less user-friendly, particularly for non-technical users [9, 10, 11].

A *Knowledge Representation and Reasoning* (KRR) approach to smart contracts. In the area of deductive AI and ontological reasoning on databases, logic-based approaches built on top of KRR formalisms are gaining increasing attention in industrial settings, with many successful financial applications [12, 13, 14]. Modern logical languages manage to strike a good balance between expressive power and computational complexity, resulting in compact and efficiently executable formalizations of complex domains, for instance, being able to capture SPARQL under OWL 2 entailment regimes [15] and so enabling ontological reasoning. The *declarative* paradigm sustains simplicity, transparency, compactness, and understandability of code, which becomes algorithm-independent and closer to the high-level specifications, policies, and standards. The *well-defined semantics* of KRR languages fosters non-ambiguity, ease of use for non-technical users, and correctness. The intrinsic *step-by-step nature of logical reasoning* is conceptually close to notions of *explainability* and thus supports decision transparency.

The thesis of this vision paper is that a KRR framework for smart contracts that addresses FATE by design is both theoretically and practically viable. For the theory, we show that, by building on the underpinnings of logic-based reasoning, the features important to achieving FATE desiderata can be obtained and rigorously justified. In terms of application, we show that our framework is adaptable to serve as both an interpreted and a compiled execution mode for real-world contracts.

Contributions to industrial advances. In recent industrial EDBT work done by the Central Bank of Italy [16], they started to investigate the possibility of encoding complex smart contracts in DatalogMTL [17, 18], a temporal extension of the Datalog language [19] of databases. They obtained promising results, which highlighted the potential of a KRR approach in the specific case of a derivative contract. In this work, a joint effort of the Central Bank of Italy, TU Wien, and the University of Oxford, we aim to go substantially further by: (i) proposing a **full-fledged and general framework for smart contracts** that sustains FATE concerns; (ii) leveraging the vast amount of **experience from the database community to achieve enabling properties**; (iii) using our framework to study and implement **proof of concepts for many smart contracts** where FATE is a core desideratum of a central bank, namely, *tokenisation*, *peer-to-peer transactions*, *decentralized financial markets*, and *smart legal contracts*.

2. Overview of the framework and related work

We use a form of *declarative logical object-oriented approach* and encode the behaviour of a *class of smart contracts* as a set Σ of reasoning rules—or programs—working on a database D of temporal facts. A temporal fact of D is such that it holds in a given time interval, for example, $price(123, 2)@[2023\text{-}09\text{-}01,2023\text{-}09\text{-}02]$ defines the price 2 for the asset 123 in a two-day interval.

To model the rules of Σ, we introduce *DatalogMTL*^{S}, a variant of DatalogMTL with features of practical utility. A smart contract is then an *instance of a smart contract class*, whose time-dependent status is represented as a database D of temporal facts. Instances are *stateful* objects and the contract execution consists in invocations, akin to method calls, that are carried out by the involved parties. Calls result in updates to the status D through the addition of new facts. The semantics of a call is operationally described as the application of the rules of Σ (denoted as $\Sigma(D)$) to the temporal facts of D, extended with call-specific facts.

DatalogMTL^S rules are sets of $head \leftarrow body$ logic implications where the body is a conjunction of atoms and the head is an atom. As a general guideline, whenever the body of a rule is satisfied by a conjunction of facts in D at a point in time t, the evaluation of the rule triggers the insertion in D of a new fact for the head atom, holding at t. For example, the rule '$position(x, w) \leftarrow buy(x, a, q), price(a, p), w = p * q$' states that, for every point in time t, the position w of a trader x that buys an amount q of an asset a of price p is obtained as $p * q$. So if D contains the price fact $price(123, 2)$ and $buy(0x241F, 123, 12)$, both holding at $[2023\text{-}09\text{-}01,2023\text{-}09\text{-}02]$, a new $position(0x241F, 24)$ holding in the same time interval will be added to D.

In DatalogMTL^S, temporal operators can be used to either modify the temporal binding of body atoms to facts of D, or to alter the temporal validity of the generated facts. For instance, an expression of the form $\diamondsuit_{[0,1d]} position(x, w)$ in the body holds at a point in time t, if the trader x had at least an open position w in the interval $[t-1, t]$, while an expression of the form $\boxplus_{[0,1d]} position(x, w)$ in the head, states that the position will be open in the interval $[t, t+1]$ for every t. In the following, we show a smart contract class defining a simple financial market:

$$R1: \quad accepted(\$\text{sender}, y) \leftarrow \#open(y), \neg marketClosed.$$
$$R2: \quad \underline{\boxplus} position(x, y, k) \leftarrow accepted(x, y), price(p), k = y * p.$$
$$R3: \quad return(x, g), \underline{\boxplus} position(x, y, 0) \leftarrow \#close(), price(p), position(\$\text{sender}, y, k), g = y * p - k.$$

At a specific point in time in which the market is not closed, a trader tries to open a position by investing an amount y (Rule $R1$). The constant $\$sender$ is a call-level variable that at runtime binds to the invoking trader. When the transaction is accepted (Rule $R2$), the position of the trader x on the amount y is updated by multiplying by the current price p. The $\underline{\boxplus}$ operator stands for a new future temporal validity of the position fact. Finally, when the position is closed (Rule $R3$), the final profit g is computed based on the current price.

Execution modes. From a practical perspective, our framework implements $\Sigma(D)$ by enabling three execution alternatives, each offering specific properties, as reported in Figure 1: (i) *on-reasoner execution*: the rules are applied natively by a reasoning system supporting DatalogMTL or DatalogMTL^S such as *Temporal Vadalog* [20] or *MeTeoR* [21]; (ii) *on-chain execution*: the DatalogMTL^S programs are verifiably translated into the language of a target system, for instance, *Solidity* [22] or Bitcoin *Script* [23] and executed within the target systems; (iii) *off-chain execution*, the rules are applied with an on-reasoner execution with persistent effects on a

blockchain and cryptographically verifiable computation.

The expert intention is substantiated as a natural language (NL) contract or directly encoded in a DatalogMTL^S program Σ, and the use of large language models (LLM) can bridge the gap between the natural language specification of the contract and the encoding of its DatalogMTL^S version [24].

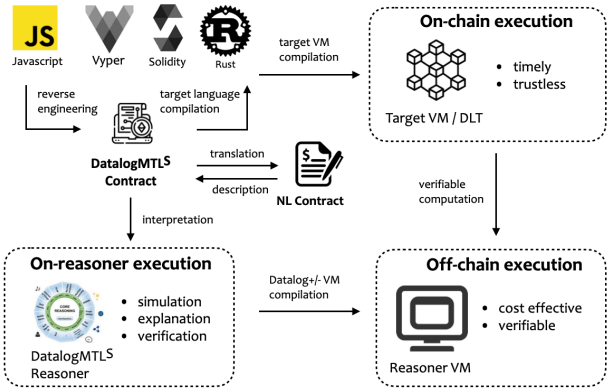

Figure 1: **Overview of the DatalogMTL^S framework.**

In the on-reasoner execution mode, to evaluate Σ, reasoners use variants of the CHASE procedure [25] specialized for the temporal extensions [26]. They offer *full explainability* of the produced facts as a side effect of the inference process of the chase. Also, they are suited to be used for *simulation* and *runtime verification* purposes (see Section 5) as step-by-step debugging can be emulated by incrementally adding facts to D and monitoring the results entailed by the application of Σ. On the other hand, the execution relies on a trusted centralized system.

Conversely, on-chain execution offers a *trustless* paradigm, only requiring that the user acknowledges the translation of the DatalogMTL^S code into the target language, which can be verified through open-sourced translators (see Section 4). This trustlessness is underpinned by the validation properties ensured by the distributed consensus protocol embraced by the blockchain, guaranteeing the integrity of the mined blocks. What is more, on-chain execution is characterized by its *timeliness*, as results are promptly included in the first mined block.

However, on-chain execution comes with a high cost and is ill-suited for complex applications. In contrast, off-chain execution is widely regarded as a practical and efficient alternative, and there is a large body of related work such as state-channels [27], Plasma [28], and Zero-Knowledge Rollups [29]. In particular, specific protocols have been proposed, that help attest the integrity of off-chain computation (i.e., *verifiable computation*), such as ZK-SNARK [30] and ZK-STARK [31]. Towards this direction, the construction of specialized virtual machines compiling succinct ZK proofs for DatalogMTL^S executions is envisaged here, but beyond the scope of this vision paper and a matter of future work.

Quality. Investigating and assessing the properties of a smart contract stands as a paramount concern for regulators, financial institutions, and central banks. Beyond CHASE-based explainability, our framework establishes a twofold path for the scrutiny and validation of smart contracts: during static compile-time and dynamically at runtime. For compile-time checks, as contracts are specified in logic, dedicated tools can be developed as future work, that apply formal methods for *logical verification*, with good decidability expectations in the presence of bounded input sources [32]. In contrast, acquiring dynamic checks in our framework is a relatively straightforward process, involving the real-time querying of the system to examine outcomes dependent on Σ and D. We believe that these dynamic checks hold significant promise

for easy integration by the aforementioned stakeholders into their monitoring and compliance tools. From a technical standpoint, we represent dynamic checks as *temporal conjunctive queries*.

For example, a possible validation for our smart contract in the financial market example above involves inquiring whether, in the past year, a user has repeatedly accumulated losses as a result of closing positions. This may indicate a deficiency in the associated process checks and so a compliance problem. We employ a combination of the $\boxminus$ operator from DatalogMTL^S, which signifies the persistent continuity of its argument in the past, and the $\diamondminus$ operator, indicating the occurrence of a fact within a specified past interval, as we have seen:

$$Q \leftarrow \boxminus_{[0,1y]} \diamondminus_{[0,1m]} (\#close, return(x, g), g < 0).$$

Termination and Complexity. Fact entailment in DatalogMTL is a decidable task, in particular PSPACE in data complexity [17]; therefore we have *guaranteed termination* and *guaranteed computational complexity*. Moreover, the rules modelling real-world smart contracts need to allow for the derivation of facts into present and future time points, while the propagation towards the past is almost never required. Under this condition, the set Σ belongs to the *forward-propagating fragment*, namely, DatalogMTL^{FP} [33], for which a *finite representation* of infinite models is always possible [18]. It is important to point out that in DatalogMTL the use of arithmetic and recursion can, in general, lead to undecidability [34] and a comprehensive study of arithmetic in DatalogMTL has not been provided yet. However, our framework conditions the activations of the rules on the specific smart contract functions being called, which reduces the cases of potentially harmful recursion.

3. Industrial use cases

We now give a set of smart contracts of industrial relevance to see our framework in action.

ERC-20. The ERC-20 [35] is a well-known and widely adopted Token Standard that implements an API for tokens within smart contracts. The following DatalogMTL^S smart contract implements a simple ERC-20 contract with a fixed supply S.

$$\begin{aligned}
R1:&\ \boxplus totalSupply(S), \bar{\boxplus} balanceOf(\$\text{sender}, S) \leftarrow \#init(S).\\
R2:&\ \bar{\boxplus} balanceOf(\$\text{sender}, 0) \leftarrow \neg balanceOf(\$\text{sender}, X), \#create().\\
R3:&\ \bar{\boxplus} balanceOf(\$\text{sender}, B_s - A),\\
&\quad \bar{\boxplus} balanceOf(to, B_r + A) \leftarrow balanceOf(\$\text{sender}, B_s), balanceOf(to, B_r),\\
&\qquad\qquad B_s \geq A, \#transfer(to, A).
\end{aligned}$$

Rule $R1$ initializes the smart contract state by adding the facts $\boxplus totalSupply(S)$ and $\bar{\boxplus} balanceOf(\$\text{sender}, S)$. Note that using the $\bar{\boxplus}$ operator allows to overwrite the balance in case of a transfer. Rule $R2$ allows the sender to initialize a balance, if not already done earlier. Rule $R3$ implements the "transfer" function from the sender address \$sender to the recipient address *to* of amount A. Atoms of the form $balanceOf(address, X)$ in the body are used to query the balance X of *address* (a common pattern in logic programming), the condition $B_s \geq A$ imposes that there is enough balance from the sender account to complete the transfer, and the head of the rule uses the $\bar{\boxplus}$ to update the balances accordingly. We omitted *transferFrom* and *approve*.

Legal Recourse. A legal recourse against a contract refers to the options available to parties when a contract is breached or when there is a dispute regarding the terms or performance of the contract. When a contract is executed via a smart contract, blockchain immutability poses a

significant challenge when altering or amending the contract effects once its terms have been carried out. The use of declarative languages, which enhances transparency, helps mitigate this issue by minimizing the ambiguity inherent in natural language, thereby reducing the likelihood of legal disputes [36]. However, unforeseen events such as force majeure, varying jurisdictional interpretations, and contract bugs may necessitate adjustments to the contract outcome, even after the transaction has been finalized. Moreover, it is important to distinguish between enabling legal recourse in a contract and appointing an arbiter, as an actor in the contract itself, who makes decisions regarding the transaction outcome in cases where the involved parties fail to reach an agreement. Legal recourse is typically considered a post-transaction event, which may also involve the arbiter and occurs after the transaction has been completed.

$$
\begin{aligned}
R1:&\quad \boxplus partA(addr_A), \boxplus partB(addr_B),\\
&\quad \boxplus arbiter(addr_a), \boxplus contractId(id_c) \leftarrow \#init(addr_A, addr_B, addr_a, id_c)\\
R2:&\quad \#Contract(x)[addr_A, addr_B] \leftarrow contractId(x), partA(addr_A), partB(addr_B).\\
R3:&\quad balanceFlowRec(x1{-}x2, y1{-}y2, z1{-}z2) \leftarrow \diamondsuit_{[1,1]} balance(x1, A), balance(x2, A), \diamondsuit_{[1,1]} balance(y1, B), balance(y2, B),\\
&\quad \diamondsuit_{[1,1]} balance(z1, C), balance(z2, C), partA(A), partB(B), contractId(C).\\
R4:&\quad \#Contract(x).undoAndRefund(A) \leftarrow partA(A), arbiter(x, \$\text{sender}), contractId(x), \#legalOutcomeA().\\
R5:&\quad \#Contract(x).undoAndRefund(B) \leftarrow partB(B), arbiter(x, \$\text{sender}), contractId(x), \#legalOutcomeB().\\
R6:&\quad \#Contract(x).undoAndCancel(A, B) \leftarrow partA(A), partB(B), arbiter(\$\text{sender}), contractId(x), \#legalOutcomeCancel().
\end{aligned}
$$

Rule $R1$ initializes the contract; Rule $R2$ initializes the insured contract; Rule $R3$ manages the balances of the parties involved; Rule $R4$ (resp. $R5$) handles the case when the recourse resolves in favour of party A (resp. B); and Rule $R6$ undoes the effects of the insured contract. The trigger atoms $\#Contract(x).undoAndRefund$ allow to interact with the ensured contract.

4. Compilation in Solidity

While a complete and detailed design of a compilation technique of DatalogMTL^S contracts to Solidity smart contracts is out of this paper's scope, in this section, we give the intuition of a comprehensive approach for the translation and exemplify by showing meaningful patterns.

Types and Variables. Predicates occurring in a DatalogMTL^S program belong to either a *status schema* $\mathcal{S}$, a *transient schema* $\mathcal{T}$, or a *function schema* $\mathcal{F}$. Generated atoms whose predicates are in $\mathcal{S}$, since they persist across CHASE computations (i.e., VM function calls) in the status database D, are mapped into contract state variables and data structures; transient atoms are mapped in local function data, and function or trigger atoms are used to generate Solidity functions. If the predicate is unary, the atom is mapped to a variable, while predicates for two or more arguments are mapped in Solidity structs. There might be user-defined exceptions, e.g., *balanceOf* can be compiled into a map. Furthermore, each position of a predicate must be mapped to a Solidity type (e.g., the argument of *supply* and other amounts to type '`uint256`', account addresses, such as the argument in *buyer* and *seller* above, are mapped to the type '`address`'). The compilation process must analyze the program and check whether the provided type mapping is consistent. The occurrence of special call variables (e.g., $sender and $this.balance) are translated into `msg` attributes (`msg.sender` and `this.balance`, respectively).

Functions. The rules with the *#init* trigger atom are converted into constructors, and each variable in the argument list is converted into a constructor argument, typed according to the specified type mapping. Similarly, the other trigger rules (i.e., with a trigger predicate in

```
contract SimpleERC20Contract {
  uint256 immutable totalSupply;
  mapping(address => uint256) public balanceOf;

  constructor(uint256 _S) {
      totalSupply = _S;
      balanceOf[msg.sender] = _S;
  }

  function create() public {
    if (balanceOf[msg.sender] != 0) { revert(); }
    balanceOf[msg.sender] = 0;
  }

  function transfer(address _to, uint256 _A) public {
    uint256 Bs = balanceOf[msg.sender];
    uint256 Br = balanceOf[to];
    if (!(Bs >= _A)) { revert(); }
    balanceOf[msg.sender] = Bs - _A;
    balanceOf[_to] = Br + A;
  }
}
```

Figure 2: The Solidity code generated from the ERC-20 DatalogMTL^S program example.

the body) are converted into functions; if there is more than one rule with the same trigger predicate in the body, we use function overloading to distinguish the different behaviours in the Solidity code. Generated head atoms of the form $\boxplus P(\mathbf{t})$ are translated into the initialization of immutable variables, while $\overline{\boxplus}$ are translated into variable updates. Predicates that occur in temporal operators with an arbitrary time interval, such as $\boxplus_{[t_1,t_2]} P(\mathbf{t})$, $\boxminus_{[t_1,t_2]} P(\mathbf{t})$ and $\diamondsuit_{[t_1,t_2]} P(\mathbf{t})$, correspond to variables that must be read via internal getter functions, which return the right value depending on the timestamp of the current function call. Depending on the time intervals for a specific predicate, a data structure that indexes its values appropriately, e.g., by intervals in which certain values hold, might be needed. For example, if the only temporal operator that appears in front of a predicate is $\diamondsuit_{[0,1]}$, we only need to remember the value of the associated variable in the previous time step.

Control flow. Advanced queries (e.g., nested temporal operators, joins, etc.) on the current contract state database might require advanced program synthesis techniques to generate gas-efficient on-chain computation. Other expressions in the body of a rule (such as comparisons, equality check conditions, negated atoms, atoms with free variables) correspond to control flow constructs in Solidity (i.e., if-then-else clauses, error handling). The parsing of the DatalogMTL^S must consider the program's derivation paths starting from each trigger rule.

Figure 2 shows the output of the compilation process over the example DatalogMTL^S program for the ERC-20 token. The predicates are partitioned as follows: $\mathcal{S} = \{totalSupply, balanceOf\}$, $\mathcal{T} = \{\}$ and $\mathcal{F} = \{\#init, \#create, \#transfer\}$. The argument of the *totalSupply* predicate is mapped to the `uint256` type, and *balanceOf* is mapped to a mapping from `address` to `uint256`, where the first argument of the predicate is the key of the mapping. The constructor function is generated from *R*1, while `create` and `transfer` are generated from *R*2 and *R*3, respectively. Note that the definition of the `create` function is necessary for the DatalogMTL^S semantics but pleonastic in Solidity since the values of mapping are initialized to default values of their type; in particular, `uint256` defaults to 0. The condition on the available balance in the `transfer` function is negated and, in case the condition is satisfied, the transaction is reverted.

5. Formal Verification

In this section, we argue that our framework can enable formal verification of smart contracts written in DatalogMTL^S. According to the taxonomy proposed by Tolmach et al. [37], our contract modeling approach can be ascribed to the *contract-level* category, which is concerned

with the high-level behavior of a smart contract under analysis and without considering technical details of its implementation and execution. In our framework, the correctness of the smart contract's on-chain behaviour and execution on the target platform primarily follows on the correctness of the compilation step and on the correctness of the DatalogMTL^S program.

An example of how formal properties can be verified is by additional program rules that formalize *invariants* in the DatalogMTL^S language. These are formalized using rules of the form $\bot \leftarrow A_1, \dots, A_m$, meaning that, if all expressions $A_1, \dots, A_m$ are true (that is, the invariant does not hold), then the function call that triggered that rule must be reverted. These rules are compiled into `assert` instructions and can be seen as a runtime validation technique that rejects all transactions that violate the invariants, as in [38]. For example, In the ERC-20 smart contract, we might add some invariant conditions that must be true at any time during the lifetime of the smart contract, such as:

•The sum of user balances is equal to *totalSupply* [39] (the operator *msum* is an aggregation operator that computes the sum, see [40, 26])

$$\mathit{actualTotalSupply}(\mathit{msum}(\langle\langle(B)\rangle\rangle) \leftarrow \mathit{balanceOf}(\_, B).$$
$$\bot \leftarrow \mathit{actualTotalSupply}(N_1), \mathit{totalSupply}(N_2), N_1 != N_2.$$

•The sums of sender and receiver balances before and after the transfer are equal [41]:

$$\bot \leftarrow \diamondsuit_{[0,1]} \#\mathit{transfer}(), \mathit{address}(\mathit{to}), \diamondsuit_{[0,1]} \mathit{balanceOf}(\$\text{sender}, B_s), \diamondsuit_{[0,1]} \mathit{balanceOf}(\mathit{to}, B_r),$$
$$\mathit{balanceOf}(\$\text{sender}, B'_s), \mathit{balanceOf}(\$\text{sender}, B'_r), B_s + B_r = B'_s + B'_r.$$

More sophisticated formal verification approaches could be developed specifically for DatalogMTL programs. For example, we could extend the approach for program verification based on using *Constrained Horn Clauses (CHCs)* [42, 43], already used by the *Solidity Compiler's Model Checker (SolCMC)* [44], to support also temporal operators.

6. Conclusion

This preliminary work proposes a framework for expressing and evaluating smart contracts, focusing on improving the explainability and transparency of traditional smart contract development. We achieve this by building on decades of research in KRR, particularly in declarative logic-based programming, leveraging the expressive power of DatalogMTL. Inspired by a recent application of such formalism for modelling smart contracts [45, 16], we generalize those approaches and develop a foundational framework which supports the formalization and evaluation of arbitrary smart contracts. While in this paper we focused more on the vision and its industrial applications in the financial sector, we can already foresee many research avenues as future works. First, we aim to develop novel techniques (e.g., zero-knowledge techniques for off-chain execution) and implementations for realizing each execution mode. Next, it would be interesting to investigate the impact of developing smart contracts in DatalogMTL^S, in terms of ease of use, explainability, and code quality. Finally, we want to devise and apply formal verification techniques for DatalogMTL^S smart contracts.

References

[1] J. I. Hong, Teaching the fate community about privacy, Commun. ACM 66 (2023) 10–11.

[2] Á. A. Cabrera, E. Fu, D. Bertucci, K. Holstein, A. Talwalkar, J. I. Hong, A. Perer, Zeno: An interactive framework for behavioral evaluation of machine learning, in: CHI, ACM, 2023, pp. 419:1–419:14.

[3] T. Li, Y. Agarwal, J. I. Hong, Coconut: An IDE plugin for developing privacy-friendly apps, Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. 2 (2018) 178:1–178:35.

[4] E. Napoletano, B. Curry, What is defi? understanding decentralized finance, http://bitly.ws/xd7Y, 2021. Last accessed on 2022-11-28.

[5] I. Swaps, D. Association, Legal guidelines for smart derivatives contracts: the isda master agreement, 2019.

[6] P. E. Naeini, J. Dheenadhayalan, Y. Agarwal, L. F. Cranor, An informative security and privacy "nutrition" label for internet of things devices, IEEE Secur. Priv. 20 (2022) 31–39.

[7] J. Buolamwini, T. Gebru, Gender shades: Intersectional accuracy disparities in commercial gender classification, in: FAT, volume 81 of *Proceedings of Machine Learning Research*, PMLR, 2018, pp. 77–91.

[8] B. of Italy, Public consultation on the working document relating to phase one of the research project on smart contracts, https://bit.ly/3xQKRL8, 2023. [Online; accessed 12-Oct-2023].

[9] G. Ciatto, R. Calegari, S. Mariani, E. Denti, A. Omicini, From the blockchain to logic programming and back: Research perspectives, in: WOA, 2018, pp. 69–74.

[10] M. Li, J. Weng, A. Yang, J. Weng, Y. Zhang, Towards interpreting smart contract against contract fraud: A practical and automatic realization, Cryptology ePrint Archive, Paper 2020/574, 2020. URL: https://eprint.iacr.org/2020/574, https://eprint.iacr.org/2020/574.

[11] E. Regnath, S. Steinhorst, Smaconat: Smart contracts in natural language, in: 2018 Forum on Specification and Design Languages (FDL), 2018, pp. 5–16. doi:10.1109/FDL.2018.8524068.

[12] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, J. E. L. Gayo, S. Kirrane, S. Neumaier, A. Polleres, R. Navigli, A. N. Ngomo, S. M. Rashid, A. Rula, L. Schmelzeisen, J. F. Sequeda, S. Staab, A. Zimmermann, Knowledge graphs, CoRR abs/2003.02320 (2020).

[13] T. Baldazzi, L. Bellomarini, E. Sallinger, Reasoning over financial scenarios with the vadalog system, in: EDBT, OpenProceedings.org, 2023, pp. 782–791.

[14] L. Bellomarini, D. Fakhoury, G. Gottlob, E. Sallinger, Knowledge graphs and enterprise AI: the promise of an enabling technology, in: ICDE, IEEE, 2019, pp. 26–37.

[15] G. Gottlob, A. Pieris, Beyond SPARQL under OWL 2 QL entailment regime: Rules to the rescue, in: IJCAI, 2015, pp. 2999–3007.

[16] A. Colombo, L. Bellomarini, S. Ceri, E. Laurenza, Smart derivative contracts in datalogmtl, in: EDBT, OpenProceedings.org, 2023, pp. 773–781.

[17] P. A. Walega, B. C. Grau, M. Kaminski, E. V. Kostylev, Datalogmtl: Computational complexity and expressive power, in: IJCAI, ijcai.org, 2019, pp. 1886–1892.

[18] L. Bellomarini, M. Nissl, E. Sallinger, Query evaluation in datalogmtl - taming infinite query results, CoRR abs/2109.10691 (2021).

[19] S. Ceri, G. Gottlob, L. Tanca, What you always wanted to know about datalog (and never dared to ask), TKDE 1 (1989) 146–166.
[20] L. Bellomarini, L. Blasi, M. Nissl, E. Sallinger, The temporal vadalog system, in: RuleML+RR, volume 13752 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 130–145.
[21] D. Wang, P. Hu, P. A. Walega, B. C. Grau, Meteor: Practical reasoning in datalog with metric temporal operators, in: AAAI, AAAI Press, 2022, pp. 5906–5913.
[22] T. S. Authors, Solidity documentation, https://docs.soliditylang.org/en/v0.8.21/, 2023. [Online; accessed 12-Oct-2023].
[23] B. Wiki, Script, https://en.bitcoin.it/wiki/Script, 2023. [Online; accessed 12-Oct-2023].
[24] T. Baldazzi, L. Bellomarini, S. Ceri, A. Colombo, A. Gentili, E. Sallinger, Fine-tuning large enterprise language models via ontological reasoning, in: International Joint Conference on Rules and Reasoning, Springer, 2023, pp. 86–94.
[25] D. Maier, A. O. Mendelzon, Y. Sagiv, Testing implications of data dependencies, ACM Transactions on Database Systems 4 (1979) 455–468.
[26] L. Bellomarini, M. Nissl, E. Sallinger, Monotonic aggregation for temporal datalog, in: RuleML+RR (Supplement), volume 2956 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021.
[27] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, P. McCorry, Sprites and state channels: Payment networks that go faster than lightning, in: Financial Cryptography, volume 11598 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 508–526.
[28] M. Harishankar, D. Akestoridis, S. V. Iyer, A. Laszka, C. Joe-Wong, P. Tague, Plasma go: A scalable sidechain protocol for flexible payment mechanisms in blockchain-based marketplaces, CoRR abs/2003.06197 (2020).
[29] Ethereum.org, Zero-knowledge rollups, https://shorturl.at/wIYZ4, 2023. Last accessed on 2022-11-28.
[30] T. Chen, H. Lu, T. Kunpittaya, A. Luo, A review of zk-snarks, CoRR abs/2202.06877 (2022).
[31] E. Ben-Sasson, I. Bentov, Y. Horesh, M. Riabzev, Scalable, transparent, and post-quantum secure computational integrity, IACR Cryptol. ePrint Arch. (2018) 46.
[32] F. D. Cosmo, Verification of prev-free communicating datalog programs, in: CILC, volume 3428 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023.
[33] P. Walega, B. Cuenca Grau, M. Kaminski, E. Kostylev, Datalogmtl: Computational complexity and expressive power, in: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence, 2019.
[34] B. C. Grau, I. Horrocks, M. Kaminski, E. V. Kostylev, B. Motik, Limit datalog: A declarative query language for data analysis, SIGMOD Rec. 48 (2019) 6–17.
[35] Ethereum Improvement Proposals, ERC-20: Token Standard, 2015. URL: https://eips.ethereum.org/EIPS/eip-20.
[36] C. Laneve, A. Parenti, G. Sartor, Legal contracts amending with, in: International Conference on Coordination Languages and Models, Springer, 2023, pp. 253–270.
[37] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, Z. Li, A survey of smart contract formal specification and verification, ACM Computing Surveys (CSUR) 54 (2021) 1–38.
[38] A. Li, J. A. Choi, F. Long, Securing smart contract with runtime validation, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and

Implementation, 2020, pp. 438–453.
[39] Á. Hajdu, D. Jovanović, solc-verify: A modular verifier for solidity smart contracts, in: Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11, Springer, 2020, pp. 161–179.
[40] A. Shkapsky, M. Yang, C. Zaniolo, Optimizing recursive queries with monotonic aggregates in deals, in: ICDE, 2015, pp. 867–878.
[41] L. Alt, C. Reitwiessner, Smt-based verification of solidity smart contracts, in: Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV 8, Springer, 2018, pp. 376–388.
[42] N. S. Bjørner, A. Gurfinkel, K. L. McMillan, A. Rybalchenko, Horn clause solvers for program verification, in: Fields of Logic and Computation II, volume 9300 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 24–51.
[43] A. Gurfinkel, Program verification with constrained horn clauses (invited paper), in: CAV (1), volume 13371 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 19–29.
[44] L. Alt, M. Blicha, A. E. J. Hyvärinen, N. Sharygina, Solcmc: Solidity compiler's model checker, in: CAV (1), volume 13371 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 325–338.
[45] M. Nissl, E. Sallinger, Modelling smart contracts with datalogmtl, in: EDBT/ICDT Workshops, volume 3135 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022.