Towards Bridging Traditional and Smart Contracts with Datalog-based Languages

Markus Nissl[1], Emanuel Sallinger[1,2]

[1] *TU Wien, Vienna, Austria*
[2] *University of Oxford, Oxford, UK*

Abstract
The distinction of traditional and smart contracts has been subject for long discussions over the last years. While there is an agreement that a one-to-one mapping of the whole contract is not the desired goal, there is an understanding that traditional and smart contracts have certain properties in common. In this paper, we present our technical insights we derived with our industrial partners by discussing the importance of a rule-based language for bridging the gap between traditional and smart contracts and introduce a translation tool for logic-based Smart Contracts for supporting well-known blockchain platforms.

Keywords
Contracts, Blockchain, Datalog

1. Introduction

Distributed ledger technology (e.g., blockchains) represents a vibrant area of new technology spreading into different research domains. In particular, smart contracts, i.e. executable code on the blockchain that facilitates the process of executing and enforcing the terms of an agreement between (untrusted) parties, create a rich field of applications in areas such as supply chain management [1], decentralised finance [2] or legally binding agreements [3].

Example 1. *Real Estate. Buying a property has many connected agreements such as warranty, payment agreements or credit agreements. A legally binding agreement usually describes the obligations of the seller, the cash flows to the seller depending on the construction process, the warranty rights of the buyer, and many more agreements. One easily sees that smart contracts are capable of streamlining the process of buying real estate, not only simplifying the process but also saving money as no escrow agent or other parties are required.*

Usually, when lawyers draft legal agreements, such as Example 1, they have to consider many different aspects of the agreement such as the currently valid law and the implications on the requirements of the client. They may even have an iterative process of improving the legal text of the contract with the client to optimally fit the clients' goals. Using the same process for the creation of smart contracts causes several challenges, which we list next [4, 5, 3]:

Datalog 2.0 2022: 4th International Workshop on the Resurgence of Datalog in Academia and Industry, September 05, 2022, Genova - Nervi, Italy
mnissl@dbai.tuwien.ac.at (M. Nissl); sallinger@dbai.tuwien.ac.at (E. Sallinger)

- Lawyers are usually not programmers and hence it is unlikely that the have the technical knowledge to build smart contracts on their own. On the other hand, programmers often do not know the exact legal considerations hidden in the contract to build a reliable smart contract. Even if a lawyer has some coding background, there are still clients who are not experts in coding, making an iterative process with the client impossible. Even the creation of standard templates from domain experts of both fields for certain kinds of agreements still do not solve the problem when an extension is required.
- Smart contracts are usually written in object-oriented programming languages such as Solidity [6] for the Ethereum blockchain. The procedural style of these languages makes it hard for non-experts to verify whether the legal considerations are reflected in code.
- A legal contract contains many clauses which have no direct executable action. For example, the place of jurisdiction will not trigger any event on the blockchain, while a required payment will initiate a transfer of the tokens from the buyer to the seller. Depending on the use case of the contract one may want to include or exclude such information.

While some of the challenges are strategic decisions, some of the issues can be tackled by using logic-based smart contract languages. It is our experience, that logic-based rules improve the readability of the code for non-experts and improve the communication between different stakeholders [7]. Different suggested solutions for logic-based languages include the use of domain-specific languages [8], finite state machines [9], Prolog [5, 10], Active-U-Datalog [11], or formal contract logic [4]. We also recently introduced in the setting of financial activities the concept of modelling smart contracts with DatalogMTL [12] to emphasize the need of expressing temporal operations in such settings.

While these concepts give the possibility to provide the toolset, their usage is currently limited: Not only is the research regarding logic-based languages in its infancy, there is usually either no distributed ledger technology natively supporting the suggested language, or the blockchain is not established and supported by a wide range of people, which can easily cause security issues (e.g., creating forks and rewriting the immutable history).

Contributions. In summary, while logic-based smart contracts would have all the advantages discussed so far, it is all but impossible for such a language to gain acceptance until it is supported by a major, widely adopted current blockchain. Achieving this is precisely the goal of this paper. Our main contributions are:

- A **theoretical discussion** of each component of our DatalogMTL language including required characteristics of our language due to typical procedural smart contract settings.
- A novel **translation system** converting DatalogMTL smart contracts into executable Solidity smart contracts, the predominant platform in practice.
- An **experimental evaluation** including a discussion of possible improvement potentials.

Organization. The remainder of the paper is structured as follows: In Section 2 we summarize DatalogMTL. In Section 3 we introduce DatalogMTL as smart contract language building on the concepts introduced in our previous work. In Section 4 we present our translation engine, which we evaluate in Section 5. We discuss related work in Section 6 and conclude this work in Section 7.

2. DatalogMTL

DatalogMTL [13, 14, 15] extends Datalog with the metric temporal logic (MTL) in two ways: (i) it assigns to each fact a set of valid timepoints (i.e., a validity interval) and (ii) it contains temporal operators that modify the validity of the interval. We expect knowledge regarding MTL, but note that readers familiar with LTL[16] will find the familiar operators used in a different context.

Intervals. Formally, intervals are defined over an ordered set of rational numbers $\mathbb{Q}$ where each interval ϱ is of the form $\langle \varrho^-, \varrho^+ \rangle$ with the endpoints $\varrho^-, \varrho^+ \in \mathbb{Q} \cup \{-\infty, \infty\}$, for each $t \in \varrho$ we have $\varrho^- \leq t \leq \varrho^+$ and $\langle$ is either open ('(') or closed ('[') and $\rangle$ is either open (')') or closed (']'). An interval is *punctual* if it is of the form $[t, t]$, in which case we write t instead, *positive* if $\varrho^- \geq 0$, and *bounded* if $\varrho^-, \varrho^+ \in \mathbb{Q}$.

Syntax. DatalogMTL with stratified negation is defined over a function-free first-order vocabulary consisting of disjoint sets of constants, variables and predicates. Each predicate is equipped with a non-negative arity. A *term* is either a constant or a variable. An *atom* is of the form $P(\boldsymbol{\tau})$, where P is a predicate and $\boldsymbol{\tau}$ is a tuple of n terms, with n matching the arity of P. A *literal* is an expression of the form

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

where ϱ is a positive interval. A *rule* is an expression of form

$$A_1 \wedge \cdots \wedge A_i \wedge \text{not } A_{i+1} \wedge \cdots \wedge \text{not } A_{i+j} \rightarrow B$$

where $i, j \geq 0$, each A_k is a literal and B is a literal not mentioning $\diamondminus$, $\diamondplus$, $\mathcal{S}$ or $\mathcal{U}$. The conjunction is the body of the rule, where the literals $A_1 \wedge \cdots \wedge A_i$ denote positive body atoms and $A_{i+1} \wedge \cdots \wedge A_{i+j}$ denote negated body atoms, and the literal B is the rule head. Instead of $\wedge$ we sometimes use "," to denote conjunction. A rule is *safe* if each variable occurs in at least one positive body atom. A *program* Π is a finite set of safe rules and *stratifiable* if there exists a stratification of a program Π. A stratification of Π is a function σ mapping each predicate P in Π to positive integers s.t. for each rule $\ldots P'(\boldsymbol{\tau}') \ldots \rightarrow P(\boldsymbol{\tau})$ in Π we have $\sigma(P') \leq \sigma(P)$ and for each rule $\ldots \text{not } P'(\boldsymbol{\tau}') \ldots \rightarrow P(\boldsymbol{\tau})$ in Π we have $\sigma(P') < \sigma(P)$. A program, rule, atom is *ground* if it contains no variables. A fact is an expression over an interval ϱ of the form $P(\boldsymbol{\tau})@\varrho$ where $P(\boldsymbol{\tau})$ is a ground atom. A dataset D is a finite set of facts.

Semantics. An interpretation $\mathfrak{M}$ is based on a domain for the individual variables and constants that specifies for each time point $t \in \mathbb{Q}$ and each ground atom $P(\boldsymbol{\tau})$, whether $P(\boldsymbol{\tau})$ is satisfied at t, written as $\mathfrak{M}, t \models P(\boldsymbol{\tau})$. The notion of satisfiability of an interpretation $\mathfrak{M}$ is extended inductively to ground literals according to Figure 1. An interpretation $\mathfrak{M}$ is a model of $\text{not } A$ ($\mathfrak{M}, t \models \text{not } A$), if $\mathfrak{M}, t \not\models A$, of a ground rule r, if $\mathfrak{M}, t \models A_k$ for $0 \leq k \leq i$ and $\mathfrak{M}, t \models \text{not } A_k$ for $i + 1 \leq k \leq i + j$ for every t, of a rule when it satisfies every possible grounding of the rule and of a a program, if it satisfies every rule in the program and the program has a stratification. An interpretation $\mathfrak{M}$ is a model of a fact $P(\boldsymbol{\tau})@\varrho$, if $\mathfrak{M}, t \models P(\boldsymbol{\tau})$ for all $t \in \varrho$ and a model of a set of facts D if it satisfies each of these facts. A program Π and a dataset D entail a fact $P(\boldsymbol{\tau})@\varrho$ ($(\Pi, D) \models P(\boldsymbol{\tau})@\varrho$) if $\mathfrak{M} \models P(\boldsymbol{\tau})@\varrho$ for each model of both Π and D.

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

Figure 1: Semantics of ground literals

Negation and Arithmetics. As smart contracts vitally require negation and arithmetics, we allow mild variants of both of them. First, we allow sequential stratified negation [17]. This allows the usage of negated body atoms in recursive rules in case the negated body atoms maintain the monotonicity property. Analogously to stratified negation [17], we allow sequential stratified arithmetic (in particular $+$,$-$,$\times$,$\div$). We note that this is a much milder form of arithmetic as e.g. in Vadalog [18], where arithmetic is also allowed in full recursion. With the help of these two extensions we are able to express many of today's use cases of smart contracts (e.g., the transfer of tokens require arithmetic, toggling a value requires the negation of the previous state).

Example 2. *Consider following DatalogMTL rule modelling a agreement – signed at some point in the past – of a payment after the successful fulfillment of a specific* $\mathit{Achievement}$.

$$\diamondminus_{[0,\infty)}\mathrm{SignedPaymentAgreement}(\mathit{Buyer}, \mathit{Seller}, \mathit{Amount}, \mathit{Achievement}),$$
$$\mathrm{SuccessfulFulfillment}(\mathit{Achievement}) \rightarrow \mathrm{Payment}(\mathit{Buyer}, \mathit{Seller}, \mathit{Amount})$$

3. Modelling Smart Contracts with DatalogMTL

In this section we discuss the modelling of smart contracts with DatalogMTL over the integer fragment [14]. We have introduced the modelling of smart contracts at EcoFinKG [12], a financial workshop, and present in this paper an extended form where we consider special symbols/variables used in blockchains (we focus on Ethereum-based blockchains) and add a discussion and detailed explanation on the derivation of each component. We introduce our running example:

Example 3. *Smart contracts often follow best-practice patterns, such as in Solidity [19], the leading smart contract language. We consider one of the main patterns here, namely that contracts often are modelled as state machines. Let us consider a state machine with the following four stages: Init, AcceptingBids, Execution, and Finished. A contract is being initialized, stays in the "accepting bids" state for exactly 10 days, then is in execution and finally finished.*

3.1. Smart Contract Initialization

In procedural smart contract languages one usually sets up the initial values with the constructor which is invoked when deploying the smart contract in the blockchain. In DatalogMTL this corresponds to providing the initial values as a given dataset at a certain timestamp. For the timestamps, we have two options:

- *Local timestamp.* The smart contract does not depend on any other timestamps (either time-information such as the current date or information from other contracts). Then one can initialize the timestamp t with 0 or any other number.
- *Global timestamp.* The smart contract uses the timestamp of the blockchain for initialization. This allows to use a shared timestamp and the actual current time inside the contract. As for the contract itself, the initial value has no influence on the reasoning (it is just shifted to a different timestamp), we always use a globally shared timestamp. This is, we set the $t = now()$, where now is the current timestamp in the blockchain.

Example 4. *(continued) The initial values are given by the following dataset encoding the initial stage of the state machine:* $\{\mathrm{Stage}(\mathit{Init})@t\}$*, where* t *is the deployment timestamp as discussed above.*

3.2. Invoking a Smart Contract

Usually, to invoke a smart contract in Ethereum one calls a method of a smart contract. Each call is initialized by a transaction, which allows the called smart contract to invoke other smart contracts. In DatalogMTL we only have rules that fire (i.e., derive the rule head) when the body is satisfied. In order to model an invocation of a method in DatalogMTL, we mark predicates as *activators*. These activators can be "called" by adding the activator for the current timestamp to the dataset. By writing rules that use these activators in the body one hence triggers the derivation of further facts. As we use activators for triggering rules, they have to obey the following restrictions to enforce compatibility with Ethereum: (i) an activator is only allowed in the body of the rule, (ii) only one activator is allowed to be executed per timestamp (to have an implicit method call order and prevent conflicting states).

Example 5. *(continued) The set of activators for the bidding state machine consists of the following activators:* $\{\mathrm{Bid}, \mathrm{Execute}\}$.

3.3. State Transitions

One usually encodes in a method of a smart contract the transition of data/state before and after the execution of a method. We already covered in the previous section the activation via an "activator". This activator triggers a set of rules that state how the current dataset at timepoint t is changed for the next state at timepoint $t + 1$. This implies that such rules are not allowed to derive facts for entries in the past.

Example 6. *(continued) The rules for the state machine are given as follows:*

$$\text{Stage}(\mathit{Init}) \rightarrow \boxplus_{(0d,10d)}\text{Stage}(\mathit{AcceptingBids}) \tag{1}$$

$$\text{Stage}(\mathit{Init}) \rightarrow \boxplus_{[10d,10d]}\text{Stage}(\mathit{Execution}) \tag{2}$$

$$\text{Stage}(\mathit{Execution}), \neg\text{Execute}, \text{Bid}(\_,\_) \rightarrow \boxplus_{[1,1]}\text{Stage}(\mathit{Execution}) \tag{3}$$

$$\text{Stage}(\mathit{Execution}), \text{Execute} \rightarrow \boxplus_{[1,1]}\text{Stage}(\mathit{Finished}) \tag{4}$$

$$\text{Stage}(\mathit{Execution}), \text{Execute} \rightarrow \text{DoAction}\ldots \tag{5}$$

$$\text{Stage}(\mathit{AcceptingBids}), \text{Bid}(S, A) \rightarrow \text{VBid}(S, A) \tag{6}$$

$$\text{VBid}(S, A), \text{HighBid}(\_, CA), A > CA \rightarrow \boxplus_{[1,1]}\text{HighBid}(S, A) \tag{7}$$

$$\text{VBid}(\_, A), \text{HighBid}(S, CA), A <= CA \rightarrow \boxplus_{[1,1]}\text{HighBid}(S, CA) \tag{8}$$

$$\text{HighBid}(S, CA), \neg\text{VBid}(\_,\_), \text{Bid}(\_,\_) \rightarrow \boxplus_{[1,1]}\text{HighBid}(S, CA) \tag{9}$$

$$\text{HighBid}(S, CA), \neg\text{VBid}(\_,\_), \text{Execute} \rightarrow \boxplus_{[1,1]}\text{HighBid}(S, CA) \tag{10}$$

Rule 1 defines the duration of the $\mathit{AcceptingBids}$ stage, Rule 2 marks the start of the $\mathit{Execution}$ and Rule 3 extends this stage in case there was no Execute*-action. Rule 4 translates the stage to $\mathit{Finished}$ in case of an execution and Rule 5 simulates a possible action of the execution. Rules 6-10 manage the bidding phase, which defines on how the data is changed in the possible settings (there is a bid which is higher/not higher and there is no bid). Note, that we model also the "unchanged" values (3, 9-10) in DatalogMTL to derive a well-defined next state (requiring always some given activator for a given timepoint for executing the rules only for the eligible time).*

3.4. Interaction between Contracts

So far, we focused on modelling a single smart contract. However, a method of a smart contract can invoke a different (target) smart contract and use the return value in the remaining method. In order to handle this case in DatalogMTL, we require the following extensions:

- *Namespaces* to uniquely identifies a given smart contract.
- *Triggers* to "trigger" an activator of a different smart contract. Note that due to the restrictions of one activator per timestamp, this creates a temporal chain of calls.
- A *Default Activator* (Caller.default) to handle the passing of "return values" to the calling smart contract. In case different default activators are required, one can introduce additional intermediary smart contracts, which default activator triggers a named activator of the original contract.

Example 7. *(continued) A namespace for the running example could be "BidingContract" and the* $\mathrm{DoAction}$ *could be an invocation of a contract (e.g., namespace* Token*) to transfer tokens, e.g.,* $\mathrm{Token.transferFrom}(\mathit{Src}, \mathit{Tgt}, 20)$. *After a successful application it uses the default activator to trigger the calling smart contracts so that it can continue the execution.*

3.5. The Contract Definition

Combining the introduced concepts, we derive the following definition:

Definition 1. *A smart contract is a quadruple* (N, Π, D, A) *where:*

- N *is a unique name of the contract (namespace).*
- 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 containing predicates which may be invoked by other smart contracts or parties.*

3.6. Blockchain-specific Parameters

While the introduced concept so far allows one to model several use cases, there is one open point for discussion, namely blockchain-specific parameters. In this paper, we study the most important properties of Solidity and how we can ensure the usage in DatalogMTL. Typically, one distinguishes between three different classes of properties:

- Block-specific properties, such as the block number, or the timestamp.
- Transaction-specific properties, such as the initiator of the transaction (origin) considering the full chain of executions starting from an external call to the activator.
- Message-specific properties, such as the caller or the number of coins sent to the invoked contract.

We inject these properties by introducing pre-defined predicates which are added to the dataset, e.g., $\mathrm{Block}(\mathit{Number})$, $\mathrm{Transaction}(\mathit{Origin})$ or $\mathrm{Msg}(\mathit{Caller})$. The only property we do not consider in this mapping is the timestamp as this causes ambiguity and hence has to be treated in a special way, which the following two points highlight:

- *Timestamp per block.* In Ethereum, the timestamp is provided per block and the only requirement for the timestamp is that it is higher than the previous block. This means, that the timestamp received in a method is the timestamp of the block. That is, each transaction, and hence each method invocation (i.e., message) shares the same timestamp and the actual timestamp is dependent on the miner of the block and may not model the exact time.
- *Time and ordering.* For modelling smart contracts we require two different kinds of timestamps. One that models the real time, for example to check if something happened within the last 24 hours, and one to model the sequential order of method invocations.

In order to handle this representation problem in DatalogMTL, we decided to assign each message its own timestamp. This is possible as the main goal is the conversion to the Ethereum platform, where the natural mapping discussed above can be used.

What we have seen is that for the purposes of this paper, namely translation, this yields a quite natural, relatively simple structure of time. We remark that for future work that includes verification of smart contracts, there is an interesting subtlety between the "logical" clock of messages, and the real-world clock (as the number of messages per block vary and are unknown in advance). While this is far beyond the scope of this paper, we want to note it is an interesting item for possible research.

4. Translating DatalogMTL to Solidity

In the previous section we established the required formalism to model a Ethereum-compatible smart contract in DatalogMTL. In this section, we present our proof of concept for translating DatalogMTL rules into Solidity. Figure 2 shows a general overview of the process, where each step is discussed in the following.

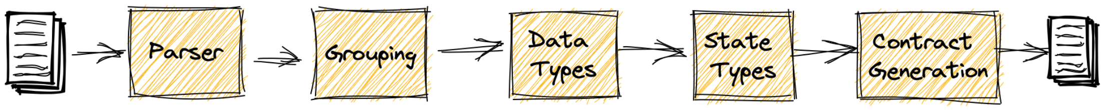


Figure 2: Overview of Translation Process

Phase 1 - Parser. The first phase is all about parsing and normalizing the program. For this, we read a DatalogMTL smart contract from file and apply a normalization procedure to ensure that each rule has a head without a temporal operator and the body either has literals containing no temporal operator, or consists of exactly one literal with a single temporal operator.

Example 8. *(continued) Rule (3) of the running example would be rewritten by introducing an auxiliary predicate and moving the temporal operation into the body as follows:*

$$\mathrm{Stage}(\mathit{Execution}), \neg\mathrm{Execute} \rightarrow \mathrm{Temp1}(\mathit{Execution})$$
$$\diamondminus_{[1,1]}\mathrm{Temp1}(\mathit{Execution}) \rightarrow \mathrm{Stage}(\mathit{Execution})$$

Phase 2 - Grouping of Rules and Method Detection. The goal of this step is to group rules executed together and sort them by execution order to create a method in a Solidity contract. By the usage of the activators we directly derive the starting points of the methods. These points can be used to trace the derivation to check whether additional rules are activated. Next to that, we find some other common patterns in rules:

- *Negated activators* are often used for describing the continuation of the current state which does not require any handling in Solidity. If this is not the case, such negated activator has to be checked before each method call.

- *Global rules* may not be triggered by any activator, i.e., they can depend on the current state of the smart contract. Similar to the second case of negated activators, these rules have to be enforced before any method call. Such rules are detected by checking the body of the rule whether it is not only directly enforced by a rule containing an activator, e.g., there is a initial state or some time delay in between.
- *Initial rules.* These are similar to global rules, but are only executed on deployment of the smart contract. These rules typically live in the constructor and have no time-dependent condition in the future (as otherwise they would be global rules).

Example 9. *(continued) The group of rules for the running example are the execution group (4-5) containing the* $\mathrm{Execute}$ *activator, the bidding rules (6-8) containing the* Bid *identifier and the global group (1-2) which are activator independent. Rule (3) and (9) are detected as negated activators copying the current state and are removed as no further consideration in Solidity is required*[a]*.*

[a]Note that we have added those rules at the beginning to allow the smart contract to be interpreted by a Datalog reasoner. When the only goal is the conversion to Solidity, one can ignore the rules when writing the smart contract in DatalogMTL.

Phase 3 - Data Types. We distinguish between the following two types:

- *Term Type.* We derive the data type of a term where possible from the internal values (e.g., from the dataset). However, when there is an activator, we are unaware of which values are provided by the user (e.g., is it 8bit or 256bit integer). Therefore, we extend the activator syntax to provide data types for the activators.
- *Atom Type.* An atom can be either stored by one or more variables, each either being a primitive, array or map. First, we distinguish whether there exists only one valid atom per time unit, then single values are sufficient and per term a variable is created. Otherwise, we check if there is a pattern for accessing the atom to identify a map type (e.g., there is an access in body and head and some variables are shared).

Example 10. *(continued) The activator for bidding would be* $\mathrm{Bid}(Int)$*. Note that with the introduction of the message-specific parameters the sender* S *is extracted into an own atom* $\mathrm{Msg}(Caller)$*. The atom type of* $\mathrm{HighBid}$ *would consists of multiple primitives.*

Phase 4 - State Types. For the state of the smart contract we have to distinguish between *local state* where the derived fact is not shared with any other method and does not need historical information, state variables where we only require the *latest state* (are not time-dependent), variables where we need a *last-timestamp of action*, and variables were we need a *history of changes* for a time period. For detecting the local state, we use the grouping to check whether there exists not another group that contains the same atom and uses information from other time units. In such a case, we can inline the state at contract generation. For detecting the latest state atoms, we check whether the state always depends on the previous timestamp in

DatalogMTL. To distinguish between the remaining cases, we differentiate on whether we have to check if some condition is currently fulfilled or whether there is a more complex comparison where the other case is not enough[1].

Example 11. *(continued) Variables* $\mathrm{Stage0}$, $\mathrm{HighestBid0}$ *and* $\mathrm{HighestBid1}$ *are identified as global state, where the numbers reference to the term position. In addition a variable storing the timestamp when* $\mathrm{Stage}(\mathit{InitState})$ *become true is created.*

Phase 5 - Contract Generation. What now remains is the processing of the rules per group. We start by adding preliminary checks, then continue per activator. This is, we structure the rules by a dependency graph and convert each rule to Solidity. In case there is an option to handle multiple rules at the same time, we try to find specific structures that follow a specific pattern. For example, for the bidding rules, we detect that these rules are a typical if-else structure and furthermore that only the if part is relevant.

Example 12. *(continued) The generated function for bidding is as follows, with checks for the correct phase and a valid offer. Note that __global is a modifier executed before the function that handles automatic state transitions.*

```
function bid(int a_) public __global {
  require(compareStrings(_stage0, "AcceptingBids"),
          "invalid request");
  require(a_ > _highestbid1, "invalid request");
  _highestbid0 = msg.sender;
  _highestbid1 = a_;
}
```

5. Evaluation

In this section, we discuss the results of our smart contract language and generation process by encoding the ERC-20 token in our language and comparing the generated code with the original Solidity code [20]. The results of our generation are given in Figure 3. We provide the corresponding DatalogMTL program in the Appendix.

In general, the generation of the smart contract shows similar code as the compared original code. This means, that the supposed steps of the pipeline achieve the desired goals. In a detail comparison, we detected some possible improvement potentials: (i) reduce code, (ii) check for variable range (e.g., overflows), (iii) improve variable types to reduce execution costs.

Besides the discussed limitations, we want to remark that the current implementation is a proof-of-concept not supporting all cases yet. The most relevant part is to detect the required state types and their dependencies given by the rules, which we identified as one point that needs more detailed investigation. Still, as the results have shown, the generation process is feasible and we continue sharpening the generator to cover more advanced tasks in the future.

[1]Note that by the definition of state transitions, non-local state is only allowed to be set for future time points.

```
contract ERC20 {
    mapping(address => int) _balanceof1;
    mapping(address => mapping(address => int)) _allowances2;

    constructor() {
        _balanceof1[msg.sender] = 2000;
    }

    function transfer(address to_, int value_) public {
        require(value_ <= _balanceof1[msg.sender], "invalid request");
        int newbalance1_ = _balanceof1[msg.sender] - value_;
        int newbalance2_ = _balanceof1[msg.sender] + value_;
        _balanceof1[msg.sender] = newbalance1_;
        _balanceof1[to_] = newbalance2_;
    }

    function approve(address spender_, int value_) public {
        _allowances2[msg.sender][spender_] = value_;
    }

    function transferFrom(address from_, address to_, int value_) public {
        require(_allowances2[from_][msg.sender] >= value_, "invalid request");
        require(value_ <= _balanceof1[from_], "invalid request");
        int newbalance1_ = _balanceof1[from_] - value_;
        int newbalance2_ = _balanceof1[from_] + value_;
        _balanceof1[from_] = newbalance1_;
        _balanceof1[to_] = newbalance2_;
        int newvalue3_ = _allowances2[from_][msg.sender] - value_;
        _allowances2[from_][msg.sender] = newvalue3_;
    }
}
```

Figure 3: Generated ERC20 Solidity Contract

6. Related Work

One can group the related work around logic-based smart contracts into two categories: work including and work not including temporal properties.

Non-temporal-based related work. Idelberger et al. [4] suggest the formulation via formal contract logic, a (deontic) defeasible logic. Frantz and Nowostawski [8] decompose institutions and suggest a mapping of the components to Solidity smart contracts using a domain-specific language. Stancu and Dragan [10] suggested a Python-to-Prolog interface to forward clauses to a Prolog process for verification of the contract. Suvorov and Ulyantsev [9] suggest to use LTL to express possible state transitions in a finite state machine which can be used to generate a Solidity smart contract.

Temporal-based related work. Hu and Zhang [11] propose a smart contract model based on Active-U-Datalog (an extension of Datalog that can remove or add atoms from the dataset) with a temporal extension given by a periodic expression and lower and upper bound where this expression is valid. Critically, this approach deletes elements which are not valid at the current time point disallowing to check the existence of previous values. In comparison, we use an append-only data structures and model the change by going forward in time. Ciatto et al. [5] suggest a new language based on Prolog offering several temporal predicates to get the current timestamp, delay the execution for some time units or execute a method periodically. This suggestion is the closest to our solution, as it is the only solution considering temporal properties, however neglecting intervals and requiring an own blockchain. Our paper builds on our work presented at EcofinKG [12] providing new key insights in the smart contract design and a novel translation technique to Solidity.

7. Conclusion

This work, to the best of our knowledge, is the first one to have a temporal logic-based smart contract language that is executable on a widely used blockchain. While it is still a proof of concept, it already handles many of the subtleties and challenges, in particular blockchain-specific variables, state types, etc. It is able to produce meaningful code for widely used patterns, including the ERC-20 token. In future work, we want to concentrate on improving the translation engine and studying on the framework around the creation of a traditional contract by strengthening the interdisciplinary collaboration.

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] S. Wang, D. Li, Y. Zhang, J. Chen, Smart contract-based product traceability system in the supply chain scenario, IEEE Access 7 (2019) 115122–115133.

[2] A. Singh, What is decentralized finance or defi explained, 2021. URL: https://medium.com/brandlitic/dc0ce376f7b2, accessed: 2021-12-20.

[3] C. Staff, The use of smart contracts for legally binding agreements, 2021. URL: https://www.gemini.com/cryptopedia/smart-contracts-legally-binding, accessed: 2022-06-09.

[4] 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*, 2016, pp. 167–183.

[5] 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*, 2019, pp. 45–53.

[6] S. Team, Solidity, 2022. URL: https://soliditylang.org, accessed: 2022-06-28.

[7] L. Bellomarini, G. Galano, M. Nissl, E. Sallinger, Rule-based blockchain knowledge graphs: Declarative AI for solving industrial blockchain challenges, in: RuleML+RR (Supplement), volume 2956 of *CEUR Workshop Proceedings*, 2021.

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

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

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

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

[12] M. Nissl, E. Sallinger, Modelling Smart Contracts with DatalogMTL, in: EDBT/ICDT Workshops, volume 3135 of *CEUR Workshop Proceedings*, 2022.

[13] S. Brandt, E. G. Kalayci, V. Ryzhikov, G. Xiao, M. Zakharyaschev, Querying log data with metric temporal logic, J. Artif. Intell. Res. 62 (2018) 829–877.

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

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

[16] A. Pnueli, The temporal logic of programs, in: FOCS, IEEE Computer Society, 1977, pp. 46–57.

[17] C. Zaniolo, Logical foundations of continuous query languages for data streams, in: Datalog, volume 7494 of *Lecture Notes in Computer Science*, 2012, pp. 177–189.

[18] G. Berger, G. Gottlob, A. Pieris, E. Sallinger, The space-efficient core of vadalog, in: PODS, ACM, 2019, pp. 270–284.
[19] Ethereum, Common patterns, 2022. URL: https://docs.soliditylang.org/en/v0.8.15/common-patterns.html, accessed: 2022-06-28.
[20] Ethereum, Erc20 contract, 2018. URL: https://github.com/ethereum/ethereum-org/blob/master/solidity/token-erc20.sol, accessed: 2022-06-05.

A. Programs

In this section, we provide the full programs of our examples in DatalogMTL as given to our translation engine, having some subtle differences to the rules presented in the work, i.e., we list the head first and use ":-" instead of $\rightarrow$.

A.1. ERC20 token

In addition to the generated program in Figure 3 and the linked source code of the Solidity version, we provide here the full version encoded in DatalogMTL.

A.2. Running example

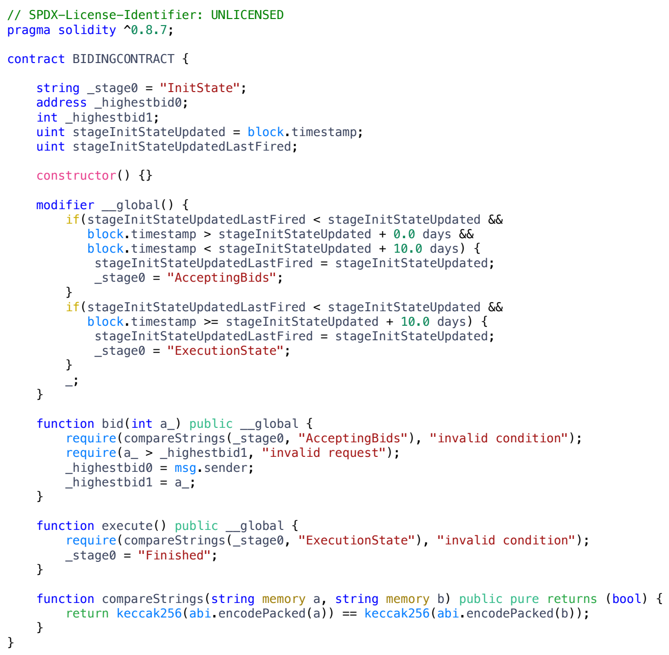
In addition to the excerpt in Example 12 and the linked Solidity code of the presented pattern, we provide here the full version in DatalogMTL as well as the generated version.

```
@namespace erc20.
@activator transfer("Address","Int").
@activator approve("Address","Int").
@activator transferFrom("Address","Address","Int").

% Init rule, set balanceOf for caller
balanceOf(Caller,2000) :- msg(Caller).

% party initiates transfer
tran(Caller,To,Value) :- msg(Caller),transfer(To,Value).

% party approves transfer in future
allowances(Caller,Spender,Value) :- msg(Caller),approve(Spender,Value).

% Check whether balanceOf is enough,then transfer is verified
vTran(A,B,X) :- tran(A,B,X),balanceOf(A,Y),X<=Y.

% Apply transfer
[+][1,1] balanceOf(A,NewBalance) :- vTran(A,B,X),balanceOf(A,Y),
    NewBalance=Y-X.
[+][1,1] balanceOf(B,NewBalance) :- vTran(A,B,X),balanceOf(B,Y),
    NewBalance=Y+X.
[+][1,1] balanceOf(B,X) :- vTran(A,B,X),not balanceOf(B,_).

% third party applies transfer
tran(From,To,Value) :- msg(Caller),transferFrom(From,To,Value),
    allowances(From,Caller,ApprovedValue),ApprovedValue>=Value.

% update approved value
[+][1,1] allowances(From,Caller,NewValue) :- msg(Caller),
    transferFrom(From,To,Value),vTran(From,To,Value),
    allowances(From,Caller,ApprovedValue),NewValue=ApprovedValue-Value.

% Copy rules
[+][1,1] balanceOf(A,Y) :- balanceOf(A,Y),not vTran(A,_,_),not vTran(_,A,_).
[+][1,1] allowances(A,B,Y) :- allowances(A,B,Y),not vTran(A,_,_).
[+][1,1] allowances(A,B,Y) :- allowances(A,B,Y),not transferFrom(A,_,_).
[+][1,1] allowances(A,Caller,Y) :- allowances(A,Caller,Y),not msg(Caller).
```

Figure 4: ERC20 program in DatalogMTL

```
@namespace bidingContract.

@activator bid("Int").
@activator execute.

stage("InitState").
[+](0d,10d) stage("AcceptingBids") :- stage("InitState").
[+][10d,10d] stage("ExecutionState") :- stage("InitState").

[+][1,1] stage("ExecutionState") :- stage("ExecutionState"),not execute.
[+][1,1] stage("Finished") :- stage("ExecutionState"),execute.
action :- stage("ExecutionState"),execute.

inBid(A) :- bid(A), stage("AcceptingBids").
[+][1,1] highestBid(Caller,A) :- msg(Caller),inBid(A),highestBid(_,CA),A>CA.
[+][1,1] highestBid(S,CA) :- msg(Caller),inBid(A),highestBid(S,CA),A<=CA.
[+][1,1] highestBid(S,CA) :- highestBid(S,CA),not inBid(_).
```

Figure 5: Running example in DatalogMTL



```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.7;

contract BIDINGCONTRACT {

    string _stage0 = "InitState";
    address _highestbid0;
    int _highestbid1;
    uint stageInitStateUpdated = block.timestamp;
    uint stageInitStateUpdatedLastFired;

    constructor() {}

    modifier __global() {
        if(stageInitStateUpdatedLastFired < stageInitStateUpdated &&
           block.timestamp > stageInitStateUpdated + 0.0 days &&
           block.timestamp < stageInitStateUpdated + 10.0 days) {
            stageInitStateUpdatedLastFired = stageInitStateUpdated;
            _stage0 = "AcceptingBids";
        }
        if(stageInitStateUpdatedLastFired < stageInitStateUpdated &&
           block.timestamp >= stageInitStateUpdated + 10.0 days) {
            stageInitStateUpdatedLastFired = stageInitStateUpdated;
            _stage0 = "ExecutionState";
        }
        _;
    }

    function bid(int a_) public __global {
        require(compareStrings(_stage0, "AcceptingBids"), "invalid condition");
        require(a_ > _highestbid1, "invalid request");
        _highestbid0 = msg.sender;
        _highestbid1 = a_;
    }

    function execute() public __global {
        require(compareStrings(_stage0, "ExecutionState"), "invalid condition");
        _stage0 = "Finished";
    }

    function compareStrings(string memory a, string memory b) public pure returns (bool) {
        return keccak256(abi.encodePacked(a)) == keccak256(abi.encodePacked(b));
    }
}
```

Figure 6: Generated Running Example