Smart Derivative Contracts in DatalogMTL

Andrea Colombo
Politecnico di Milano°

Luigi Bellomarini
Banca d'Italia*

Stefano Ceri
Politecnico di Milano°

Eleonora Laurenza
Banca d'Italia*

ABSTRACT

Derivative contracts, either from traditional finance or reproduced in the crypto world in the form of smart contracts, are a very popular financial instrument widely used in the industry and of great interest to supervisors, financial intermediaries, and investment banks. With the expansion of Decentralized Finance (DeFi) environments, where no or little intermediation is involved, these contracts are also being deployed into a technological infrastructure, the blockchain, in the form of executable scripts. Unfortunately, these scripts are often criticized as their business logic is very complex, also due to the need to handle the temporal dimension, the implementations are hard to explain and communicate, and the adopted languages are highly procedural and very technical. Consequently, DeFi derivatives are often untrusted by non-IT users and hard to supervise. This joint work of the researchers of the Central Bank of Italy and the Polytechnic University of Milan aims at bridging this gap by presenting the first declarative, logic-based, and executable implementation of a DeFi derivative contract. In particular, leveraging the extensive experience of the database and reasoning-based AI communities about logical languages for knowledge representation and reasoning, we adopt the DatalogMTL language for temporal reasoning to encode the Ethereum Perpetual Future (ETH-PERP) smart contract. We show how the language conveys simplicity, understandability, and transparency for non-technical users. To validate our implementation, we execute the smart contract in Vadalog, a modern reasoner supporting DatalogMTL.

KEYWORDS

derivatives, smart contracts, Datalog, DatalogMTL, reasoning

1 INTRODUCTION

A derivative can be defined as a financial instrument that derives its value from one or more underlying variables [23]. This instrument takes the form of a contract between counterparts who agree on some terms and schedules. The underlying variable of the contract can be any object, also digital. In most cases, the underlying variables are the prices of traded assets, such as stocks or indexes [37]. There are many different types of derivatives, which differ in terms and conditions. *Futures* are a type of derivative contract, where the parties agree to buy (or sell) a standardized asset with a predetermined date and price. For example, futures are used in the energy market by distribution companies that wish to secure a price and the supply of raw materials, such as gas, from producers for a future month, covering the risk of high increases in prices. However, derivatives are also very frequently used for mere trading purposes, without any real interest in the underlying asset. With the spread of *blockchain* technology and *Decentralized Finance* (DeFi), new technologies and ways of trading derivatives are emerging. DeFi refers to financial transactions conducted without any intermediation but through a computer code on a decentralized public ledger [31], which has become extremely popular with its applications in the crypto world. Central to the functioning of a DeFi application is the formulation of contracts in a machine-readable format, a so-called *smart contract* [38]. The standard way of writing such contracts is by using a procedural language that encodes each contractual term and defines what the program should do in each step, addressing the hard challenge of explicitly handling all the time-related aspects. Many stakeholders are investigating the potential of deploying traditional derivative contracts in the form of smart contracts and in a DeFi environment, with efforts mainly dedicated to the implementation of financial smart contracts in existing popular imperative languages such as the scripting language *Solidity* for the *Ethereum* ecosystem [29] or in new ad-hoc ones such as *Daml* [24]. However, existing approaches are receiving increasing criticisms from the community because of overly complex business logic, little explainability and consequential lack of transparency of the contract, which is also hard to describe and communicate, being often unsuitable for non-IT users [12, 25, 36]. From their perspective, also supervision authorities, including central banks and other national authorities, would greatly benefit from increased transparency of these objects in their manifold roles, such as enhancing the stability in financial markets [9]. Nevertheless, little is being done in the literature and in the technical community to address such concerns, with more focus on developing ways and tools to use existing paradigms more easily rather than exploring new ones [42].

Contribution. This joint work of the researchers of the Central Bank of Italy and the Polytechnic University of Milan capitalizes on the growing experience about *logic-based languages for temporal reasoning* from the database and AI communities and proposes the first declarative implementation of a derivative contract, the *ETH Perpetual Future* (ETH-PERP) smart contract, developed by the *Syntethix community* on the *Kwenta platform* [18]. In particular, our main contributions are:

- An **encoding of the ETH-PERP** that uses the recently introduced *DatalogMTL* language for **temporal reasoning** [39]. DatalogMTL is an extension with *Metric Temporal Logic* (MTL) of the famous *Datalog* language of databases [11]. Through the ETH-PERP industrial case study, we show how thanks to a *non-trivial joint use of temporal operators and recursion*, with DatalogMTL we can seamlessly handle the complex temporal aspects of a derivative smart contract.
- A **systematic description of our rules**, showing in practice that our approach achieves the goal of building an *explainable* and *executable* derivative contract. As a side-product, we offer a **simple reformulation and a walk-through of the ETH-PERP business logic**, which we consider valuable for the broader community and has been so far unavailable, given the only presence of a Solidity encoding of the ETH-PERP contract.

° Department of Electronics, Information and Bioengineering.
* The views and opinions expressed in this paper are those of the authors and do not necessarily reflect the official policy or position of the Bank of Italy.

- A **validation** of our implementation by executing the DatalogMTL code within Vadalog [7], a state-of-the-art system supporting temporal reasoning.

Organization. The rest of the paper is organized as follows. In Section 2 we lay out the preliminaries about DatalogMTL, its implementation in the Vadalog system and we introduce some relevant concepts about derivatives and their implementation as smart contracts. In Section 3 we provide the core contributions, describing our DatalogMTL implementation of the ETH-PERP contract. Section 4 is dedicated to the empirical evaluation. Section 5 concludes the paper.

2 PRELIMINARIES

In this section, we recall the syntax and semantics of DatalogMTL and we briefly introduce the Vadalog reasoner, which will be used in our experiments. We then give some general concepts about derivatives and their implementation as smart contracts.

2.1 DatalogMTL

DatalogMTL [8, 41] is a recently introduced extension of Datalog [11] with operators from *Metric Temporal Logic* (MTL) interpreted over the rational timeline and with stratified negation under stable model semantics. In this section, we recap the syntax and semantics of DatalogMTL.

Syntax. DatalogMTL uses the MTL operators $\boxminus$, $\boxplus$, $\diamondminus$, $\diamondplus$, $\mathcal{S}$, and $\mathcal{U}$ to extend Datalog to perform temporal reasoning tasks. These operators are indexed with intervals ϱ with non-negative bounds [39]. An interval is of the form $\langle t_1, t_2\rangle$, with parentheses and square brackets allowed in all possible combinations and $t_1, t_2 \in \mathbb{Q} \cup \{-\infty, \infty\}$, such that $t_1 \leq t_2$; it is *punctual* if $t_1 = t_2$. A *relational atom* is an expression of the form $P(s)$, where P is a predicate of arity n and s is a tuple of variables or constants matching the arity. A *metric atom* M extends a relational atom by allowing MTL operators. It is defined by the following grammar:

$$M ::= \top \mid \bot \mid P(s) \mid \boxminus_\varrho M \mid \boxplus_\varrho M \mid \diamondminus_\varrho M \mid \diamondplus_\varrho M \mid M \mathcal{S}_\varrho M \mid M \mathcal{U}_\varrho M$$

where $P(s)$ is an atom and s is a positive interval. A rule is an expression of the form:

$$M_1 \wedge \cdots \wedge M_k \wedge \neg M_{k+1} \wedge \cdots \wedge \neg M_{k+m} \rightarrow M'$$

for $k, m \geq 0$ and where $M_1, \ldots, M_{k+m}$ are *literals* (jointly referred to as the *rule body*) and M' (*rule head*) is specified as:

$$M' ::= \top \mid \bot \mid P(\boldsymbol{s}) \mid \boxminus_\varrho M' \mid \boxplus_\varrho M'$$

where $P(s)$ ranges over relational atoms and ϱ over positive intervals. The literals $M_1, \ldots, M_k$ are positive body literals of the rule, and $M_{k+1}, \ldots, M_{k+m}$ are the negated ones. If each variable occurs in some positive body atom, the rule is *safe*. If no variable occurs, the rule is called *ground*. A DatalogMTL program is a finite set of safe rules. In the rest of the paper, we will refer to DatalogMTL$^{\text{FP}}$ [40] programs, where only the $\boxminus$, $\diamondminus$ operators are allowed and will not make use of the $\mathcal{S}$, and $\mathcal{U}$ operators. Moreover, we will always refer to the $[1, 1]$ interval and therefore omit the respective operator subscript.

A *stratification* of a program Π organizes Π in layers of subprograms, the *strata*, and can be formally defined as a function σ that maps predicates of Π into positive integers such that, for each rule $r \in \Pi$ and all predicates P: $\sigma\left(P^+\right) \leq \sigma(P), \sigma\left(P^-\right) < \sigma(P)$, where P^+ is a positive body literal, P^- is a negated body literal and P is the head of the rule [19]. A *fact* is an expression of the form $P(\boldsymbol{s})@\varrho$, where $P(\boldsymbol{\tau})$ is ground and ϱ a non-empty interval. A *database* is a finite set of facts.

Semantics. The semantics of a DatalogMTL$^{\text{FP}}$ program (and we shall omit the *FP* superscript from hereinafter) is given by an interpretation $\mathfrak{M}$ that specifies for each time point $t \in \mathbb{Q}$ and for each ground atom $P(\boldsymbol{a})$, whether $P(\boldsymbol{a})$ is satisfied at t, in which case we write $\mathfrak{M}, t \models P(\boldsymbol{a})$. An interpretation $\mathfrak{M}$ is a model of a fact $P(\boldsymbol{a})@\varrho$, if $\mathfrak{M}, t \models P(\boldsymbol{a})$ for all $t \in \varrho$ and a model of a set of facts (or a *database*) D if it is a model of each fact in D.

This notion extends to ground literals M. Informally speaking, the temporal operators $\diamondminus_\varrho$ (*diamond minus*) and $\boxminus_\varrho$ (*box minus*) define the satisfaction of M by $\mathfrak{M}$ at t, based on the satisfaction of M by $\mathfrak{M}$ in some past time intervals. In particular, $\mathfrak{M}$ satisfies $\boxminus_\varrho M$) at t, if $\mathfrak{M}$ continuously satisfies M in the interval $\langle t - \varrho^+, t - \varrho^-\rangle$ and $\mathfrak{M}$ satisfies $\diamondminus_\varrho M$ at t, if $\mathfrak{M}$ satisfies M at least once in the interval $\langle t - \varrho^+, t - \varrho^-\rangle$, where ϱ^- and ϱ^+ are the lower and upper bounds of ϱ. In this sense, the "minus operators" are *forward propagating*, whence the superscript *FP*. More formally:

$$\begin{array}{ll}
\mathfrak{M}, t \models \top & \text{for each } t \in \mathbb{Q} \\
\mathfrak{M}, t \models \bot & \text{for no } t \in \mathbb{Q} \\
\mathfrak{M}, t \models \boxminus_\varrho M & \text{iff } \mathfrak{M}, s \models M \text{ for all } s \text{ with } t - s \in \varrho \\
\mathfrak{M}, t \models \diamondminus_\varrho M & \text{iff } \mathfrak{M}, s \models M \text{ for some } s \text{ with } t - s \in \varrho
\end{array}$$

Also, $\mathfrak{M}$ satisfies $\neg M$ ($\mathfrak{M}, t \models \neg M$), if $\mathfrak{M}, t \not\models M$. An interpretation $\mathfrak{M}$ is a *model of a rule* if it satisfies every possible grounding of the rule, and a *model of a program*, if it satisfies every rule in the program and the program has a stratification.

Is it easy to observe that in the presence of punctual intervals, the semantics of $\diamondminus$ and $\boxminus$ coalesces and either operator can be adopted interchangeably. Towards understandability, our preference is to use both, and let the choice be guided by the conceptual meaning of the operator from case to case, which may either refer to a continuous validity ($\boxminus$) or to an eventual one ($\diamondminus$).

A program Π and a database D entail a fact $P(\boldsymbol{a})@\varrho$ ($(\Pi, D) \models P(\boldsymbol{a})@\varrho$) if $\mathfrak{M} \models P(\boldsymbol{a})@\varrho$ for each model of both Π and D. In operational and informal terms, we will refer to the *execution* (or run) of a program Π with a database D as the process of augmenting D by adding to it all the facts entailed by Π and D. To do so, reasoning systems typically adopt CHASE procedures [26].

2.2 Temporal Vadalog System

Vadalog is a Datalog-based engine for performing complex logic reasoning tasks [6]. The system adopts the Vadalog language, an extension of the *Warded Datalog*$^{\pm}$ [22] tractable fragment of the Datalog$^{\pm}$ [10] family of languages. In particular, it features existential quantification, full recursion, aggregations, and other features of practical utilities. A recent evolution of the system, the *Temporal Vadalog System*, supports DatalogMTL, allowing full recursion and aggregation also in the temporal context [5, 7].

2.3 Derivative Contracts

Derivatives represent a very relevant instrument in the financial world. The latest figures by the Bank for International Settlements (BIS) report a total notional value of all derivatives trades amounting to $600 trillion at the end of 2021 [33]. These trades refer to either exchange markets, where standardized contracts are traded, or over-the-counter markets (i.e., outside the market), where counterparts can agree on their own terms or define new contracts. Traditional *futures* are standardized contracts where parties agree to sell or buy an asset at a specific price and date in the future. These terms are usually predefined by exchange markets, which offer these instruments to their customers. However, it is always possible to trade futures over-the-counter under the same or different conditions. One way to do so can be through the use of *smart contracts*, computer programs consisting in a

set of rules which are run on a blockchain [27]. Smart contracts are widely used in the crypto world for a variety of purposes, such as for writing derivatives whose underlying is a crypto asset (mostly cryptocurrencies, such as Ether or Bitcoin). These kinds of derivatives are called *Smart Derivative Contracts* and are attracting the interest of the financial industry, which is exploring the possibility of applying this technology also in traditional derivatives like futures [28]. In the rest of the paper, we will focus on a "*future-like*" derivative contract (i.e., a perpetual future) written in the form of a smart contract in *Solidity*, the scripting language of the Ethereum ecosystem [20].

Futures, Perpetuals and Spot Market. Choosing a derivative instead of a spot market (i.e., directly buying an asset like Ether) and, in particular, a perpetual future, allows speculating on the future price of a given asset by buying ("going long") or selling ("going short") those contracts. *Leverage* is the most common reason why traders are attracted to futures: it works as a so-called capital multiplier, which implies that to take a position, the entire sum of money need not be provided, but a fraction of it is sufficient. The difference between perpetual and typical futures is that the latter have an expiration date, while the perpetual ones do not expire until the trader closes her position. Consequently, there are no daily settlement events (i.e., daily computation and settling of returns until the expiration date) that ensure convergence between future price and underlying asset. Perpetual futures have instead a *funding rate mechanism*, which incentivizes the market skew to remain balanced (equal long/short open interest), ensuring convergence on a regular basis [21].

Related Work. The idea of using DatalogMTL to encode generic smart contracts has been recently suggested [32]. We position ourselves in that line. We move a step forward and look into a real-world smart contract in the financial field. To the best of our knowledge, with this work, we offer for the first time an executable declarative implementation of a real smart derivative.

The interest in smart derivative contracts does not limit to the stakeholders wishing to offer a new way of conducting trading. Big financial groups, banks, and clearing houses are also bound to benefit from smart derivatives as well, as reflected in the different ongoing research tracks. Some of them focus on the automation of the risk management activities related to derivatives, such as the timely adjustment of the collateral to fulfill margin requirements [28]. Interesting debates on which parts of a derivative contract should be automated are ongoing [13]. A clear advantage of full automation is the reduced time and resources spent for fulfilling regulatory report obligations [29], with benefits for all actors involved, from intermediaries to authorities. This is especially true when automation meets flexible tools and technology that can be effectively integrated with AI-based solutions [4].

Looking further, related research tracks are those pursuing the legal aspects of implementing derivatives in a blockchain technology, which could deeply affect the work in this area [34].

3 CASE STUDY: ETH-PERP IN DATALOGMTL

Kwenta [17] is a decentralized derivatives trading platform, deployed in a blockchain, which offers access to real-world synthetic assets. In particular, Kwenta allows traders to get exposure to a decentralized future market of cryptocurrencies, especially the Ether one. The main product the *Kwenta* platform offers is the ETH-PERP smart contract, which is a perpetual future on the Ether cryptocurrency deployed in the *Optimism Mainnet* [1] (i.e., one of the Ethereum blockchains) under a *Solidity* encoding. In particular, the original implementation in Solidity involves over 3k lines of code [15]. As discussed in specialized forums, a Solidity-based implementation of a smart contract—and similarly for any other major language used in the crypto world—does not allow a common understanding for all the parties involved: a contract should in fact "*provide natural language terms that are already used and understood in the real world, with the aim of easing human reasoning*" [36]. In this work, we want to leverage the clarity, understandability, modularity, simplicity, and shared-semantics characteristics of a declarative database language like Datalog to address this issue. In particular, we look at DatalogMTL to open up a new way forward for the encoding of derivative smart contracts, using the ETH-PERP as a use case that we claim to be representative of the complex time-dependent business logic governing these kind of contracts.

The ETH-PERP market. *Kwenta* launched the ETH-PERP smart contract in March '22 and, as of November '22, it has attracted the interest of a large community, with 27k unique traders, who have performed around 100k trades for a total volume of 3.4$ billions. The market is continuously growing, with many new traders joining in, especially since September '22 [14].

3.1 The DatalogMTL Program for ETH-PERP

We first lay out the domain constraints related to our specific case—perpetual futures on *Synthetix*, the protocol *Kwenta* is based on—and then illustrate our implementation strategy.

Domain Constraints. A position on the Kwenta smart contract may be long or short. If its sign is positive, the position is considered *long* and it increases in value as the price of the underlying asset rises. If it is negative, the position is called *short* and its value increases as the underlying asset loses value. The Kwenta protocol requires that all trades are opened against a so-called *pooling counterparty* (i.e., the smart contract itself) and that a specific account cannot open more than one position at once (but an existing one can be modified). The price of the ETH-PERP is obtained from an *external oracle*, as usual in smart contracts.

Implementation Strategy. We implement ETH-PERP as a DatalogMTL program Π. We adopt a memory-resident execution model, in which the program runs in a Vadalog process that continuously takes as input the actions that the users (=traders) send to the smart contract, namely *methods*, and updates multiple *state amounts* as a consequence, such as users' positions, margins as well as the overall market along with its metrics. This simulates the usual functioning of a smart contract, which basically operates as a stateful system. More in detail, the state of Π is represented by a database D of temporal facts, where the user inserts the input facts to call the methods. Method calls are encoded by relational facts of the predicates representing the method. As a result of its execution (see Section 2), Π augments D by adding the entailed facts, which represent state updates.

Notice that, importantly, the adoption of DatalogMTL allows an entirely monotone reasoning process, in which neither deletions nor updates of facts in D are needed: as they are temporally annotated, insertions are sufficient to model the state evolution.

Example 3.1. For example, let us assume that D currently contains the fact *margin(123abc,97$)@2022-11-09*, which specifies the margin in dollars for the account 123*abc* as of yesterday, 2022-11-09. Today, the user owning that account calls the method *tranM* by adding the following fact to D: *tranM(123abc, 3$)@2022-11-10*. It represents an order of deposit of 3 dollars on the account *123abc*. The program applies its internal business logic and updates the current status by adding the fact *margin(123abc,100$)@2022-11-10* to D, to mean that the margin of the account 123*abc* has risen to 100$ as of today.

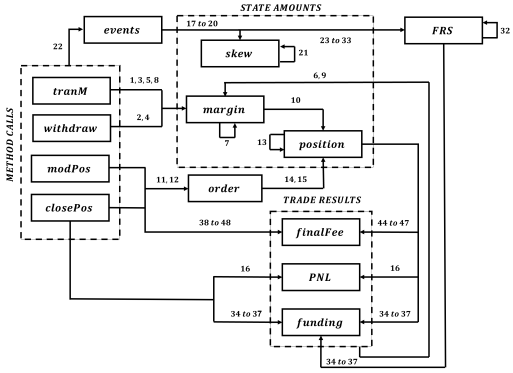

Figure 1: Simplified dependency graph of our DatalogMTL program. Arrows denote rule application.

Overview of the DatalogMTL Program. We organized the DatalogMTL programs into the following different modules:

- MARGIN. Opening a position in the ETH-PERP smart contract requires a margin account. In other words, before entering the market, each user should have some funds transferred into a sort of investing account, which we call *margin*. Only after this action, a position, either long or short, can be opened. This module captures the logic to handle margin (Section 3.3).
- POSITION. Whenever a trader sends an order, she is changing her position size. We want to track in each time point the state of the system and execute orders. All of this is possible through the use of an *order-book* (Section 3.4).
- RETURNS. When closing a position, the returns derived from the trading activity are computed (Section 3.5).
- F-RATE. The funding rate is a balancing incentive for the two sides of the market and replaces the daily settlement mechanism of traditional futures. According to this mechanism, positions on the heavier side of the market are charged a funding rate, while positions on the lighter side receive funding. The side depends on the skew of the market: if the skew is positive (i.e., the sum of the *long* positions is greater than the sum of the *short* ones), the heavier side will be the *long* one and they will be charged funding, while accounts in a *short* position will receive funding. We operationally define the *individual funding* as the one computed with an instantaneous funding rate charged over time against the notional value of each position, and paid into or out of its margin. This means that funding accrues continuously until the trader closes its position (Section 3.6).
- FEES. Each interaction with the smart contract is subject to a fee. This is generally in line with all financial transactions, although in DeFI the purpose of fees is different, as they are charged for the prevention of spam actions and for the maintenance of the blockchain infrastructure [35], instead of being a remuneration for the intermediation role. This is also true within the ETH-PERP smart contract, in which two kinds of fees are charged: a *taker* fee rate, ϕ_t, and a *maker* fee rate, ϕ_m (Section 3.7).

Before delving into the modules, we start in Section 3.2 by showing the predicates capturing methods and state amounts.

3.2 Input Methods and State Amounts

A trader who wants to interact with the ETH-PERP smart contract and open a position will use the following **methods**:

- *Transfer Margin.* A call to this method, encoded as a fact of the form *tranM(A, M)@t*, is an order of transferring some funds (M) into the smart contract issued by user A at time t. For example, *tranM(123abc,20$)@2022-11-03* is an order to deposit 20$ at time *2022-11-03* into the account with ID *123abc*.
- *Withdraw Margin.* A withdrawal order at time t, *withdraw(A)@t*, indicates that the account is being closed and all available funds are withdrawn. For example, the fact *withdraw(123abc)@2022-11-13* implies the shutdown of account *123abc*.
- *Modify Position.* A fact *modPos(A, S)@t* entails that an account A is opening a new position (either long or short depending on the sign) or modifying an existing one by S units. For example, with the atom *modPos(123abc, 0.5)@2022-11-06* we mean an order of opening a long position of size 0.5 for *123abc*.
- *Close Position.* A fact *closePos(A)@t* is an order by user A to close its position implies the computation of returns, funding, and fees. For example, *closePos(123abc)@2022-11-16* is an order by the trader to close its previously opened position at *2022-11-16*.

Moreover, we use facts *price(P)@t* to track the ETH-PERP price.

The program encodes the following **state amounts**.

- *Margin.* Facts *margin(A,M)@t* track an open margin account over time, updating M in case of later events, such as deposits. For example, the fact *margin(123abc,75$)@2022-11-08* means that the margin account *123abc* is open and it amounts to 75$.
- *Position.* The *position(A,S,N)@t* facts follow the evolution of each user's position. For example, the fact *position(123abc, -0.14, -68$)@2022-10-16* implies that the account *123abc* has a short position and that its notional value is −68$.
- *Skew of the market.* We use facts *skew(K)@t* to track the updates of the market. For example, if D contains *skew(354.8)@2022-10-03*, it means that on that day the skewness is positive.

We now have all the ingredients to describe the rules of the modules, in the next sections. Figure 1 summarizes the main dependencies between the predicates of our DatalogMTL program. We finally provide some further discussion in Section 3.8.

3.3 Margin Management (MARGIN)

The following rules of the MARGIN module model the first time an account is opened as well as the later deposits.

$$tranM(A, M) \rightarrow isOpen(A) \tag{1}$$

$$\boxminus isOpen(A), \neg withdraw(A) \rightarrow isOpen(A) \tag{2}$$

$$tranM(A, M), \neg \boxminus isOpen(A) \rightarrow margin(A, M) \tag{3}$$

$$withdraw(A) \rightarrow changeM(A) \tag{4}$$

$$tranM(A, M) \rightarrow changeM(A) \tag{5}$$

$$closePos(A) \rightarrow changeM(A) \tag{6}$$

$$\diamondminus margin(A, M), \neg changeM(A) \rightarrow margin(A, M) \tag{7}$$

$$\boxminus isOpen(A), \diamondminus margin(A, X), tranM(A, Y), M = X + Y \rightarrow margin(A, M) \tag{8}$$

$$\diamondminus margin(A, X), PNL(A, PL), finalFee(A, C), funding(A, IF), M = X + PL - C + IF \rightarrow margin(A, M) \tag{9}$$

Rules 1 and 2 define the predicate *isOpen* that holds when the margin for user A is first opened. It is then recursively propagated over time until there is a *withdraw* method call, which may close the account. Rule 2 shows a pattern of non-trivial but elegant and light joint use of recursion and temporal operators, which will be extensively adopted in our implementation: the idea is shifting the validity interval of facts binding to the body atom with the temporal operator (in this case facts for *isOpen*, holding at $t - 1$ as specified by the $\boxminus$ operator) by generating a new fact

of the same atom that holds as of t. The recursion makes this propagation progress, until the body applies, which in this case is controlled by the $\neg withdraw$ condition.

The atom *isOpen* is instrumental in understanding if a *tranM* call is a first-time deposit or a later one. In fact, rule 3 activates only when the margin account is being opened, initializing it with the to M. Rules 4, 5 and 6 specify alternative conditions indicating a margin update. Rule 4 stops the temporal propagation of the margin when a withdrawal happens. Rule 5 accounts for additional deposits or partial withdrawals. Rule 6 together with rule 9 adds or subtracts returns, funding, and fees to the margin derived from the participation in the market. These rules interact with many predicates that will be defined in later modules. Rule 7 recursively propagates the margin over time when no changing event is occurring. Rule 8 deals with later deposits and updates the margin accordingly, activating in combination with rule 5. We have already seen some of these rules in action in Example 3.1.

3.4 Order-book and Positions (POSITION)

The ORDER module comprises the rules governing the order-book and the tracking of each position over time:

$$tranM(A, M), \neg \boxminus isOpen(A), \\ S = 0, N = 0 \rightarrow position(A, S, N) \quad (10)$$

$$modPos(A, S) \rightarrow order(A, S) \quad (11)$$

$$closePos(A), S = 0 \rightarrow order(A, S) \quad (12)$$

$$\diamondminus position(A, S, N), \neg order(A, \_), \\ isOpen(A) \rightarrow position(A, S, N) \quad (13)$$

$$\diamondminus position(A, Y, Z), price(P), \\ modPos(A, X), S = X + Y, \\ N = Z + X * P \rightarrow position(A, S, N) \quad (14)$$

$$closePos(A), S = 0, N = 0 \rightarrow position(A, S, N) \quad (15)$$

Rule 10 initializes the position of account A as soon as the margin account has been opened. S is the size of the position; the notional value N is the corresponding value in USD, allowing the computation of returns from the trading activity. Rules 11 and 12 collect into a single predicate all the orders from the counterparts (i.e., the order-book): in case of a *modPos* call, the position will be modified by S units while, when a *closePos* order is sent, the order-book will register a reset of the position. Rule 13 activates in all time points when there are no orders and so shifts existing positions over time until the margin account is closed. Rule 14 deals with the modification of the position size and updates both S and N. Rule 15 closes a position, resetting S and N.

Example 3.2. Let us suppose a user sends a *tranM(123abc, 60$@2022-11-10* order, a *modPos(123abc, 0.4)@2022-11-12* order, and that D does not contain a *isOpen(123abc)@2022-11-10* fact, which means, from the previous block, that the margin has not been opened yet. The first order generates the atoms *position(123abc, 0, 0$)@2022-11-10* and *position(123abc, 0, 0$)@2022-11-11*, while the latter adds *position(123abc, 0.4,28$)@2022-11-12* to D, with 28$ being the notional value (computed using the price) of a position of size 0.4.

3.5 Profits and Losses (RETURNS)

Let q be the position size measured in units of the base asset (i.e., ETH), and let p_t be the price of the asset (i.e., the ETH-PERP contract) at time t. The notional value v_t can be computed as $v_t = qp_t$. In other words, v_t is the (signed) dollar value of the base currency units on a position. Long positions will have positive notional, shorts will have negative notional. Using the subscript e to define the entry time point, the profit or loss of a position at time t can be defined as $r_t = v_t - v_e$.

Hence, the computation of returns (PNL) from the trading activity can be defined by a single rule:

$$closePos(A), \boxminus position(A, S, N), price(P), \\ PL = S * P - N \rightarrow PNL(A, PL) \quad (16)$$

Rule 16 activates when the settlement takes place (i.e., a *closePos* call) and computes the returns as a difference between the current notional value and the starting one.

Example 3.3. Let us suppose that D contains the facts *position(123abc, 0.7, 39$)@2022-10-06*, *price(47$)@[2022-10-07]* and *closePos(123abc)@2022-10-07*. Rule 16 adds the atom *PNL(123abc,-6.1$)@2022-10-07* to D, computing the loss generated by the trade.

3.6 The Funding Rate Mechanism (F-RATE)

Before formally explaining the mechanism, we introduce in Figure 2 some market metrics that will ease understanding.

Metric	Formula
Market Size	$Q = \sum_{c \in C} \lvert q^c \rvert$
Market Skew	$K = \sum_{c \in C} q^c$
Max Funding Rate	$i_{max} = 0.1$
Max Proportional Skew	$W_{max} = \frac{300000000}{p_t}$
Istantaneous Funding Rate	$i_t = clamp(\frac{-K_{t-1}}{W_{max}}, -1, 1)\frac{i_{max}}{86400}$

Figure 2: Market metrics.

The function *clamp* restricts the first term to be in the range $(-1, 1)$ and 86400 are the epochs in seconds that make up 1 day. The funding rate i can be either positive or negative, depending on the skew of the market, and it changes whenever someone interacts with the smart contract. When i is positive, it means that "longs pay shorts", while when it is negative "shorts pay longs". The funding flow per base unit at time t is then $f(t) = i_t p_t$.

As funding accrues continuously, the individual (cumulative) funding to be paid to a position opened at t_i and closed at t_j is:

$$IF_q = q \int_{t_i}^{t_j} f(t)dt = q[F(t_j) - F(t_i)] \quad (a)$$

with $j > i$. The funding flow between t_i, t_j can be easily computed with a summation of smaller intervals, such that in each of them the skew and the price are constant. We can rewrite:

$$\int_{t_i}^{t_j} f(t)dt = \sum_{k \in K} i_{t_k} p_{t_k} (t_k - t_{k-m}) \quad (b)$$

where K is the set of time points $k \in [i, j]$ corresponding to interactions with the smart contract; m is the difference between two consecutive k values. To simplify the computation of the individual funding, ETH-PERP lets the cumulative funding flow be updated only when the skew changes. The base asset price can in fact change between two events, but any inaccuracy induced as a result of this assumption can be neglected [16]. Therefore, the accumulated funding per base unit can be simply seen and computed as a finite time series, namely, the *funding rate sequence* (FRS), which is the set of $F(t_k) = \sum_n^k i_{t_n} p_{t_n} (t_n - t_{n-m})$ since the market started, and is updated any time someone interacts with the contract. Individual funding will be then computed by accessing the correct epochs of the time series and settled

whenever a position is closed. Let us see how individual funding can be computed with the next examples.

Example 3.4. Let us assume that the market opened at t_0 and that the account A opens a position q_a at t_1 and closes it at t_4. Now, B interacts with the contract at t_2: the funding rate sequence will be consequently updated at t_1, t_2, and t_4, generating $F(t_1), F(t_2)$ and $F(t_4)$, respectively. The first value $F(t_1)$ will be computed as $F(t_1) = i_{t_1} p_{t_1}(t_1 - t_0)$. The second value of the funding rate sequence is $F(t_2) = F(t_1) + i_{t_2} p_{t_2}(t_2 - t_1)$ and the third one is $F(t_4) = F(t_2) + i_{t_4} p_{t_4}(t_4 - t_2)$. Therefore, computing the individual funding accrued by account A can be easily done as $IF_A = q_a[F(t_4) - F(t_1)]$.

Example 3.5. Let us consider again Example 3.4 and suppose that position q_a has been modified at t_3 by s units and closed at t_4. In this case the total individual funding accrued is computed as $IF_A = q_a[F(t_3) - F(t_1)] + (q_a + s)[F(t_4) - F(t_3)]$.

We are now ready to focus on encoding the computation of the set of $F(t_k)$ (i.e., the *funding rate sequence*). This is the main object needed for computing any individual funding. The rules in the first subset (rules 17 to 20) aggregate in one unique predicate all the interactions that took place with the smart contract, regardless of the user.

$$tranM(A, M), S = 0 \rightarrow event(sum(S)) \tag{17}$$

$$withdraw(A), S = 0 \rightarrow event(sum(S)) \tag{18}$$

$$modPos(A, S) \rightarrow event(sum(S)) \tag{19}$$

$$closePos(A), \boxminus position(A, S, N) \rightarrow event(sum(-S)) \tag{20}$$

These *events* define all the time points when the funding rate sequence is going to be updated. In addition, they enable the update of the market skew. Rules 17 and 18 refer to interactions with the smart contract related to margin updates and, therefore, they are not influencing the market skew (i.e., the skew is modified by $S = 0$). Rule 19 collects all the modified position events, which exactly updates the skew by S units. Conversely, a close position order does not include the size S: the system has to derive this value from the *position* atom it has computed at the previous epoch, changing its sign (rule 20). In case multiple *events* occur at the same time point, a temporal *sum*, aggregating all values of S of all accounts A and grouping by t, is applied. Aggregations appear in multiple logical contexts and the need for a careful definition of their semantics (procedural and model-based) arose in all of them. In this work, we adopt a simple stratified semantics [30], which is enough for our purposes. Such semantics easily extends to the temporal context, and *temporal aggregations* have been recently introduced in DatalogMTL [5] and are supported by the Vadalog system. The second subset of rules, 21 and 22, updates the market skew and is defined as follows:

$$\diamondminus skew(K), \neg event(\_), isOpen(\_) \rightarrow skew(K) \tag{21}$$

$$\diamondminus skew(X), event(S), K = X + S \rightarrow skew(K) \tag{22}$$

The predicate $skew(K)@t$ is initialized at the start of the market with $K = 0$ (or with the value K at the start of the interval under analysis). Rules 21 and 22 check whether an event took place at each time point and update the skew K accordingly. With K and the price available, the only missing element needed for the update of the *funding rate sequence* is the time difference in seconds between two consecutive events, $(t_j - t_i)$. A simple way to obtain it is by subtracting *Unix timestamps*. The Unix timestamp is the number of seconds that have elapsed since January 1, 1970, and it is an alternative way to represent time through integers. In the Vadalog implementation of DatalogMTL, we will adopt a *unix(t)* cast operator for conversions.

$$start@t \rightarrow Tdiff(unix(t), unix(t)) \tag{23}$$

$$\diamondminus Tdiff(T1, T2), \neg event(\_), isOpen(\_) \rightarrow Tdiff(T1, T2) \tag{24}$$

$$\diamondminus Tdiff(T1, T2), event(S)@t \rightarrow Tdiff(T2, unix(t)) \tag{25}$$

$$Tdiff(T1, T2), event(S) \rightarrow Diff(T2 - T1) \tag{26}$$

Rule 23 initializes the predicate $Tdiff(T1, T2)$ from a fact binding to $start@t$ that denotes the market starting point (or, as for the skew, the starting point of the interval under analysis). The two arguments of *Tdiff* denote the lower and upper bounds of an interval, respectively. Observe that in this rule we are "promoting" a temporal annotation to a ground value in the head, which is beyond the possibilities offered by DatalogMTL theoretical fragment, but supported by Vadalog for practical purposes. Rule 24 shifts the facts for *Tdiff* over time in case no interaction with the contract takes place and the market is open. When an interaction takes place, rule 25 activates and updates the interval bounds: the lower bound will now be the time point of the previous event, while the upper bound will be the time point of the current event. Finally, rule 26 computes the interval length since the last interaction and filters only the facts related to events.

The final subset of rules defines the *unrecorded funds* and appends the new F value to the sequence. The accrued funding sequence is initialized with the predicate $FRS(F)@t$ where $F = 0$ when the market started. Each time an interaction occurs, the accrued funding (i.e., the last element of the summation) is computed and added to F. This requires the computation of the *istantaneous funding rate*, i_t. The following rules formally describe these steps:

$$event(S), \boxminus skew(K), price(P), I = \frac{-K}{W_{max}} P \rightarrow rate(I) \tag{27}$$

$$rate(I), I > 1 \rightarrow clampR(1) \tag{28}$$

$$rate(I), I < -1 \rightarrow clampR(-1) \tag{29}$$

$$rate(I), -1 < I < 1 \rightarrow clampR(I) \tag{30}$$

$$clampR(I), price(P), Diff(T), UF = I * P * T \frac{i_{max}}{86400} \rightarrow unrFund(UF) \tag{31}$$

$$\diamondminus FRS(F), \neg unrFund(\_), isOpen(\_) \rightarrow FRS(F) \tag{32}$$

$$\diamondminus FRS(X), unrFund(UF), F = X + UF \rightarrow FRS(F) \tag{33}$$

Rule 27 computes the rate that has to be passed to the *clamp* function, as described in Section 3.6 and implemented in rules 28 to 30. Rule 31 computes the unrecorded funding UF generated since the last interaction, which we add to the last value of the funding sequence, as done by rules 32 and 33.

Individual Funding. With the funding sequence available, the individual funding is defined as follows:

$$\boxminus position(A, S, N), FRS(F), modPos(A, C), S = 0, AF = 0 \rightarrow indF(A, F, AF) \tag{34}$$

$$\diamondminus indF(A, F, AF), \neg order(A, \_) \rightarrow indF(A, F, AF) \tag{35}$$

$$\diamondminus indF(A, F, PAF), FRS(F), modPos(A, C), \boxminus position(A, S, N), AF = PAF + S(F - PAF) \rightarrow indF(A, F, AF) \tag{36}$$

$$\diamondminus indF(A, PF, AF), closePos(A), FRS(F), \boxminus position(A, S, N), IF = AF + S(F - PF) \rightarrow funding(A, IF) \tag{37}$$

Rule 34 initializes $indF(A, F, AF)@t$, which stores the individual funding AF for account A and the corresponding $F(t)$. Rule 35 shifts over time such facts in case no order is sent. Rule 36 activates when there is a $modPos(A, S)@t$ method call and computes the intermediate individual funding AF. Rule 37 computes the final individual funding IF after the closing of a trade, which we add to the margin (Rule 9).

3.7 Exchange Fees (FEES)

The distinction between *maker fee* and *taker fee* is used as another mechanism to reduce the skew of the market: a maker is someone that reduces the skew and will be applied the lower fee rate. Conversely, a taker is someone that increases the skew and will be charged a higher rate. So, when opening, modifying, or closing a position in t, different scenarios may apply, according to the order of submission and the market conditions (i.e., the skew):

$$fee_t = \begin{cases} |\Delta_q p_t \phi_t| & \text{if } K_t > 0, \Delta_q > 0 \\ |\Delta_q p_t \phi_m| & \text{if } K_t < 0, \Delta_q > 0 \\ |\Delta_q p_t \phi_m| & \text{if } K_t > 0, \Delta_q < 0 \\ |\Delta_q p_t \phi_t| & \text{if } K_t < 0, \Delta_q < 0 \end{cases}$$

where Δ_q is the increase (decrease) in the size of the position and K_t is the market skew. In an opening position event, $\Delta_q = q$, while, in case of a close position event, $\Delta_q = -q$.

In order to encode in DatalogMTL rules, we first initialize a dedicated predicate that stores the total fees for each account:

$$tranM(A, M), \neg \boxminus isOpen(A), C = 0 \rightarrow fee(A, C) \quad (38)$$

$$\diamondminus fee(A, C), \neg order(A, \_), isOpen(A) \rightarrow fee(A, C) \quad (39)$$

In detail, rule 38 creates the *fee* predicate once an account is first created and initializes it with $C = 0$, where C is the cumulative fees charged to account A. Rule 39 recursively shifts over time the predicate *fee*, in case no event is triggered by the owner of the account. In case an order is received, the following two chunks of rules will compute the fees. First, we analyze the rules for a *modPos* event, then we will focus on a *closePos* one.

$$modPos(A, S), price(P), \diamondminus fee(A, OldC), skew(K), K > 0, S > 0, C = OldC + |SP\phi_m| \rightarrow fee(A, C) \quad (40)$$

$$modPos(A, S), price(P), \diamondminus fee(A, OldC), skew(K), K < 0, S > 0, C = OldC + |SP\phi_t| \rightarrow fee(A, C) \quad (41)$$

$$modPos(A, S), price(P), \diamondminus fee(A, OldC), skew(K), K > 0, S < 0, C = OldC + |SP\phi_t| \rightarrow fee(A, C) \quad (42)$$

$$modPos(A, S), price(P), \diamondminus fee(A, OldC), skew(K), K < 0, S < 0, C = OldC + |SP\phi_m| \rightarrow fee(A, C) \quad (43)$$

We identify four possible cases depending on the order being long or short and on the market skewness. Rules 40 and 41 compute the new exchange fees C when the order is to increase (respectively, decrease) a position and the skew is positive (negative). Rule 42 and 43 refer to the cases when the skew is negative, while the order is respectively long and short.

$$closePos(A), \boxminus position(A, S, N), skew(K), price(P), \diamondminus fee(A, OldC), K > 0, S < 0, C = OldC + |SP\phi_m| \rightarrow finalFee(A, C) \quad (44)$$

$$closePos(A), \boxminus position(A, S, N), skew(K), price(P), \diamondminus fee(A, OldC), K < 0, S < 0, C = OldC + |SP\phi_t| \rightarrow finalFee(A, C) \quad (45)$$

$$closePos(A), \boxminus position(A, S, N), skew(K), price(P), \diamondminus fee(A, OldC), K > 0, S > 0, C = OldC + |SP\phi_t| \rightarrow finalFee(A, C) \quad (46)$$

$$closePos(A), \boxminus position(A, S, N), skew(K), price(P), \diamondminus fee(A, OldC), K < 0, S > 0, C = OldC + |SP\phi_m| \rightarrow finalFee(A, C) \quad (47)$$

$$closePos(A), C = 0 \rightarrow fee(A, C) \quad (48)$$

Rules 44 to 47 are similar to the previous chunk but they deal with a *closePos* event. However, as a first difference, here the size of the order, which is unknown, is retrieved through the *position* predicate (i.e., the size S). The second difference is that these rules store the final result, i.e., the final fees from a completed trade, into a new predicate, *finalFee*. In fact, rule 48 generates "reset" *fee* facts, so as to store the fees if A starts a new trade.

Example 3.6. Let us assume that D contains *skew(1342.2)@2022-11-19* and *price(1200$)@2022-11-19*, without any open position (which means D contains *position(123abc, 0, 0$)@2022-11-18* and *fee(123abc,0)@2022-11-18*). Let us consider an order to open a position, *modPos(123abc, 0.02)@2022-11-19*. As the skew is positive and the order is long, the fees are computed through rule 40, generating the fact *fee(123abc,0.084)@2022-11-19* (with $\phi_m = 0.0035$).

3.8 Discussion

To conclude the illustration of our solution, we provide some final remarks about termination and negation.

Termination. We argue that our DatalogMTL program for ETH-PERP eventually terminates. Datalog programs always terminate also in the presence of recursion, by the finiteness of symbols in D [3]. However, since we adopt algebraic operations, more care is needed as their joint use is a well-known cause of non-termination, e.g., for infinite increments. Yet, this joint use is not present in our program, where algebraic operators are confined within recursive temporal rules that produce facts with shifted validity intervals (e.g., rules 8, 9, etc.), in fact, breaking recursion.

With this premise, the only remaining cause of non-termination in DatalogMTL is the creation of infinitely many facts with increasing temporal validity, which may be the case, for instance, of our temporal recursive rules 2, 7, and so on. Our working assumption here is that, eventually, the market will be closed and all the margins withdrawn. By construction, this implies that none of the bodies of the temporal recursive rules will activate any longer, in fact preventing non-termination. In simpler words, our program runs indefinitely but shuts down gracefully.

Negation. A final informal remark about the use of negation is required. By construction, the dependency graph of our program (the one where nodes are atoms are edges denote rule-induced dependencies) does not contain cycles involving negative edges (those generated by rules having negations in the body). This condition is enough to witness that the program has a stratification [3] and therefore a consistent use of negation.

4 EXPERIMENTAL SETTINGS IN VADALOG

We evaluated our approach by executing the DatalogMTL program in the Vadalog reasoner. The primary goal of our experimental setting is *validating the correctness of the approach* by letting ETH-PERP "live and evolve" in Vadalog throughout *three 2-hours intervals having different initial conditions* and comparing the results of the trades with the real ones on the blockchain. As a secondary goal, we also want to verify good *performance*

characteristics to confirm the possibility of realistically executing the smart contract in a reasoning system like Vadalog.[1]

HW and SW Configuration. We ran all the experiments on a Windows 11 machine with AMD Ryzen 5 5500U and 8 GB 3200 MHz DDR4 memory. Vadalog has been compiled with JDK 17.0.4.

In Section 4.1 we describe the datasets we use as input and for validation. In Section 4.2 we present the experimental results.

4.1 Input and Validation Datasets

Input Dataset. In order to let our implementation of ETH-PERP realistically live and evolve in Vadalog in the given intervals, we stimulated it with the real actions performed by the users. We retrieved such actions from *Optimism Mainnet* [1], and mapped them into method calls, i.e., by generating the corresponding DatalogMTL facts (*tranM*, *modPos*, *closePos*, and *withdraw*).

Validation Dataset. To validate the results of our execution, we compared the *funding rate sequence* (i.e., the set of $F(t)$) and the results of the single trades (i.e., *PNL*, fee, and *funding*) with the corresponding values from the blockchain. More precisely, we obtained the real values by querying the *Mainnet Subgraph* [2], which is a decentralized protocol for querying blockchain data and extract measures that would be hard to obtain otherwise.

In Figure 3 we describe the input data and the analysed intervals. The number of events represents the total number of interactions with the contract that occurred during the time window. The fourth column shows how many trades have been completed during the corresponding interval. In the *Skew* column, there is the skew at the beginning of the interval.

Date	Interval (GMT)	# events	# trades	Skew
2022-09-27	10.30 - 12.30	267	59	-2445.98
2022-10-07	18.00 - 20.00	108	16	1302.88
2022-10-12	14.00 - 16.00	128	29	2502.85

Figure 3: Input data.

4.2 Experimental Results

The analysis of the funding rate sequence computation is shown in Figure 4: the Vadalog program reproduces the evolution of the real FRS with differences in the order of $1e{-}12$, so with perfect accuracy. Moreover, since FSR is a cumulative series, this might result in an error trend, which however keeps insignificant.

Besides validating the overall correspondence of our run with the real market, we also compared individual trades. To this end, we computed some summary statistics on the errors between values computed by the Vadalog program and the ones available in the Subgraph. As Figure 5 shows, for all the analyzed intervals, the DatalogMTL program returns the same results exhibiting minimal errors. This means that not only does our system simulate the market, but reacts to any user interaction exactly like the Kwenta ETH-PERP. It is finally worth remarking that this behavior is achieved through the use of temporal operators, which manage to easily handle and track changes in each time point, continuously adding temporal facts which are always up-to-date and allow the computation of all metrics.

Performance. We point out that the main ambition of this work is not high performance, but explainability, understandability and all the other forces of a declarative approach, as we have discussed. Nevertheless, to plausibly confirm a prospective execution of a smart contract in a reasoning environment, we verified

[1]The full DatalogMTL program and all the used datasets are online: http://bitly.ws/xhGq. Vadalog can be made available by the authors upon request.

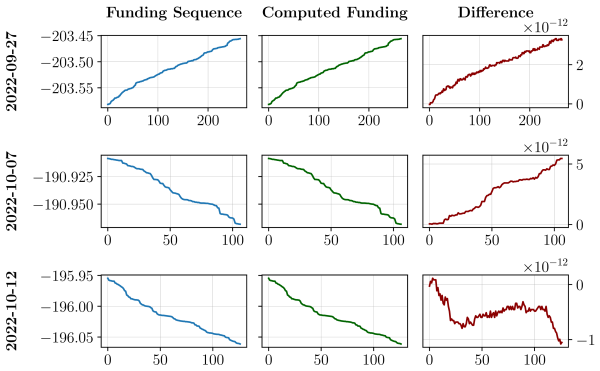

Figure 4: Comparison between the Subgraph FRS and the DatalogMTL FRS. In the third column we show the difference.

	Returns	Fee	Funding
Mean	3.545513e-15	-9.093255e-17	-4.789471e-15
Std. Dev.	5.574446e-14	3.767407e-16	1.200630e-13

Figure 5: Mean and standard deviation of errors between metrics computed with Vadalog and the Subgraph data.

that the computation for the three intervals under analysis took 1140, 540, and 420 seconds respectively. Hence, in all cases, the runtime is much smaller than the corresponding interval, confirming the possibility to roll out the solution.

5 CONCLUSION

In this work we kept walking the thin line between database theory and industrial practice, putting recent temporal extensions of the famous Datalog language for databases into action to provide a fully declarative implementation of a real smart derivative.

Paradigms that sustain understandability, transparency, and ease of communication are of relevance for a Central Bank and for financial authorities. Furthermore, we believe that the application in complex financial areas of knowledge representation and reasoning formalisms conceived in the academic space encourages once more to look at these languages and the uprising reasoning systems as professional tools for core industrial settings.

In our case, given the complexity of the financial instrument at hand, we believe our contribution can be easily replicated or adapted for other derivatives and represents a step towards a declarative paradigm for smart derivatives (and smart contracts in general), which are too often obscure or anyway hard to understand, communicate, analyze, and simulate. Looking further, extensions to our program could be adopted by private market players for internal risk management activities, for instance, to be able to swiftly react to the evolution of each margin account over time, or for automatically reporting up-to-date data to authorities, like the size of the position at each time point.

As a final consideration, it could be stimulating to debate whether a blockchain deployment of a DatalogMTL—or anyway logic-based / fully declarative / semantically unambiguous—program is at the forefront of interest for the community and we do believe this paper could help spur such discussion. In the positive case, further hard challenges, which we considered out of scope for this paper (e.g., which blockchains, which consensus protocols; how to distribute computation at best between logical nodes), will naturally arise and interest the database community.

REFERENCES

[1] 2022. Optimism Documentation. http://bitly.ws/xd7H. [Last accessed on 2022-11-28].

[2] 2022. Subgraph Documentation. http://bitly.ws/xd7C. [Last accessed on 2022-11-28].

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[4] Luigi Bellomarini, Giuseppe Galano, Markus Nissl, and Emanuel Sallinger. 2021. Rule-based Blockchain Knowledge Graphs: Declarative AI for Solving Industrial Blockchain Challenges. In *Proceedings of the 15th International Rule Challenge, 7th Industry Track, and 5th Doctoral Consortium @ RuleML+RR 2021 (CEUR Workshop Proceedings, Vol. 2956)*, Ahmet Soylu, Alireza Tamaddoni-Nezhad, Nikolay Nikolov, Ioan Toma, Anna Fensel, and Joost Vennekens (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-2956/paper60.pdf

[5] Luigi Bellomarini, Markus Nissl, and Emanuel Sallinger. 2021. Monotonic Aggregation for Temporal Datalog. In *Proceedings of the 15th International Rule Challenge, 7th Industry Track, and 5th Doctoral Consortium @ RuleML+RR (CEUR Workshop Proceedings, Vol. 2956)*, Ahmet Soylu, Alireza Tamaddoni-Nezhad, Nikolay Nikolov, Ioan Toma, Anna Fensel, and Joost Vennekens (Eds.). CEUR-WS.org.

[6] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog system. *Proceedings of the VLDB Endowment* 11, 9 (May 2018), 975–987. https://doi.org/10.14778/3213880.3213888

[7] Nissl Bellomarini, Blasi and Sallinger. 2022. The Temporal Vadalog System. In *RuleML+RR 2022 [To appear]*.

[8] Sebastian Brandt, Elem Güzel Kalayci, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. 2018. Querying Log Data with Metric Temporal Logic. *J. Artif. Int. Res.* 62, 1 (may 2018), 829–877. https://doi.org/10.1613/jair.1.11229

[9] R. Buenaventura and V. Ross. 2013. Transparency and financial stability. *Financial Stability Review* 17 (April 2013), 111–119. https://ideas.repec.org/a/bfr/fisrev/20111711.html

[10] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2012. A general Datalog-based framework for tractable query answering over ontologies. *J. Web Semant.* 14 (2012), 57–83.

[11] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (1989), 146–166.

[12] Giovanni Ciatto, Roberta Calegari, Stefano Mariani, Enrico Denti, and Andrea Omicini. 2018. From the Blockchain to Logic Programming and Back: Research Perspectives. In *WOA*. 69–74.

[13] Christopher D. Clack and Ciaran McGonagle. 2019. Smart Derivatives Contracts: the ISDA Master Agreement and the automation of payments and deliveries. https://doi.org/10.48550/ARXIV.1904.01461

[14] Synthetix Community. 2022. Data on Kwenta platform. http://bitly.ws/xd6U. Last accessed on 2022-11-28.

[15] Synthetix Community. 2022. ETH-PERP Solidity code. http://bitly.ws/zxBa. Last accessed on 2022-11-28.

[16] Synthetix Community. 2022. Kwenta Documentation. http://bitly.ws/xd7q. Last accessed on 2022-11-28.

[17] Synthetix Community. 2022. Kwenta Website. http://bitly.ws/xd7w. Last accessed on 2022-11-28.

[18] Synthetix Community. 2022. Synthetix System Documentation. http://bitly.ws/xd7I. Last accessed on 2022-11-28.

[19] David J Tena Cucala, Przemysław A Wałęga, Bernardo Cuenca Grau, and Egor Kostylev. 2021. Stratified Negation in Datalog with Metric Temporal Operators. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 7 (May 2021), 6488–6495. https://doi.org/10.1609/aaai.v35i7.16804

[20] Chris Dannen. 2017. *Introducing Ethereum and Solidity*. Apress. https://doi.org/10.1007/978-1-4842-2535-6

[21] Alex Ferko, Amani Moin, Esen Onur, and Michael Penick. 2021. Who Trades Bitcoin Futures and Why? *SSRN Electronic Journal* (2021). https://doi.org/10.2139/ssrn.3959984

[22] Georg Gottlob and Andreas Pieris. 2015. Beyond SPARQL under OWL 2 QL Entailment Regime: Rules to the Rescue. In *IJCAI*. AAAI Press, 2999–3007.

[23] John C. Hull. 2010. *Options, Futures, and other Derivatives* (7th ed.). Pearson.

[24] Shaul Kfir and Camille Fournier. 2019. DAML: The contract language of distributed ledgers. *Commun. ACM* 62, 9 (2019), 48–54.

[25] Ming Li, Jian Weng, Anjia Yang, Jiasi Weng, and Yue Zhang. 2020. Towards Interpreting Smart Contract against Contract Fraud: A Practical and Automatic Realization. Cryptology ePrint Archive, Paper 2020/574. https://eprint.iacr.org/2020/574 https://eprint.iacr.org/2020/574.

[26] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing Implications of Data Dependencies. *ACM Trans. Database Syst.* 4, 4 (1979), 455–469.

[27] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. 2018. An Overview of Smart Contract and Use Cases in Blockchain Technology. In *9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 1–4. https://doi.org/10.1109/ICCCNT.2018.8494045

[28] Massimo Morini. 2017. How the Business Model Must Change to Make Blockchain Work in Financial Markets: A Detailed Example on Derivatives, Two Years Later. *SSRN Electronic Journal* (2017). https://doi.org/10.2139/ssrn.3075540

[29] Massimo Morini. 2020. Financial Derivatives with Blockchain and Smart Contracts. *Blockchain Research Institute* (2020).

[30] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. The magic of duplicates and aggregates. In *VLDB*.

[31] E. Napoletano and B. Curry. 2021. What Is DeFi? Understanding Decentralized Finance. http://bitly.ws/xd7Y. Last accessed on 2022-11-28.

[32] Markus Nissl and Emanuel Sallinger. 2022. Modelling Smart Contracts with DatalogMTL. In *Proceedings of the Workshops of the EDBT/ICDT 2022 Joint Conference, Edinburgh, UK, March 29, 2022 (CEUR Workshop Proceedings, Vol. 3135)*, Maya Ramanath and Themis Palpanas (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-3135/EcoFinKG_2022_paper4.pdf

[33] Bank of International Settlement. 2022. Statistical release: OTC derivatives statistics at end-December 2021.

[34] Randy Priem. 2020. Distributed ledger technology for securities clearing and settlement: benefits, risks, and regulatory implications. *Financial Innovation* 6, 1 (Feb. 2020). https://doi.org/10.1186/s40854-019-0169-6

[35] Kaihua Qin, Liyi Zhou, Yaroslav Afonin, Ludovico Lazzaretti, and Arthur Gervais. 2021. *CeFi vs. DeFi – Comparing Centralized to Decentralized Finance*. Papers 2106.08157. arXiv.org. https://ideas.repec.org/p/arx/papers/2106.08157.html

[36] Emanuel Regnath and Sebastian Steinhorst. 2018. SmaCoNat: Smart Contracts in Natural Language. In *2018 Forum on Specification and Design Languages (FDL)*. 5–16. https://doi.org/10.1109/FDL.2018.8524068

[37] Sylvie Riederová and Kamila Růžičková. 2011. Historical development of derivatives' underlying assets. *Acta Universitatis Agriculturae et Silviculturae Mendelianae Brunensis* 59, 7 (2011), 521–526. https://doi.org/10.11118/actaun2011590705

[38] International Swaps and Derivatives Association. 2019. Legal Guidelines for Smart Derivatives contracts: the ISDA master agreement.

[39] Przemyslaw Walega, Michal Zawidzki, and Bernardo Cuenca Grau. 2022. Reasoning Techniques in DatalogMTL. In *Proceedings of Datalog 2.0*.

[40] Przemyslaw Andrzej Walega, Bernardo Cuenca Grau, Mark Kaminski, and Egor V. Kostylev. 2020. Tractable Fragments of Datalog with Metric Temporal Operators. In *IJCAI*. ijcai.org, 1919–1925.

[41] Przemysław A. Wałęga, David J. Tena Cucala, Egor V. Kostylev, and Bernardo Cuenca Grau. 2021. DatalogMTL with Negation Under Stable Models Semantics. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*. 609–618. https://doi.org/10.24963/kr.2021/58

[42] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems* 105 (April 2020), 475–491. https://doi.org/10.1016/j.future.2019.12.019