Knowlog^K: A Declarative Language for Reasoning about Knowledge in Distributed Systems

Matteo Interlandi

University of Modena and Reggio Emilia
{matteo.interlandi}@unimore.it

Abstract. In the last few years, database researchers started to investigate how recursive queries and deductive languages can be applied to find solutions to the new emerging trends in distributed computing. We think that their works are extremely valuable, but a missing piece in their proposals is the capability to express statements about the knowledge state of distributed nodes. In fact, reasoning about the state of remote nodes is fundamental in distributed contexts in order to design and analyze protocol behavior or perform coordinated actions. To reach this goal, we leveraged Datalog$^\neg$ with an epistemic modal operator, allowing the programmer to directly express nodes' state of knowledge instead of low level communication details. As a result, complex algorithms can be implemented with significative less orders of magnitude in terms of lines of code. Moreover, reasoning about high level properties of distributed programs can be performed. To support the effectiveness of our proposal, we introduce as example, the declarative implementation of a well-known protocol employed to execute distributed databases transactions: the two phase commit protocol.

1 Introduction

Pushed by the new interest that Datalog is acquiring in the database community, the goal of this paper is to open a new direction in the investigation on how Datalog could be adopted to program distributed systems. Many authors have stated how logic programming in general [1] and Datalog in particular [2] seems to particularly fit the representation of distributed programs implementation and properties. The demonstration of the CALM conjecture [3] the last year at PODS is one more proof on how this two worlds are tied together. We think that a missing point is the possibility to express statements about the knowledge state of distributed nodes in Datalog. In fact, the ability to reason about the knowledge state of remote nodes has been demonstrated [4] to be a fundamental tool in multi-agent systems in order to specify global behaviors and properties of protocols.

Motivated by all these facts, we leveraged Datalog$^\neg$ with an epistemic modal operator, allowing the programmer to express directly nodes' state of knowledge instead of low level communication details. We also provide a bottom-up evaluation algorithm grounded on the evaluation process for active [5, 6] and modal deductive databases [7]. To support our assertions, we describe our implementation of the two phase commit protocol.

The remainder of the paper is organized as follow: Section 2 contains some preliminary notations about Datalog dialects Knwolog is based upon. Section 3 describes what we intend for a distributed system and introduces some concept such as *global state* and *run* that will be used in Section 4 to specify the modal operator K. In addition Section 4 contains the two phase commit protocol implementation in Knowlog. The paper finish with Section 5 which specifies the Knowlog semantics and conclusions.

2 Preliminaries

In order to define Knowlog, we first introduce some principles of Datalog$^\neg$ [8, 6], and Datalog$^\neg$ augmented with temporal constructs [5, 9]. A Datalog$^\neg$ rule is an expression in the form:

$$H(\bar{u}) \leftarrow B_1(\bar{u}_1), ..., B_n(\bar{u}_n), \neg C_1(\bar{v}_1), ..., \neg C_m(\bar{v}_m)$$

where $n, m \geq 0$, H, B_i, C_j are relation names $i = 0, ..., n$ and $j = 0, ..., m$ and $\bar{u}, \bar{u}_i, \bar{v}_j$ are tuples of appropriate arities. Tuples are composed by *terms* and each term can be a constant in the domain **dom** or a variable in the set **var**. We will use interchangeably terms *predicates*, *relations* and *tables*. As usual $H(\bar{u})$ is referred as the *head*, $B_i(\bar{u}_i)$, $C_j(\bar{v}_j)$ as the *body*, and in general $H(\bar{u})$, $B_i(\bar{u}_i)$ and $C_j(\bar{v}_j)$ as *atoms*. A *literal* is an atom (in this case we refer to it as *positive*) or the negation of an atom. If $m = 0$, the rule is in Datalog form, while if $m = n = 0$ the rule is expressing a *fact* or equivalently a *groud atom*. A predicate appearing in the head of a rule can be used as a body predicate in another rule, therefore implementing recursive computation. We allow *build-in* predicates to appear in the body of rules. Thus, we allow relation names such as $=$,$\neq$,$\leqslant$, $<$, $\geqslant$, and $>$. We also allow *aggregate operations* in rule heads in the form $R(\bar{u}, \Lambda < \bar{\mu} >)$ with Λ one of the usual aggregate functions and $< \bar{\mu} >$ defining the grouping of arguments $\bar{\mu}$ [**?**]. In this paper we assume that each rule is *safe*, i.e. every variable occurring in a rule-head appears in at least one positive literal of the rule body. Then, a *Datalog*$^\neg$ *program* Π is a set of safe rules. As usual, we refer to the *itensional* part of the database schema $idb(\Pi)$, as the set of relations that appears in at least one Π's rule-head, while we refer to the *extensional database* $edb(\Pi)$ as the set of relations that do not appear in any Π's rule-head. For a database schema $\mathbf{R}$, a *database instance* is a finite set $\mathbf{I}$ constructed by the union of the relation instances over R with $R \in \mathbf{R}$ a relation name and where each relation instance is a finite set of facts.

As introductory example, we use the program depicted in Listing 1.1 where with two rules we are able to trivially compute the transitive closure of an intensional relation. In our example we used an *edb* relation `link` containing tuples in the form `(S,D)` which specifies the existence of a link between the source node S and the destination node D. In addition we employ an intensional relation path with tuples, as above, in the form (S,D), which is computed starting from the link relation (r1) and recursively adding a new path when, roughly speaking, it exists a link from A to B and already exists a path from B to C (r2).

```
r1: path(X,Y):-link(X,Y)
r2: path(X,Z):-link(X,Y),path(Y,Z)
```

Listing 1.1. Simple Recursive Datalog Program

2.1 Time in Datalog$^\neg$

With the language we are introducing, we want to model programs for distributed systems. These systems are not static, but evolving with time. Therefore it will be useful to enrich Datalog$^\neg$ with some notion of time. To reach this goal we follow the road traced by Statelog [5] and Dedalus [9]. A new schema $\mathbf{R}^T$ is defined starting from $\mathbf{R}$ incrementing the arity of each relation $R \in \mathbf{R}$ by one, and introducing a new build-in relation succ with arity two. succ(x, y) is interpreted true if $y = x+1$. By convention, the new extra term, called *time-step identifier*, appears as the first component in every relation and has values in $\mathbb{N}$. A predicate over $\mathbf{R}^T$ has, therefore, the (reified) form $R(s, t_1, ..., t_n)$ or equivalently $[s]R(t_1, ..., t_n)$ where s is the time-step identifier term $s \in S \cup \mathbb{N}$ with S a new set of variables disjoined from **var**. Introduced the new definition of relation that incorporates the time-step identifier term, with $\mathbf{I}[0]$ is named the *initial database instance* comprising all ground atoms existing at the initial time 0 and with $\mathbf{I}[n]$ the instance at the time-step n. A consequence of this approach is that tuples by default are considers *ephemeral*, i.e., they are valid only for one single time-step. Obviously, tuples can became *persistent* - once derived, for example at time s, they last for every time $t \geq s$ - if they are stored in *persistent* relations. For each persistent relation $P \in \mathbf{R}^T$, a new δ relation del_P is added to the database schema with the semantics that facts in $[s]del\_P$ will be removed from P at time $t = s+1$. In addition, a new *persistency* rule is added to the program in order to move towards time-steps tuples that don't have to be deleted. δ relations can be used both in the head and in the body of rules, but they cannot be persistent. In addition, predicates related to the next time-step can be specified only in head-rules. With this simple formalism we are not only able to maintain base relations, but also to achieve materialized views: i.e. persisting *idb* relations.

Among the different temporal extensions of Datalog$^\neg$ available in the literature, we embrace the Dedalus [9] notation, thus programs' rules are divided in two sets: *inductive* and *deductive*. The former set contains all the rules employed for transfer tuples among time-steps, while the latter encompasses the rules that are local into a single time-step. Some syntactic sugar is adopted in order to better characterize rules and relations: all the time specifier prefix are omitted together with the succ relation, this deductive rules appears as usual Datalog$^\neg$ rules, while a next suffix is introduced in head relations to characterize inductive rules. In Listing 1.2 the simple program of the previous section is rewritten to introduce the new formalism: a persistent relation rule (`r1`) and a rule for the modification of the `link` relation if a tuple is issued to the ephemeral relation `link_down`, representing the happening of an event on the link which cause the link to be disconnected.

```
r1: link(X,Y)@next:-link(X,Y),¬del_link(X,Y)
r2: del_link(X,Y):-link_down(X,Y)
r3: path(X,Y):-link(X,Y)
r4: path(X,Z):-link(X,Y),path(Y,Z)
```

Listing 1.2. Inductive and Deductive Rules

3 Distributed Logic Programming

Before starting the discussion on how we leverage the language with epistemic operators, we first introduce our model of distributed system and how communication among nodes is performed. We define a distributed message-passing system to be a non empty finite set N of share-nothing nodes joined by bidirectional communication links. Each node identifier has a value in the domain **dom** but here we consider the set $N = \{1, ..., n\}$ of node identifiers, where n is the total number of nodes in the system.

We identify with *adb* a new set of *accessible* relations encompassing all the tables in which facts are created remotely or need to be delivered to another node. These relations can be viewed as tables that are horizontally partitioned among nodes and through which nodes are able to communicate. Each relation $R \in$ *adb* contains a *location specifier* term [10]. This term maintains the identifier of the remote node to which every new fact inserted into the relation R must be issued. Hence, the nature of *adb* relations can be considered twofold: at one perspective they act as a normal relation, but from another perspective they are local buffers associated to relations stored in remote nodes. As pointed out in [2, 9], modeling communication using relations provides major advantages. For instance, the disordered nature of sets appears particularly appropriate to represent the basic communication channel behavior in which messages are delivered out of order. If reliability, sequentiality or higher level properties must be ensured in the communication, they can be obtained adding program modules implementing the required property.

Continuing with the examples introduced in the previous sections, we can now use it to actually program a distributed routing protocol. In order to describe the example of Listing 1.3 we can imagine a real network configuration where each node has locally installed the program, and where each `link` relation reflect the actual state of the connection between nodes. For instance, we will have the fact `link(A,B)` in node A's instance if it exists a communication link between A and node B. The location specifier term is identified by the `@` prefix.

```
r1: link(X,Y)@next:-link(X,Y),¬del_link(X,Y)
r2: del_link(X,Y):-link_down(X,Y)
r3: path(@X,Y):-link(X,Y)
```

```
r4: path(@X,Z):-link(X,Y),path(@Y,Z)
```

Listing 1.3. Distributed Program

The semantics of the program in Listing 1.3 is the same as in the previous section, even though operationally it substantially differs. In fact, in this new version, computation is performed simultaneously on multiple distributed nodes. Communication is achieved through rule `r4` which, informally, specify that a path from a node A to a node C exists if there is a link from A to another node B and this last knows that exist a path from B to C.

The body of rule `r4` has relations stored in different nodes, therefore implicitly assuming that tuples are exchanged between nodes and that the computation of the join is local to a single node. In order to have an explicit notion of communication (the location specifier of the head differ from location specifier of the relations in the body), and the computation local to a single node (the relations in the body all have the same location specifier), a *rule localization rewrite* algorithm is employed [10]. Rule `r4` is hence rewritten in the two rules of Listing 1.4.

```
r4_a: link_r(@Y,X):-link(X,Y)
r4_b: path(@X,Z):-link(Y,X),link_r(@Y,X),path(@Y,Z)
```

Listing 1.4. Rewriting of Rule `r4` of Listing 1.3

We denote inductive and deductive rules as *local*, while rules in the form of `r4_a` and `r4_b` as *communication* rules. It is interesting to note that based on the employed rule rewriting algorithm, different protocols can emerge [2]. The behavior of programs can be shaped using rewriting algorithms that are appropriate to the given context.

3.1 Local State, Global State and Runs

In every point in time, each node is in some particular *local state* incapsulating all the information the node possesses. The local state s_i of a node $i \in N$ can then be defined as a tuple $(\Pi_i, \mathcal{I}_i)$ where Π_i is the finite set of rules composing the node i program, and $\mathcal{I}_i \subseteq \mathbf{I}[n]_i$ is a set of facts belonging to the node i.

We define the *global state* of a distributed system as a tuple $(s_1, ..., s_n)$ where s_i is the node i's state. We define how global states may change over time through the notion of *run*, which binds time values to global states. If we assume time values to be isomorphic to the set of natural numbers, we can define the function r as $r : \mathbb{N} \to \mathcal{G}$ where $\mathcal{G} = \{S_1 \times ... \times S_n\}$ with S_i be the set of possible local state for node $i \in N$. We refer to the a tuple (r, t) consisting of a run r and a time t as a *point*. If $r(t) = (s_1, ..., s_n)$ is the global state at point (r, t), we define $r_i(t) = s_i$, for $i = 1, ..., n$; thus, $r_i(t)$ is node i's local state at the point (r, t). To note that the (real) time and the notion of time(-step) incapsulated in the program so far are two different entity. We will investigated in the future how this notions can be linked to define for example, synchronous or asynchronous systems.

A system may have many possible runs, indicating all the possible ways the global state of the system can evolve. Following [4] we define a *system* as a set of runs. Using this definition we are able to deal with a system not as a collection of interacting nodes but, instead, directly modeling its behavior, abstracting away many low level details. We think that this approach is particularly important in our scope of maintaing in our language an high level of declarativity.

4 Reasoning About Knowledge in Distributed Systems

In knowledge-based systems, processes are able to accomplish actions not only based on their local state, but also on the knowledge the process has, i.e., the information the process has about the state of the system. In the model we have developed so far, all the actions that a process can accomplish are consequences of its local state. In fact, if we consider two runs of a system, with global states respectively $g = (s_1, ..., s_n)$ and $g' = (s'_1, ..., s'_n)$, s and s' are *indistinguishable* for process i, and we will write $s \sim_i s'$ if i has the same local state both in s and s', i.e., $s_i = s'_i$.

How can we incorporate in our framework statements about knowledge in order to better design distributed algorithms encompassing multiple processes? It has been shown in [4] that a system $\mathcal{S}$ can be viewed as a *Kripke frame*. A Kripke frame is a tuple $\mathcal{F} = (W, \mathcal{K}_1, ..., \mathcal{K}_n)$ where W is a non empty set of *possible worlds* (in our case global states) and $\mathcal{K}_i$ with $i \in N$ is a binary relation in $W \times W$ which is intended to capture the *possibility relation* according to process i: i.e., $(s,t) \in \mathcal{K}_i$ if process i consider world t possible given its information in world s. Or, in other words, we want $\mathcal{K}$ to be equivalent to the $\sim$ relation, therefore maintaining the intuition that a process i considers t possible in global state s if they are indistinguishable. To model this situation, $\mathcal{K}$ must be an *equivalent relation* on $W \times W$. $\mathcal{K}$ to be an equivalence relation must be a binary relation satisfying the following properties:

– *reflexive*: for all $s \in W$, $(s,s) \in \mathcal{K}$
– *symmetric*: for all $s, t \in W$, $(s,t) \in \mathcal{K}$ iif $(t,s) \in \mathcal{K}$
– *transitive*: for all $s, t, u \in W$, if $(s,t) \in \mathcal{K}$ and $(t,u) \in \mathcal{K}$ then $(s,u) \in \mathcal{K}$

To map each rule and fact to the global states in which they are true, we define an *interpreted system* Γ as the tuple $(\mathcal{R}, \pi)$ with $\mathcal{R}$ a system over a set of global states $\mathcal{G}$ and π as an interpretation function which maps first-order clauses to global states. More formally, we build a structure over the Kripke frame $\mathcal{F}$ in order to map each program Π_i and each ground atom in $\mathcal{I}_i$ to the possible worlds in which they are true. To reach this goal, we define a *Kripke structure* $M = (\mathcal{F}, U, \pi)$ where $\mathcal{F}$ is a Kripke frame, U is the *Herbrand Universe*, π is a function which map every possible world to a *Herbrant interpretation* over first-order clauses $\Sigma_{\Pi,\mathbf{I}}$ associated with the rules of the program Π and the input instance $\mathbf{I}$, and $\Pi = \bigcup_{i=1}^{n} \Pi_i$, $\mathbf{I} = \bigcup_{i=1}^{n} \mathbf{I}_i[0]$. To be precise, $\Sigma_{\Pi,\mathbf{I}}$ can be constructed starting from the program Π and translating each rule $\rho \in \Pi$ in its first-order *Horn clause* form. This process creates the set of sentences Σ_Π. In order to get the logical theory $\Sigma_{\Pi,\mathbf{I}}$, starting from Σ_{Π} we add one sentence $R(\bar{u})$ for each fact $R(\bar{u})$ in the instance [8, 11]. A valuation v on M is now a function that assign to each variable a value in U. In our settings both the interpretation and the variables valuation are fixed. This means that $v(x)$ is independent of the state, and a constant c has the same meaning in every state in which exists. Thus, constants and relation symbols in our settings are *rigid designators* [4, 7].

Given a Kripke structure M, a world $s \in W$ and a valuation v on M, the *satisfaction relation* $(M, s, v) \models \psi$ for a formula $\psi \in \Sigma_{\Pi,\mathbf{I}}$ with $s = (r, t)$ is:

- $(M, s, v) \models R(t_1, ..., t_n)$ with R a relation and $n = \mathit{arity}(R)$, iff $(v(t_1), ..., v(t_n)) \in \pi(s)(R)$
- $(M, s, v) \models \neg\psi$ iif $(M, s, v) \models \psi$
- $(M, s, v) \models \psi \wedge \phi$ iff $(M, s, v) \models \psi$ and $(M, s, v) \models \phi$
- $(M, s, v) \models \forall\psi$ iif $(M, s, v[x/a]) \models \psi$ for every $a \in U$ with $v[x/a]$ be a substitution of x with a constant a

It could be also interesting to know not only whether certain formula ψ is true in a certain world, but also the formulas that are true in all the worlds of W. In particular, a formula ψ is *valid* in a structure M, and we write $M \models \psi$, if $(M, s) \models \psi$ for every world s in W. We say that ψ is valid, and write $\models \psi$, if ψ is valid in all structures.

We now introduce the modal operator K_i in order to express what node i knows, namely which of the sentences in $\Sigma_{\Pi,\mathbf{I}}$ are known by the node i. Given $\psi \in \Sigma_{\Pi,\mathbf{I}}$, a world s in the Kripke structure M, the node i knows ψ in world s if it is true at all the worlds that i considers possible in s. Formally:

$$(M, s, v) \models K_i\psi \text{ iff } (M, t, v) \models \psi \text{ for all } t \text{ such that } (s, t) \in \mathcal{K}_i$$

This definition of knowledge has the following valid properties that are usually called *S5*:

1. *Distributed Axiom*: $\models (K_i\psi \wedge K_i(\psi \rightarrow \phi)) \rightarrow K_i\phi$
2. *Knowledge Generalization Rule*: For all structures M, if $M \models \psi$ then $M \models K_i\psi$
3. *Truth Axiom*: $\models K_i\psi \rightarrow \psi$
4. *Positive Introspection Axiom*: $\models K_i\psi \rightarrow K_iK_i\psi$
5. *Negative Introspection Axiom*: $\models \neg K_i\psi \rightarrow K_i\neg K_i\psi$

Informally, the first axiom allows us to distribute the epistemic operator K_i over implication; the knowledge generalization rule instead says that if ψ is valid, then so is $K_i\psi$. This rule differ from the formula $\psi \rightarrow K_i$, in the sense that the latter tells that if ψ is true, then node i knows it, but a process does not necessarily know all things that are true. Even though a process may not know all facts that are true, axiom 3 says that if it knows a fact, then it is true. The last two properties say that nodes can do introspection regarding their knowledge: they know what they know and what they do not know.

4.1 Incorporating Knowledge: Knowlog

In the previous section we described how knowledge assumptions can be expressed using first-order Horn clauses representing our program. We now move back to the rule form. We use $\Box$ to denote a (possible empty) sequence of modal operators K. Given a sentence in the modal Horn clause form and with modal operators K, we use the following statement to express it in a rule form:

$$\Box(H \leftarrow B_1, ..., B_n, \neg C_1, ..., \neg C_m) \tag{1}$$

with $n, m \geq 0$ and each positive literal is in the form $\Box R$, while negative literals are in the form $K_i \Box R$ where K_i is equal to the *model context.*. From [7] we adopt the term *modal context* to refer to the sequence - with the maximum length of one - of modal operators appearing in front of a rule. We put some restriction on the sequence of operators permitted in $\Box$.

Definition 1. *Given a (possibly empty) sequence of operators $\Box$, $\Box$ is in restricted form if it does not contain $K_i K_i$ subsequences, with i specifying a process identifier.*

Definition 2. *A Knowlog program is a set of rules in the form (1), containing only (possible empty) sequences of modal operators in the restricted form and where the subscript i of each modal operator K_i can be a constant or a variable.*

Informally speaking, given a Knowlog program, using modal context we are able to assign to each node the rules the node is responsible for, while atoms and facts residing in the node i are in the form $K_i \Box R$. In order to specify how communication is achieved we redefine communication rules as follow:

Definition 3. *A communication rule in Knowlog is a rule where no modal context is set and the body atoms have the form $K_i \Box R$ - namely they are prefixed with modal operators related to the same process - while the head atom has the form $K_j \Box R'$, with $i \neq j$ and not necessarily $R' \neq R$.*

In this way, we are able to abstract away all the low level details about how information is exchanged, leaving to the programmer just the task to specify what a process should know, and not how. To note that in communication rules, no atoms can be prefixed with the empty sequence.

The Two-Phase-Commit Protocol Taking cue from [12], we implemented the two-phase-commit protocol (2PC) using the epistemic operator K. 2PC is used to execute distributed databases transaction and can be divided in two phases:

– In the first phase, called the *voting phase*, a coordinator node submit to all the transaction's participants the willingness to perform a distributed commit. Consequently, each participant sends a vote to the coordinator, expressing its intention (a *yes vote* in case it is ready, a *no vote* otherwise)

– In the second phase - namely the *decision phase* - the coordinator collects all votes and decides if performing global *commit* or *abort*. The decision is then issued to the participants which act accordingly.

In the 2PC implementation of Listing 1.5, we assume that our system is composed by three nodes: one coordinator and two partecipants. We considerably simplify the 2PC protocol by disregarding failures and timeouts actions, since our goal is not an exhaustive exposition of the 2PC. Moreover, we dropped the relations persistency rules, as it is trivial to identify from the program which relations are persistent and which not. Moreover we assume that communication channels are reliable.

```
\\Initialization at coordinator
   r1: $K_C$(part_cnt(count<N>):-nodes(N))
   r2: $K_C$(transaction(Tx_id,State):-log(Tx_id,State))

\\Decision Phase at coordinator
   r3: $K_C$(yes_cnt(Tx_id,count<part>):-vote(Vote,Tx_id,part),Vote == "yes"))
   r4: $K_C$(log(Tx_id,"commit")@next:-part_cnt(C),yes_cnt(Tx_id,C1),C==C1,
        State=="vote-req",transaction(Tx_id,State))
   r5: $K_C$(log(Tx_id,"abort"):-vote(Vote,Tx_id,part),Vote == "no",
        transaction(Tx_id,State),State =="vote-req")

\\Voting Phase at partecipants
   r6: $K_P$(log(Tx_id,"prepare"):-State=="vote-req",$K_C$transaction(Tx_id,State))
   r7: $K_P$(log("abort",Tx_id):-log(Tx_id,State),State=="prepare",
        db_status(Vote),Vote=="no")

\\Decision Phase at partecipants
   r8: $K_P$(log(Tx_id,"commit"):-log(Tx_id,State_l),State_l=="prepare",
        State_t=="commit",$K_C$transaction(Tx_id,State_t))
   r9: $K_P$(log(Tx_id,"abort"):-log(Tx_id,State_l),State_l=="prepare",
        State_t=="abort",$K_C$transaction(Tx_id,State_t))

\\Communication
   r10:$K_X$transaction(Tx_id, State):-$K_C$subs(X),
        $K_C$transaction(Tx_id,State),$K_C$path(@Y,X)
   r11:$K_C$vote(Vote,Tx_id,"sub1"):-$K_{P1}$log(Tx_id,State),
        State=="prepare",$K_{P1}$db_status(Vote)
   r12:$K_C$vote(Vote,Tx_id,"sub2"):-$K_{P2}$log(Tx_id,State),
        State=="prepare",$K_{P2}$db_status(Vote)
```

Listing 1.5. Two Phase Commit Protocol

In the above example, for simplicity we wrote K_P as a modal context instead of K_{P1} and K_{P2}. This program is a typical example showing how logical programming permits to specify even complex algorithms using few lines of declarative codes, which are almost a faithful translation of algorithms specified in pseudocode [2, 1].

5 Knowlog Semantics

The first step towards the definition of the Knowlog semantics will be the specification of the *reified* version of Knowlog. For this purpose, we augment **dom** with a new set of constants $\triangle$ which will encompass the modal operators symbols. We also assume a new set of variables O that will range over the just defined set of modal operator elements. We then construct $\mathbf{R}^{TK}$ adding to each relation $R \in \mathbf{R}^T$ a new term called *knowledge accumulator* and a new set of build-in relations `K` and $\oplus$. A tuple over the $\mathbf{R}^{TK}$ schema will have the form $(k, s, t_1, ..., t_n)$ where $k \in O \cup \triangle$ identify the knowledge accumulator term, $s \in S \cup \mathbb{N}$ and $t_1, ..., t_n \in \mathbf{var} \cup \mathbf{dom}$. Conversely, a tuple over *adb* relations, i.e., relations in the head of at least one communication rule, will have the form $(k, l, s, t_1, ..., t_n)$, with l the location specifier term.

If the knowledge operator used in front of a non-*adb* relation is a constant, i.e., $\mathtt{K_S K_R}$`input("value")`, the reified version will be `input(Y,n,"value"),Y = `$\mathtt{K_S} \oplus \mathtt{K_R}$ for example at time-step n. The operator $\oplus$ is hence employed to concatenate epistemic operators. Instead, in case the operator employes a variable in order to identify a particular node, we need to introduce in $\mathbf{R}^{TK}$ relations `K(X,Y)` in order to help us in the effort of building the knowledge accumulator term. The semantics of this relations is straightforward. The first term of the `K` relation is a node identifier $i \in N$ and the `Y` term is a value in $\triangle$ determined by the K operator and the node identifier. Thus, for example, the reified version of $\mathtt{K_X}$`transaction(Tx_id, State),parts(X)` will be `K(X,Y),transaction(Y,n,Tx_id, State),parts(X)`.

What about the modal context? We already mentioned that the modal context is used to identify the node(s) where a certain rule or fact must be installed. In the reified version of Knowlog, we push the modal context $\Box$ into the knowledge accumulator term, hence initially all the facts belonging to a node $i \in N$ will have the knowledge accumulator in the form $K_i\Box$. For what concern communication rules, the process is the same as above, but this time we have to fill also the location specifier field of the head-relation. To accomplish this, given the head relation $R \in adb$ in the form $K_i\Box R(t_1, ..., t_n)$, the reified version will be $R(K_i\Box, n, i, t_1, ..., t_n)$. Using this semantics, nodes are able to communicate using the mechanism described in Section 3.

5.1 Operational Semantics

Given as input a Knowlog program Π in the reified version, first Π is separated in two subset: Π^l containing local rules (informally these are the rules that the local nodes i knows) and Π^r containing instead rules that must be installed in remote nodes. These last are rules having as a modal context K_j with $j, i \in N$, i the identifier of the local node and $i \neq j$. Following the delegation approach illustrated in [13] given a program Π_i local to node $i \in N$, we denote with Π^r_{ij} the remote rules in i related to node j and with $\Pi_j \leftarrow \Pi^r_{ij}$ the action of installing Π^r_{ij} in j's program. For what concern the evaluation of local rules, we partition the local program Π^l in inductive and deductive rule sets, respectively Π^i and

Π^d. Then a pre-processing step order the deductive rules following the dependency graph stratification. After this pre-processing step, the perfect model $\mathcal{M}_\Pi$ is computed by the Algorithm 1 where we adopt the notation introduced in [10]. In order to evaluate the stratified program Π^d we use the semi-naive algorithm

Algorithm 1 Knowlog bottom-up evaluation

```
*Input:* a program $\Pi_i$; the initial database instance $\mathbf{I}_i[0]$
*Output:* the perfect model $\mathcal{M}_{\Pi_i}$
n := 0;
$\mathcal{M}_{\Pi_i}[0] \leftarrow \mathbf{I}_i[0]$;
**repeat**
    $\mathcal{M}_{\Pi_i}[n] \leftarrow \Pi^d(\mathcal{M}_{\Pi_i}[n])$;
    $\mathcal{M}_{\Pi_i}[n+1] \leftarrow \Pi^i(\mathcal{M}_{\Pi_i}[n])$;
    $\Pi_j \leftarrow \Pi^r_{ij}$;
    n := n + 1;
**until** $\mathcal{M}_{\Pi_i}[n+1] \neq \mathcal{M}_{\Pi_i}[n]$
```

depicted in Algorithm 2 [6], where the T_{Π^d} is the *immediate consequence* operator for program Π and Δp_i is the set containing the new derived facts at round i. To be precise, in order to correctly evaluate rules and facts with modal operators, we introduce a *saturation (Sat)* and *normalization* (*Norm*) operators assisting the immediate consequence operators T_{Π^d} [7]. This because Knowlog facts and rules are labeled by modal operators and therefore the T operator must be enhanced in order to be employed in our context. More precisely, given a Knowlog instance $\mathbf{I}$ as input, $Sat(\mathbf{I})$ saturate facts in the instances following the $S5$ logic axioms. Lastly, the *Norm* operator converts in restricted form the modal operators in $T_{\Pi^d}(Sat(\mathbf{I}))$. Algorithm 2 contains the pseudo-code for the evaluation of deductive Knowlog programs. In every new derived tuple in the set

Algorithm 2 Semi-naive algorithm for Knowlog

```
*Input:* a stratified deductive program $\Pi^d = \Pi^d_0 \cup ... \cup \Pi^d_k$; $\mathcal{M}_\Pi[n]$
*Output:* the perfect model $\mathcal{M}_{\Pi^d}$
$p_0 \leftarrow \mathcal{M}_\Pi[n]$;
$\Delta p_0 \leftarrow Norm(T^0_{\Pi^d}(Sat(p_0)))$;
$p_1 \leftarrow \Delta p_0 \cup p_0$;
**for** $i := 1$ to $k$ **do**
    **repeat**
        $\Delta p_i \leftarrow Norm(T^i_{\Pi^d}(Sat(p_i))) - p_i$;
        $p_{i-1} \leftarrow p_i$;
        $\Delta p_{i-1} \leftarrow \Delta p_i$;
        $p_i \leftarrow p_{i-1} \cup \Delta p_{i-1}$;
    **until** $\Delta p_{i-1} \neq 0$
**end for**
```

Δp is managed differently based on its left-most modal operator. For instance, if the atom is in the form $K_i\Box R$ with i the local node, no particular action is performed, meanwhile if the atom is in the form $K_j\Box R$ with i the local node and $i \neq j$, the new derived fact is issued to node j.

6 Conclusion and Future Work

In this paper we present Knowlog, a programming language based on Datalog$^\neg$ leveraged with a notion of time and modal operators. Through Knowlog, reasoning about state of knowledge in distributed systems can be performed, therefore lighten the programmer's burden of expressing low level communication details. What we discussed here is a first step towards the definition of a comprehensive logical framework able to define a declarative as well as operational semantics, and generic enough to be adopted in multiple contexts. We are confident that following our approach, properties such as processes coordination and replicas consistency can be exhaustively defined. To this purpose, we will incorporate in Knowlog the *distributed knowledge*, and overall the *common knowledge* operator that has be proven to be linked to concepts such as coordination, agreement and consistency [4]. The following step will be the definition in Knowlog of weaken forms of common knowledge such as *eventual* common knowledge. Other interesting steps that will be pursued is the definition of the model-theoretic semantics of Knowlog starting from the work of [7]. We are currently developing a prototype as a plug-in on top of Bud [14].

References

1. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16** (May 1994) 872–923
2. Hellerstein, J.M.: The declarative imperative: experiences and conjectures in distributed logic. SIGMOD Rec. **39** (September 2010) 5–19
3. Ameloot, T.J., Neven, F., Van den Bussche, J.: Relational transducers for declarative networking. In: Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. PODS '11, New York, NY, USA, ACM (2011) 283–292
4. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge, MA, USA (2003)
5. Ludäscher, B.: Integration of Active and Deductive Database Rules. Volume 45 of DISDBIS. Infix Verlag, St. Augustin, Germany (1998)
6. Zaniolo, C.: Advanced database systems. Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers (1997)
7. Nguyen, L.A.: Foundations of modal deductive databases. Fundam. Inf. **79** (January 2007) 85–135
8. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
9. Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.C.: Dedalus: Datalog in Time and Space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley (December 2009)

10. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: language, execution and optimization. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. SIGMOD '06, New York, NY, USA, ACM (2006) 97–108
11. Lloyd, J.: Foundations of logic programming. Symbolic computation: Artificial intelligence. Springer (1987)
12. Alvaro, P., Condie, T., Conway, N., Hellerstein, J.M., Sears, R.: I do declare: consensus in a logic language. Operating Systems Review **43**(4) (2009) 25–30
13. Abiteboul, S., Bienvenu, M., Galland, A., Antoine, E.: A rule-based language for web data management. In: Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. PODS '11, New York, NY, USA, ACM (2011) 293–304
14. Alvaro, P., Conway, N., Hellerstein, J., Marczak, W.R.: Consistency analysis in bloom: a calm and collected approach. In: CIDR. (2011) 249–260