Modelling Smart Contracts with DatalogMTL

Markus Nissl[1], Emanuel Sallinger[1]

[1]*TU Wien, Vienna, Austria*

Abstract

Smart contracts are programs that are stored in distributed ledgers (e.g., blockchains) and are usually written in procedural languages to encode agreements between parties. Recent proposals focus on using logical languages for the specification and verification of such agreements. In this paper, in line with recent work, we analyse the usage of DatalogMTL for the creation of smart contracts. We discuss archetypal use cases and explore the temporal properties of DatalogMTL for formulating certain specifications.

Keywords

DatalogMTL, Smart Contract, Blockchain

1. Introduction

Distributed ledger technologies (for example, blockchains) provide the foundation and core infrastructure for decentralised finance (DeFi), a type of finance that does not rely on intermediaries like exchanges, banks or brokers [1]. DeFi applications are built by utilising smart contracts, which are executable code that facilitates the process of executing and enforcing the terms of an agreement between (untrusted) parties [2].

While agreements naturally follow a human-readable format (e.g., see Example 1.1), today's usual way of writing such smart contracts is by encoding rules in an object-oriented programming language such as Solidity for the Ethereum platform. The procedural style of these languages requires to define what and how certain conditions should be handled, which is an error-prone and cumbersome process and hard to verify by non-experts whether the intention is reflected in code.

Example 1.1. I lend you ten Bitcoins with an interest rate of 3% per year for three years. The interest payment is monthly.

In an attempt to tackle this issue, researchers suggest the use of logic-based smart contract languages, which allow to better represent and reason upon the conditions of a smart contract [2, 3]. Different suggested solutions include the use of Prolog [4, 5], domain-specific languages [6], finite state machines [7], Active-U-Datalog [8], or formal contract logic [3].

Let us consider Example 1.1 again. By carefully studying this example, one notices that it contains multiple temporal specifications (e.g., per month, per three years, or per year). While temporal reasoning has been used for the verification of smart contracts [9], current solutions for logic-based smart contracts, to the best of our knowledge, do not consider temporal properties at all or in a sub-optimal way, e.g., by arithmetic or by the introduction of predicates with a special temporal meaning. This is an enormous drawback, as many DeFi applications require some sort of temporal processing[1].

Requirements. In the following, we suggest what we think are the most essential temporal requirements a logic-based smart contract language has to support to provide a minimum amount of useful reasoning capabilities for DeFi applications.

1. *Validity Interval.* It is necessary to represent intervals, i.e., when a specific kind of operation is valid, not only punctual points. This is necessary for example in voting contracts, where votes are exactly allowed between the start and the end of an interval.
2. *Periodicity.* It is necessary to specify periodic patterns that encode repeating agreements. This is necessary for example to encode that the salary is paid per month.
3. *One Time Event/Delay.* It is necessary to encode single future events. This is necessary for example for shopping contracts, where a payment reminder has to be sent in case the money has not been paid.
4. *Negation.* While only indirectly related to the temporal domain, it is necessary to support negation. This is necessary to encode that something has not happened in a specific interval as the previous example highlights.
5. *Verification.* It is necessary to verify whether a given model is satisfied for a given set of rules. This helps for the verification of certain proper-

Published in the Workshop Proceedings of the EDBT/ICDT 2022 Joint Conference (March 29-April 1, 2022), Edinburgh, UK

[1]We discuss current solutions in the discussion of the related work (see Section 3)

ties and is in line with the work of using temporal logic for verification of smart contracts.

DatalogMTL. A typical language that supports all of these requirements is DatalogMTL. DatalogMTL extends Datalog with the Horn fragment of metric temporal logic (MTL) and allows us to formulate expressions such as $\boxminus_{[0,24h]} \mathit{Signal}(x)$ to state that the signal x occurred continuously over the last 24 hours or $\diamondminus_{[1h,2h]} \mathit{Signal}(x)$ to state that the signal x occurred at least once in the penultimate hour. It supports intervals (1) and delays (3) out of the box and provides the handling of periodicity (2) by the underlying reasoning language Datalog, where recursion is a fundamental core of the language. The support of negation (4) is provided by stratified negation (see preliminaries), but more complex forms of negation (i.e., stable models) have already been considered, if needed [10].

Contribution. In this work, we introduce DatalogMTL as a suitable language for formulating explainable logic-based smart contracts. In detail, our main contributions are:

- We established a set of *temporal requirements* for logic-based smart contract languages which we think are necessary for a variety of agreements.
- We *formally define* a smart contract language on top of DatalogMTL which considers the interaction with other smart contracts.
- We provide a *case study on real-world examples* on exploring DatalogMTL for Smart Contracts where we discuss the modelling of smart contracts as well as the verification.
- We provide throughout the work connections to the *economic sector*, targeting exactly the area of the workshop.

Financial and Economic Context. We briefly want to relate the work to the financial and economic context. As already highlighted in the beginning of the introduction, DeFi will become (or already is) an area with a high volume of transactions and a high market capitalization. Current DeFi solutions lack insight and do not provide explainable systems. We have the opinion that logic-based smart contracts, and especially DatalogMTL, could provide the required core for reasoning over DeFi applications and would be suitable as a basis for building systems for many economic and financial fields, such as:

- Regulatory Compliance
- Anti-money Laundering
- Financial stability assessment

This paper contributes in two ways to the discussion of DeFi applications regarding the economic sector. On the one hand, we identify the need of supporting several temporal properties in logic-based smart contract languages, which we summarized in the beginning as requirements, and on the other hand, we suggest the usage of DatalogMTL as a go-to solution and introduce DatalogMTL smart contracts for writing DeFI applications.

Organization. The remainder of this paper is organized as follows: In Section 2 we introduce DatalogMTL and provide a brief overview on Smart Contracts. In Section 3 we discuss related work in the area of logic-based smart contracts. We discuss the usage of DatalogMTL as a modelling language for Smart Contracts in Section 5 and conclude the work in Section 6.

2. Preliminaries

In this section, we briefly introduce smart contracts and present the syntax and semantic of DatalogMTL with stratified negation over the integer timeline.

2.1. Distributed Ledgers and Smart Contracts

A distributed ledger technology (DLT) is a decentralized, immutable, and append-only database across different nodes that is managed by multiple participants. At the core of the DLT is a consensus mechanism which contains procedures and rules on how transactions are validated by the nodes [11].

A blockchain is a form of DLT, where transactions are recorded and grouped into blocks, where each block includes a hash of the previous block. Other forms of DLTs are Tangle [12], built on top of a directed acyclic graph, or Corda [13], a leading DLT for regulated industries where the ledger stores per node only the facts the node is aware of.

Smart Contracts are programs stored on the distributed ledger technology that run when certain conditions are satisfied. Usually, smart contracts manage an agreement (i.e., rules) between multiple parties per code without requiring a third party. They reduce risk due to the tamper-proof property of DLTs and provide transparency to the process. Typical smart contract applications include voting, supply chains, mortgage, copyright protection, or employment arrangements [14].

2.2. DatalogMTL

DatalogMTL extends Datalog with metric temporal logic. In this section, we briefly recap the syntax and semantic of DatalogMTL over the integer timeline [15] with stratified negation [16].

Intervals. An interval is of the form $\langle t_1, t_2 \rangle$, where the left (right) bracket is either (or [() or]) and $t_1, t_2 \in \mathbb{Z} \cup \{-\infty, \infty\}$, such that $t_1 \leq t_2$. An interval is positive if $t_1 > 0$ and punctual if $t_1 = t_2$, which we write as t instead of $[t, t]$.

Syntax. We assume a disjoint set of variables and constants. An *atom* is an expression of the form $P(\boldsymbol{\tau})$, where P is a predicate of arity n and $\boldsymbol{\tau}$ is a tuple of constants and variables matching the arity. A *literal* M is an expression given by the following grammar:

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

where $P(\boldsymbol{\tau})$ is an atom and ϱ is a positive interval. A *rule* is an expression of the form

$$M_1 \wedge \cdots \wedge M_k \wedge \text{not } M_{k+1} \wedge \cdots \wedge \text{not } M_{k+m} \rightarrow M'$$

for $k, m \geq 0$ and where each M_i is a literal and M' is a literal restricted to the following grammar:

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

The conjunction of M_i is the *body* and M' is the *head* of the rule. 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. A rule is *safe*, if all variables occur in positive body literals of the rule, and *positive* if it contains no negative body literal. A DatalogMTL *program* is a finite set of safe rules. A program Π is *stratifiable* if there exists a stratification of a program Π. A stratification of Π is given as a function σ that maps each predicate in Π to positive integers such that for each rule it holds that $\sigma(P^+) \leq \sigma(P)$ and $\sigma(P^-) < \sigma(P)$ for P^+ a positive body literal, P^- a negated body literal and P the head of the rule. We say that an expression is *ground* if it contains no variables. A *fact* is an expression of the form $P(\boldsymbol{\tau})@\varrho$, where $P(\boldsymbol{\tau})$ is ground and ϱ a non-empty interval. A *dataset* is a finite set of facts.

Semantics. An *interpretation* $\mathfrak{M}$ specifies for each time point $t \in \mathbb{Z}$ and for each ground atom $P(\boldsymbol{a})$ whether $P(\boldsymbol{a})$ is satisfied at t, in which case we write $\mathfrak{M}, t \models P(a)$. This notion extends to ground literals as follows:

$$\begin{array}{lr}
\mathfrak{M}, t \models \top & \text{for each } t \in \mathbb{Z} \\
\mathfrak{M}, t \models \bot & \text{for no } t \in \mathbb{Z} \\
\mathfrak{M}, t \models \boxminus_\varrho A & \text{iff } \mathfrak{M}, s \models A \text{ for all } s \text{ with } t - s \in \varrho \\
\mathfrak{M}, t \models \boxplus_\varrho A & \text{iff } \mathfrak{M}, s \models A \text{ for all } s \text{ with } s - t \in \varrho \\
\mathfrak{M}, t \models A\ \mathcal{S}_\varrho\ A' & \text{iff } \mathfrak{M}, s \models A' \text{ for some } s \text{ with } t - s \in \varrho \\
 & \wedge\, \mathfrak{M}, r \models A \text{ for all } r \in (s, t) \\
\mathfrak{M}, t \models A\ \mathcal{U}_\varrho\ A' & \text{iff } \mathfrak{M}, s \models A' \text{ for some } s \text{ with } s - t \in \varrho \\
 & \wedge\, \mathfrak{M}, r \models A \text{ for all } r \in (t, s) \\
\mathfrak{M}, t \models \diamondminus_\varrho A & \text{iff } \mathfrak{M}, s \models A \text{ for some } s \text{ with } t - s \in \varrho \\
\mathfrak{M}, t \models \diamondplus_\varrho A & \text{iff } \mathfrak{M}, s \models A \text{ for some } s \text{ with } s - t \in \varrho
\end{array}$$

An interpretation $\mathfrak{M}$ satisfies for each literal M and every time point t not M, written $\mathfrak{M}, t \models \text{not } M$, if $\mathfrak{M}, t \not\models M$. An interpretation is a *model* of a ground rule whenever it satisfies all atoms of the body it also satisfies the head of the rule, a *model* of a rule, when it satisfies all groundings of the rule and a *model* of a program Π, written $\mathfrak{M} \models \Pi$, if it is a model of all rules in Π and the program has a stratification. An interpretation $\mathfrak{M}$ is a *model* of a fact $P(\boldsymbol{a})@\varrho$, written $\mathfrak{M} \models P(\boldsymbol{a})@\varrho$, if $\mathfrak{M} \models P(\boldsymbol{a})@t$ for all $t \in \mathbb{Z}$ within ϱ, and a *model* of a set of facts (e.g., a dataset) D if it is a model of all facts in D. A program Π and a dataset D *entail* a fact $P(\boldsymbol{a})@\sigma$, written $(\Pi, D) \models P(\boldsymbol{a})@\sigma$, if $P(\boldsymbol{a})@\sigma$ for each model of Π and D.

3. Related Work

Idelberger et al. [3] suggest the formulation via formal contract logic, a (deontic) defeasible logic. They suggest specifying a default state and use superiority relations to overwrite the state in case a specific event is triggered. This logic always only derives a single state based on current triggers that are valid at the current time. They have not discussed temporal properties at all.

Similarly, Frantz and Nowostawski [6] decompose institutions into rule-based statements, which are constructed from 5 different components (attributes (A), deontic (D), aim (I), conditions (C), or else (O)). An example is "people (A) must (D) vote (I) every four years (C), or else they face a fine (O)" and suggest a mapping of the components to Solidity smart contracts using a domain-specific language which creates a template that has to be completed manually.

Stancu and Dragan [5] suggested a Python-to-Prolog interface on top of BigchainDB and Tendermint to forward Prolog clauses to a Prolog process to verify the contract.

Suvorov and Ulyantsev [7] suggest to use a LTL specification and test scenario to create a finite state machine, which is combined with state and action definitions to generate a Solidity smart contract. They use LTL to express possible state transitions in a finite state machine, but again no temporal properties are explored such as a duration of a voting period.

Hu and Zhang [8] propose Logic-SC, a smart contract model based on Active-U-Datalog with temporal extensions. Active-U-Datalog extends Datalog rules with update atoms, which can add or remove relations from the dataset. The temporal extension is provided by a pair $\langle [begin, end], P \rangle$, where P is a periodic expression and $[begin, end]$ denotes the lower and upper bounds for the intervals in P. An example of a periodic expression is $all.Years + \{3, 7\}.Months \triangleright 2.Months$, representing intervals starting as third and seven month of every year and having a duration of 2 months. A rule for counting the total working hours is given as follows $totalhours(t_1), t_2 = t_1 + 1, +Hour(t), t \in workingTime \rightarrow -totalhours(t_1), +totalhours(t_2)$, where $workingTime$ is a periodic expression. Critically, this approach deletes elements which are not valid at the current time point and adds new elements, and hence simulates the behaviour of procedural implementations which also add or remove elements. In comparison, our approach works per time point and assumes a single action per time point. Hence, we explore the concept of append-only data structures and create a new status for each new time point.

Ciatto et al. [4] suggest a new language called Tenderfone, that is based on Prolog. It offers several entry points and built-in functions which manage the interaction with smart contracts. These functions include $\mathit{init}(\text{args})$, which gets executed once when the smart contract is deployed, $\mathit{receive}(\text{Msg}, \text{Sender})$ to invoke the smart contract and $\mathit{send}(\text{Msg}, \text{Recipient})$ to call some other contract. In addition, they offer several temporal predicates. This includes $\mathit{now}(\text{T})$ to get the current timestamp, $\mathit{when}(\text{T}, \text{Msg})$ to trigger a *receive* of the smart contract at time T, $\mathit{delay}(\text{DT}, \text{Msg})$ which is equal to $\mathit{when}(\text{T} + \text{DT}, \text{Msg})$, $\mathit{now}(\text{T})$ and the fact that $\mathit{periodically}(\text{P}, \text{Msg})$, which triggers *receive* every P time units. This suggestion is close to our solution, as it is the only solution that considers temporal properties. However, they only consider single time units and do not cover intervals.

To summarize, related work focused on the formulation of logic-based smart contracts as well as on the generation of efficient code. This paper will focus only on the formulation aspect and keeps the generation open for future work. The solution of Hu and Zhang provide through the flexibility of Datalog, the possibility to model certain temporal properties, and the solution of Ciatto et al. is the only work that actively considers temporal properties in their proposal. To the best of our knowledge, intervals have not been considered so far in the study of logic-based smart contract languages.

4. DatalogMTL Smart Contracts

In this section, we formally specify smart contracts with DatalogMTL. While the introduced program would allow one to specify the rules of the contract, possible interaction points with different parties and smart contracts as well as an initial state have to be specified.

Definition 4.1. A smart contract is a quadruple (N, Π, D, A) where:

- N is a unique name of the contract and can be seen as a namespace for the contract.
- Π is the DatalogMTL program encoding the rules of the smart contract
- D is the initial dataset encoding the initial state of the smart contract.
- A is the set of activators. This set contains atoms that expect input from others and act as interaction points, which other smart contracts or parties may invoke. Per time point, it is only allowed to fire one activator.

While Π and D map directly to the definition of DatalogMTL, N and A are in addition required for smart contracts. The namespace provides the possibility to *call* other smart contracts by generating an atom, which is prefixed with a namespace, or accessing data from other smart contracts. For example, consider a will that manages the transfer of money in case of death, then one can write a rule of the form

$$\mathit{PersonDied}(Y), \mathit{Token.balance}(Y, X), \mathit{Heritage}(Y, Z) \\ \rightarrow \mathit{Token.transfer}(Y, Z, X)$$

to specify that the whole balance of Y (accessed from the token) is transferred (by calling the token) to Y's heir Z, in case Y died.

The activators A provide input interfaces to the smart contract. The set contains all atoms in the rules, which can only be provided by external parties. Only one such activator is allowed to be fired per time point to ensure consistency with the rule (i.e., one time point exactly matches to one state of the contract). This is without loss of generality, as in case multiple activators are required, they can be combined by introducing an additional activator and creating auxiliary atoms for internal use. Note, in the following examples we use directly the activators in rule heads for readability, instead of instantiating some additional atom, i.e., we write rules of the form $P \rightarrow Q$, where Q is an activator, instead of introducing an auxiliary atom R and map the activator to the auxiliary atom $Q \rightarrow R$ which is used in the rules as head $P \rightarrow R$. We still ensure that only one external activator is used per state transition.

Example 4.1. A smart contract in DatalogMTL with the rule given in Example 1.1 is given as a quadruple (N, Π, D, A), where N is some unique identifier such as $\mathit{BitcoinLending}$, the dataset D contains the information regarding the monthly interest payment $\mathit{Interest}(0.0025)@[0, \infty)$, the activator A is the atom Borrow which is called when the lending starts and fixes the start time point, and the program Π is given as follows:

$$\mathit{Borrow}(A, B) \rightarrow \boxplus_{[0,3y]} \mathit{BorrowDur}(A, B) \quad (1)$$

$$\mathit{Borrow}(A, B) \rightarrow \boxplus_{[1m,1m]} \mathit{PayTime}(A, B) \quad (2)$$

$$\mathit{PayTime}(A, B) \rightarrow \boxplus_{[1m,1m]} \mathit{PayTime}(A, B) \quad (3)$$

$$\mathit{Interest}(X), \\ \mathit{PayTime}(A, B), \\ \mathit{BorrowDur}(A, B) \rightarrow \mathit{Token.transfer}(A, B, X) \quad (4)$$

where $\mathit{Borrow}(A, B)$ specifies that A borrows the money from B, $\mathit{BorrowDur}(A, B)$ defines the duration of the borrow contract between A and B, $\mathit{PayTime}(A, B)$ specifies the time points when A has

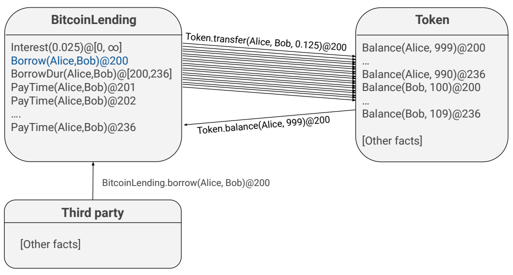

Figure 1: Calling of *borrow* by a third party in $BitcoinLending$. Accessing data (*balance*) and calling (*transfer*) of smart contract $Token$ in smart contract $BitcoinLending$.

to pay interest to B, $Interest(X)$ specifies the interest amount which has to be paid and $Token.transfer$ is a call to an external smart contract, for example a smart contract such as given in Section 5.2, where A pays B the amount X.

In detail, Rule (1) specifies the duration of the agreement with three years beginning at the time point where $Borrow$ gets called. Rules (2) and (3) define the payment time points. In detail, Rule (2) specifies the first interest payment as the first month after the time point where $Borrow(A, B)$ was activated by some party or smart contract, and Rule (3) extends this to follow-up payments recursively. Rule (4) executes the token transfer of the interest amount in case it is payment time and it is valid (i.e., during the $BorrowDur$) by executing the activator of a different smart contract.

We visualized in Figure 1 the use of activators and namespaces to access the data of another smart contract of Example 4.1, where we assume that A is Alice, B is Bob and there is some additional rule that requires access to the balance of A (for example, to check whether A is creditworthy or not) to visualize the read direction and a third party, which could either be Alice, Bob or some smart contract that start the borrowing transaction at some time point which gets added as fact to the contract (marked as blue) and derives all other facts given in the box according to the rules of the program.

5. Case Study

In the previous section, we introduced DatalogMTL smart contracts with a simple example. In this section, we study the use of DatalogMTL together with two real-world examples. We first consider a widely discussed example in the area of logic-based smart contracts, and then we discuss on how to utilize DatalogMTL for writing a smart contract for a crypto token.

5.1. License Agreement

In this case study, we focus on the specification of a license agreement[7, 3, 8]. In this section, we want to explore how such a contract can be written in DatalogMTL.

Example 5.1. Let us consider the following clauses for the right to evaluate and publish the evaluation results of a product [3]:

1. The Licensor grants the Licensee a license to evaluate the Product.
2. The Licensee must not publish the results of the evaluation of the Product without the approval of the Licensor; the approval must be obtained before the publication. If the Licensee publishes results of the evaluation of the Product without approval from the Licensor, the Licensee has 24h to remove the material.
3. The Licensee must not publish comments on the evaluation of the Product, unless the Licensee is permitted to publish the results of the evaluation.
4. If the Licensee is commissioned to perform an independent evaluation of the Product, then the Licensee has the obligation to publish the evaluation results.
5. This license will terminate automatically if the Licensee breaches this agreement.

Apart from the fact that contracts are of natural interest to the economic sector, changing the stakeholders and adapting a few sentences directly leads to an economic example, namely, the golden power check for company takeovers [17]. In this case, the licensor is the government (which has veto/approval power for acquisitions), the licensee is the “acquiring” company, and the product is the target company.

We use the atom $GrantX(A, B, P)$ to specify that Licensor A has granted Licensee B the right to do X for product P, where X is either Use to grant the right to evaluate, $Commission$ to grant the right to publish results without approval, $Appr$ to grant the right to publish the results. We further use $Publish(B, P)$ to specify that B has published the results of product P and $Comment(B, P)$ to specify that B has made a comment on the evaluation of product P, $Remove(B, P)$ to specify that B has removed the content and $Violation(B, P)$ to specify that B has violated the license for product P and $OblRemove(B, P)$ to specify that B is obliged to remove the license.

The smart contract is then defined as a quadruple (N, Π, D, A), where N is some unique identifier such as $LicenseAgreement$, the dataset D is empty, the activators are given by the set of atoms $\{Publish, Comment, GrantAppr, GrantCommission, GrantUse, Remove\}$ and the program Π is given by the following rules, where we omit the terms in the rules for readability:

$$GrantAppr \rightarrow \boxplus_{[0,\infty)} GrantAppr \qquad (1)$$

$$\begin{aligned} \mathit{Publish}, \neg \mathit{GrantAppr} &\rightarrow \mathit{OblRemove} && (2) \\ \mathit{OblRemove}, \neg \mathit{Remove}, & \\ \neg \boxminus_{[0,24]} \mathit{OblRemove} &\rightarrow \boxplus_{[1,1]} \mathit{OblRemove} && (3) \\ \boxminus_{[0,24]} \mathit{OblRemove} &\rightarrow \mathit{Violation} && (4) \\ \mathit{Comment}, \neg \mathit{GrantAppr} &\rightarrow \mathit{Violation} && (5) \\ \mathit{GrantCommission} &\rightarrow \mathit{GrantAppr} && (6) \\ \mathit{GrantUse}, \neg \mathit{Violation} &\rightarrow \boxplus_{[1,1]} \mathit{GrantUse} && (7) \end{aligned}$$

Rule (1) specifies that an approval holds forever. Rules 2-4 match clause (2). Rule 2 states that if the results are published, but they are not approved, then the licensee is obliged to remove the results. Rule (3) specifies the duration of the obligation, which is bounded to 24 hours and extended to the next time point in case no removal has happened. Note that this rule uses an extended form of stratified negation, namely, temporal stratified negation. In this form, the negated facts in recursion only propagate information from the past to the future. Rule (4) then triggers a violation in case the maximum time of 24 hours has been reached. For comments, there is no exception and hence any comment without approval is marked as a violation by Rule (5) which matches clause (3) of the agreement. Rule (6) handles independent evaluations. In case this right is given, then it is also approved. Rule (7) manages the extension of the license, in case no violation occurred.

Discussion. From the example, we can see that DatalogMTL allows to express complex contracts without the need of formulating rules with any form of explicit time-handling by only using temporal operators. By comparing the example with the established requirements from Section 1, this example gives evidence for requirement (1) by rule (1) or (4), requirement (3) by rule (4), requirement (4) by rules (2), (3), (5) and (7). We do not cover requirement (2) here, but (2) has been covered in Section 4. Requirement (5) is usually not part of the contract itself but used to verify that a contract does not reach a certain state. For example, it can be used to state that only one activator is fired at most once per time point or to state that it is not possible to publish a result without having the license to use the program, as the program cannot be evaluated. The latter case can be formulated by a rule of the following form:

$$\mathit{Publish}, \neg \diamondminus_{[0,\infty)} \mathit{GrantUse} \rightarrow \bot$$

5.2. ERC20 Token

In this case study, we analyse how an Ethereum ERC20 token contract [18] can be formalized in DatalogMTL. The standard provides some read-only functions, such as what is the current balance of a user, which we do not consider in the example, as just the atoms of the program have to be accessed. In the following, we focus on the functions, which change the state of the contract. The goal of these functions is to specify when a transfer of tokens between two parties is allowed and how a transfer affects the balance of the parties.

Example 5.2. Let us consider the write operations of the Ethereum ERC20 specification, which are in total three methods:

- *Transfer.* Party A transfers amount X to party B. Such a transfer is only allowed if the current balance of A is bigger or equal to X.
- *Approve.* Party A allows party B to withdraw amount X.
- *TransferFrom.* Party B transfers amount X from party A to party C. Such a transfer is only allowed if the current balance of A and the approval value for B from A are bigger or equal to X.

We use the atom $\mathit{Tran}(A, B, X)$ to specify that there is a request to transfer amount X from party A to party B, $\mathit{TranF}(A, C, X, B)$ to specify that there is a request from party B to transfer amount X from party A to party C, $\mathit{App}(A, B, X)$ to denote that party A allows party B to transfer amount X. In addition we specify that the requested transfer is valid by $\mathit{VTran}(A, B, X)$, and denote with $\mathit{Bal}(A, X)$ that party A's current balance is X.

The smart contract is then defined as a quadruple (N, Π, D, A), where the namespace N is some unique identifier such as Token, the dataset D contains the initial balance, which is assigned to one or more people, in this case 2000 tokens are assigned to Alice, $\mathit{Bal}(\mathit{Alice}, 2000)$, the activators are given by the set of atoms $\{\mathit{Tran}, \mathit{App}, \mathit{TranF}\}$ and the program Π is given by the following rules

$$\begin{aligned} & \mathit{Tran}(A, B, X), \mathit{Bal}(A, Y), \\ & \qquad X \leq Y \rightarrow \mathit{VTran}(A, B, X) && (1) \\ & \mathit{VTran}(A, B, X), \mathit{Bal}(A, Y) \rightarrow \boxplus_{[1,1]} \mathit{Bal}(A, Y - X) && (2) \\ & \mathit{VTran}(A, B, X), \mathit{Bal}(B, Y) \rightarrow \boxplus_{[1,1]} \mathit{Bal}(B, Y + X) && (3) \\ & \mathit{App}(A, B, Y), \\ & \mathit{TranF}(A, C, X, B), X \leq Y \rightarrow \mathit{Tran}(A, C, X) && (4) \\ & \mathit{VTran}(A, C, X), \\ & \mathit{App}(A, B, Y), \\ & \mathit{TranF}(A, C, X, B), \\ & \qquad Z = Y - X \rightarrow \boxplus_{[1,1]} \mathit{App}(A, B, Z) && (5) \\ & \mathit{Bal}(A, Y), \neg \mathit{VTran}(A, \_, \_), \\ & \qquad \neg \mathit{VTran}(\_, A, \_) \rightarrow \boxplus_{[1,1]} \mathit{Bal}(A, Y) && (6) \\ & \mathit{App}(A, B, Y), \\ & \qquad \neg \mathit{VTran}(A, \_, \_) \rightarrow \boxplus_{[1,1]} \mathit{App}(A, B, Y) && (7) \\ & \mathit{App}(A, B, Y), \\ & \mathit{VTran}(A, \_, \_), \\ & \qquad \neg \mathit{TranF}(A, \_, \_, B) \rightarrow \boxplus_{[1,1]} \mathit{App}(A, B, Y) && (8) \end{aligned}$$

Rule (1) verifies whether the balance of party A is enough for the transfer. Rules (2) and (3) apply a verified transfer by reducing and increasing the balance of the participating parties. Rule (4) applies the third-party transfer by checking whether the party is allowed to transfer the amount from party A. In such a case, a usual transfer is emitted, which is handled by rules (1) to (3). Rule (5) updates the approval amount in case such a transaction is successful. Rules (6)-(8) are the housekeeping rules, which copy the current balance and the approved value to the next unit, in case nothing changed.

Arithmetic Expressions. In this example, we have used arithmetic expressions in the rules, which have not been formally defined in DatalogMTL. We want to point out that the use of arithmetic in recursion without any restrictions yields undecidability in general [19], however in this case the arithmetic expression is bounded by activators. A formal definition of arithmetic in DatalogMTL has not been considered yet and is future work. The arithmetic operations of this example follow the default notion of addition and subtraction.

6. Conclusion

In this paper, we introduced DatalogMTL as an option to model smart contracts. We first derived the essential requirements for the necessary features in smart contracts, and then introduced DatalogMTL as a basis for a smart contract language. We explored our language by discussing two real-world examples. In the first one, we introduced a widely discussed existing example and highlighted the advantages of using DatalogMTL, while in the second one we discussed how to model tokens with DatalogMTL. Thereby, we identified the need to have a formally proved arithmetic extension of DatalogMTL to support DeFi applications which require arithmetic operations, which we will consider studying in the future. In addition, we want to focus on compiling DatalogMTL smart contracts to existing procedural smart contract languages such as Solidity to support well-established blockchains such as Ethereum.

Acknowledgments. This work was supported by the Vienna Science and Technology Fund (WWTF) grant VRG18-013, and the "rAIson data" Royal Society grant of Prof. Georg Gottlob.

References

[1] A. Singh, What is decentralized finance or defi explained, 2021. URL: https://medium.com/brandlitic/dc0ce376f7b2.

[2] M. Alharby, A. van Moorsel, Blockchain-based smart contracts: A systematic mapping study, CoRR abs/1710.06372 (2017).

[3] F. Idelberger, G. Governatori, R. Riveret, G. Sartor, Evaluation of logic-based smart contracts for blockchain systems, in: RuleML, volume 9718 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 167–183.

[4] G. Ciatto, A. Maffi, S. Mariani, A. Omicini, Smart contracts are more than objects: Pro-activeness on the blockchain, in: BLOCKCHAIN, volume 1010 of *Advances in Intelligent Systems and Computing*, Springer, 2019, pp. 45–53.

[5] A. Stancu, M. Dragan, Logic-based smart contracts, in: WorldCIST (1), volume 1159 of *Advances in Intelligent Systems and Computing*, Springer, 2020, pp. 387–394.

[6] C. Frantz, M. Nowostawski, From institutions to code: Towards automated generation of smart contracts, in: FAS*W@SASO/ICCAC, IEEE, 2016, pp. 210–215.

[7] D. Suvorov, V. Ulyantsev, Smart contract design meets state machine synthesis: Case studies, CoRR abs/1906.02906 (2019).

[8] J. Hu, Y. Zhong, A method of logic-based smart contracts for blockchain system, in: ICDPA, ACM, 2018, pp. 58–61.

[9] P. Tolmach, Y. Li, S. Lin, Y. Liu, Z. Li, A survey of smart contract formal specification and verification, ACM Comput. Surv. 54 (2022) 148:1–148:38.

[10] P. A. Walega, D. J. T. Cucala, E. V. Kostylev, B. C. Grau, Datalogmtl with negation under stable models semantics, in: KR, 2021, pp. 609–618.

[11] 101 Blockchains, What is dlt (distributed ledger technology) ?, 2021. URL: https://101blockchains.com/what-is-dlt/.

[12] S. Popov, The tangle, White paper 1 (2018).

[13] M. Hearn, R. G. Brown, Corda: A distributed ledger (2019).

[14] H. Anwar, Smart contracts: The ultimate guide for the beginners, 2018. URL: https://101blockchains.com/smart-contracts/.

[15] P. A. Walega, B. C. Grau, M. Kaminski, E. V. Kostylev, Datalogmtl over the integer timeline, in: KR, 2020, pp. 768–777.

[16] D. J. T. Cucala, P. A. Walega, B. C. Grau, E. V. Kostylev, Stratified negation in datalog with metric temporal operators, in: AAAI, AAAI Press, 2021, pp. 6488–6495.

[17] L. Bellomarini, M. Benedetti, S. Ceri, A. Gentili, R. Laurendi, D. Magnanimi, M. Nissl, E. Sallinger, Reasoning on company takeovers during the COVID-19 crisis with knowledge graphs, in: RuleML+RR (Supplement), volume 2644 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020, pp. 145–156.

[18] V. B. Fabian Vogelsteller, Eip-20, 2015. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[19] 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.