Knowlog: Knowledge + Datalog for Distributed Systems

Matteo Interlandi
Supervised by Sonia Bergamaschi
University of Modena and Reggio Emilia
Modena, Italy
matteo.interlandi@unimore.it

ABSTRACT

One of the new emerging trends that is gaining a lot of attentions in the database community is about distributed programming in large datacenters. A lot of discussions are arising around the CAP theorem [8] and how to achieve correct and efficient programs while performing less coordinated actions as possible. In order to address these issues, monotonic logic programming [4] has been employed to formally specify eventually consistent distributed programs.

We conjecture that a missing piece in the current state-of-the-art is the capability to express statements about the knowledge state of distributed nodes, i.e., statements about what a node "knows" given the current system configuration. In fact, reasoning about the state of remote nodes is fundamental in distributed contexts in order to design and analyze protocols behavior or perform coordinated actions [10]. To reach this goal, we leverage 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, 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 the well-known protocol employed to execute distributed transactions: the two phase commit protocol.

1. INTRODUCTION

Many authors have stated how logic programming in general [14] and Datalog in particular [12] seems to particularly fit in expressing distributed programs' implementation and properties. The demonstration of the CALM conjecture [6] the last year at PODS is one more proof on how these two worlds are tied together.

However, we think that a missing point in the current literature is the possibility to express statements about the knowledge state of distributed nodes in Datalog, i.e., statements defining which sentences are "*known*" by a node given its current information. In fact, the ability to reason about knowledge states has been demonstrated [10] to be a fundamental tool in multi-agent systems in order to specify global behaviors and properties of protocols. Furthermore it can be noticed that a program composed by production rules is very similar to (standard) program for multi-agent systems, where a node i executes an action a_j if the corresponding j-th condition is evaluated true. Conversely, a knowledge-base program has the form:

$$\textbf{if} < knowledge\_condition > \wedge < condition > \textbf{then} < action >$$

where the *action* is fired if the *knowledge_condition* is true and the *condition* is satisfied by the node's local state. Motivated by all these facts, we leveraged Datalog$^\neg$ with the epistemic modal operator K, allowing the programmer to express directly nodes' state of knowledge instead of low level communication details. The advantage of this formalism is that it abstracts away all the mechanisms by which the knowledge is exchanged (message passing, shared memory, etc) maintaing, thus, an high level of declarativity in the language. To support our assertions, we describe an implementation of the two phase commit protocol.

The remainder of the paper is organized as follows: Section 2 contains some preliminary notations about Datalog dialects our language is based upon. Section 3 describes what we intend for a distributed system and it presents some concepts such as *global state* and *run* that will be used in subsequent sections. Section 4 specifies the modal operator K and introduces how this modal operator can be embedded in Datalog in order to define Knowlog. In addition, Section 4 contains the two phase commit protocol implementation. The paper finishes with Section 5 which specifies Knowlog's semantics, Section 6 which contains related works and Section 7 composed by some concluding remarks and future work.

2. PRELIMINARIES

Before defining Knowlog, we first introduce some principles of Datalog$^\neg$ [2, 23], and Datalog$^\neg$ augmented with temporal constructs [19, 5]. 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 interchangeably use the words *predicate*, *relation* and *table*. 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 and express a *definite clause*, while if $m = n = 0$ the rule is expressing a *fact clause*. We refer to a fact clause directly as a *fact* or equivalently as a *groud atom* if it does not contain variable terms. A predicate appearing in the body of a rule can also be used as head in the same rule, therefore implementing recursive computation. We allow *built-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 $\bar{\mu}$ [22].

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 relations' instances over R, with $R \in \mathbf{R}$ a relation name and where each relation's instance is a finite set of facts.

As introductory example, we use the program depicted in Listing 1 where with two rules we are able to trivially compute the transitive closure of an extensional 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, a link exists from `A` to `B` and a path already exists from `B` to `C` (`r2`).

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

Listing 1: Simple Recursive Datalog Program

2.1 Time in Datalog$^\neg$

With the language that we are going to introduce, 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 a notion of time. For this purpose we follow the road traced by Dedalus$_0$ [5]. Thus, starting with considering time isomorphic to the set of natural numbers $\mathbb{N}_0$, a new schema $\mathbf{R}^T$ is defined starting from $\mathbf{R}$ by incrementing the arity of each relation $R \in \mathbf{R}$ by one, and introducing a new built-in relation `succ` with arity two. $\texttt{succ}(x, y)$ is interpreted true if $y = x + 1$. By convention, the new extra term, called *time suffix*, appears as the last attribute in every relation and has values in $\mathbb{N}_0$. A predicate over $\mathbf{R}^T$ has, therefore, the (reified) form $R(t_1, ..., t_n, s)$ or equivalently $R(t_1, ..., t_n)@s$ where s is the time suffix $s \in S \cup \mathbb{N}_0$ with S a new set of time variables disjoined from **var**. We will sometimes adopt the term *timestamp* to refer to the time suffix value. In fact, each tuple can be views as timestamped with the evaluation step in which it is valid. For conciseness we will employ the term *time-step* to denote an evaluation step.

By incorporating the time suffix term in the schema definition, we now have multiple instances for each relation, one for each timestamp. In this situation, with $\mathbf{I}[0]$ it is named the *initial database instance* comprising all ground atoms existing at the initial time 0, while with $\mathbf{I}[n]$ the instance at time n. As a consequence of this approach, tuples by default are considered *ephemeral*, i.e. they are valid only for one single time-step. Obviously, tuples can became *persistent*, i.e., once derived, for example at time s, they will eventually last for every timestamp $t \geq s$. To reach this goal, persistent relations are introduced: (I) for each intensional relation $P \in \mathbf{R}^T$ that should maintain tuples for consequent states, a new δ relation del_P is added to the database schema with the semantics that facts in $del\_P@s$ will not appear in P at state $t = s + 1$; (II) for each extensional relation R two predicates R_pos and (optionally) $R\_del \in \delta$ are added to the $\mathbf{R}^T$ schema. In addition, the following rule is added:

$$R\_pos@0 \leftarrow R$$

In this way for each extensional predicate R, one intensional relation exists that contains at least the tuple originally stored in R. The just introduced rules are the only ones permitted to involve extensional predicates. This property is called *guarded edb* [5]. (III) A new (mutual)*persistency* rule is added to the program in order to move towards time-steps tuples that don't have to be deleted:

$$P@t \leftarrow P@s, \neg del\_P@s, \texttt{succ}(s, t)$$

where $\neg del\_P@s$ is not mandatory. Predicates related to the next time-step can be specified only in rule head. With this simple formalism we are not only able to maintain base relations, but also to achieve materialized views: i.e. persisting *idb* relations. Program rules are then divided in two sets: *inductive* and *deductive*. The former set contains all the rules employed for transferring tuples through time-steps, while the latter encompasses rules that are instantaneous, i.e, local into a single time-step. Deductive rules are hence used to describe what is true in a single time-step given what is known at that point in time [5]. Some syntactic sugar is adopted to better characterize rules and relations: all time suffixes are eliminated together with the `succ` relation, and a `next` suffix is introduced in head relations to characterize inductive rules.

$$\begin{aligned} inductive &: head@next \leftarrow body \\ deductive &: head \leftarrow body \end{aligned}$$

In Listing 2 the simple program of the previous section is rewritten in order to introduce the new formalism. A rule for the modification of the `link` relation is also added, representing the happening of an event on a link which causes 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 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 distributed system model and how communication among nodes is performed. We define a distributed message-passing system to be a non empty finite set $\mathcal{N} = \{id_1, id_2, ..., id_n\}$ of share-nothing nodes joined by bidirectional communication links. Each node identifier has a value in the domain **dom** but, for simplicity, we assume that a node id_i is identified by its subscript i. Thus, in the followings we consider the set $N = \{1, ..., n\}$ of node identifiers, where n is the total number of nodes in the system.

Then, with *adb* we denote a new set of *accessible* relations encompassing all the tables in which either facts are created remotely or they 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 [17]. This term maintains the identifier of the node to which every new fact inserted into the relation R belongs. Hence, the nature of *adb* relations can be considered twofold: for one perspective they act as normal relations, but from another perspective they are local buffers associated to relations stored in remote nodes. As pointed out in [12, 5], modeling communication using relations provides major advantages. For instance, the disordered nature of sets appears particularly appropriate to represent the basic communication channel behavior by which messages are delivered out of order. If reliability, sequentiality or others higher level properties must be ensured, they can be obtained adding program modules implementing the required property.

Continuing with the same example of previous sections, we can now use it to actually program a distributed routing protocol. In order to describe the example of Listing 3 we can imagine a real network configuration where each node has the program locally installed, and where each `link` relation reflects the actual state of the connection between nodes. For instance, we will have the fact `link(A,B)` in node A instance if a communication link between A and node B exists. 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 3: Inductive and Deductive Rules

The semantics of Listing 3's program is the same of the previous sections, 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, specifies that a path from a generic node A to node C exists if there is a link from A to another node B and this last knows that a path exists from B to C. The body of rule `r4` contains relations stored in different nodes, therefore implicitly assuming that tuples are exchanged between nodes, and that the computation of the join(s) is local to a single node. In order to have an explicit notion of communication - i.e., the location specifier of the head differs from the location specifiers of the relations in the body - and the computation local to a single node - i.e., relations in the body all have the same location specifier - a *rule localization rewrite* algorithm is employed [17]. Rule `r4` is hence rewritten in the two rules of Listing 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 4: Rewriting of Rule `r4` of Listing 3

3.1 Local State, Global State and Runs

In every point in time, each node is in some particular *local state* encapsulating all the information the node is in possess. 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 node i's program, and $\mathcal{I}_i \subseteq \mathbf{I}[n]_i$ is a set of facts belonging to node i. We define the *global state* of a distributed system as a tuple $(s_1, ..., s_n)$ where s_i is node i's state. We define how global states may change over time through the notion of *run*, which binds (real) time values to global states. If we assume time values to be isomorphic to the set of natural numbers, we can define the function $r : \mathbb{N} \to \mathcal{G}$ where $\mathcal{G} = \{S_1 \times ... \times S_n\}$ with S_i be the set of possible local states 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) [10]. To note that the (real) time and the notion of time(stamp) incapsulated in the program so far are two different entities. We will investigate in the future how these 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. 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 the model we have developed so far, all computations that a node can accomplish are consequences of its local state. If we consider two runs of a system, with global states respectively $g = (s_1, ..., s_n)$ and $g' = (s'_1, ..., s'_n)$, g and g' are *indistinguishable* for node i, and we will write $g \sim_i g'$ if i has the same local state both in g and g', i.e. $s_i = s'_i$. It has been shown in [10] 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 a set of possible global states) and $\mathcal{K}_i$ with $i \in N$ is a binary relation in $W \times W$ which is intended to capture the *accessibility relation* according to node i: this is, $(w, u) \in \mathcal{K}_i$ if node i consider world u possible given its information in world w. Or, in other words, we want $\mathcal{K}$ to be equivalent to the $\sim$ relation, therefore maintaining the intuition that a node i considers u possible in global state w if they are indistinguishable, i.e., in both global states, node i has the same local state. In order to model this situation, $\mathcal{K}$ must be an *equivalence relation* on $W \times W$. $\mathcal{K}$ to be an equivalence relation must be a binary relation satisfying the following properties:

- *reflexive*: for all $w \in W$, $(w, w) \in \mathcal{K}$
- *symmetric*: for all $w, u \in W$, $(w, u) \in \mathcal{K}$ iif $(u, w) \in \mathcal{K}$
- *transitive*: for all $w, u, o \in W$, if $(w, u) \in \mathcal{K}$ and $(u, o) \in \mathcal{K}$ then $(w, o) \in \mathcal{K}$

To map each rule and fact to the global states in which they are true, we define an *interpreted system* Γ as the tuple $(\mathcal{S}, \pi)$ with $\mathcal{S}$ a system over a set of global states $\mathcal{G}$ and π an interpretation function which maps first-order clauses to global states [10]. 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* $\mathcal{M} = (\mathcal{F}, U, \pi)$ where $\mathcal{F}$ is a Kripke frame, U is the *Herbrand Universe*, π is a function which maps 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. 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 [2, 16]. A valuation v on $\mathcal{M}$ 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* [10, 20]. Given a Kripke structure $\mathcal{M}$, a world $w \in W$ and a valuation v on $\mathcal{M}$, the *satisfaction relation* $(\mathcal{M}, w, v) \models \psi$ for a formula $\psi \in \Sigma_{\Pi,\mathbf{I}}$ is as usual.

We now introduce the modal operator K_i in order to express what a generic 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 w in the Kripke structure $\mathcal{M}$, the node i knows ψ - we will write $K_i\psi$ - in world w if ψ is true in all the worlds that i considers possible in w [10]. Formally:

$(\mathcal{M}, w, v) \models K_i\psi$ iff $(\mathcal{M}, u, v) \models \psi$ for all u s.t. $(w, u) \in \mathcal{K}_i$

This definition of knowledge has the valid properties that are called *S5*. We refer the reader to [13] for a detailed discussion about the properties of the modal operator K.

4.1 Knowlog

In the previous section we described how knowledge assumptions can be expressed using first-order Horn clauses representing our program. We can now move back to the rule form. We use symbols $\Box$ and $\boxdot$ to denote a (possibly empty) sequence of modal operators K_i, with i specifying a node identifier. Given a sentence in the modal Horn clause form, 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 atom in the form $\boxdot R$.

Definition 1. The *modal context* $\Box$ is the sequence - with the maximum length of one - of modal operators K appearing in front of a rule.

We put some restriction on the sequence of operators permitted in $\boxdot$.

Definition 2. Given a (possibly empty) sequence of operators $\boxdot$, we say that $\boxdot$ is in *restricted form* if it does not contain K_iK_i subsequences.

Definition 3. A Knowlog program is a set of rules in the form (1), containing only (possibly 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, with the 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 $K_i \boxdot R$. In order to specify how communication is achieved we redefine communication rules as follows:

Definition 4. A communication rule in Knowlog is a rule where no modal context is set and body atoms have the form $K_i \boxdot R$ - they are all prefixed with a modal operators pointing to the same node - while the head atom has the form $K_j \boxdot R'$, with $i \neq j$.

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 node should know, and not *how*.

4.1.1 The Two-Phase-Commit Protocol

Inspired by [3], we implemented the two-phase-commit protocol (2PC) using the epistemic operator K. 2PC is used to execute distributed transaction and it is divided in two phases: in the first phase, called the *voting phase*, a coordinator node submits 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 5, we assume that our system is composed by three nodes: one coordinator and two participants. With `log(Tx_id,State)` and `transaction (Tx_id,State)` we denote respectively the log and the state of the transaction, where `Tx_id` is the term specifying the unique transaction identifier, while `State` is the current state of the transaction. We use the ephemeral relation `vote (Vote,Tx_id,Part)` to store the vote of the participant `Part` related to the transaction `Tx_id`. A vote is chosen by each participant based on the state of the local database (`db_ status(Vote)`). `part_cnt(count<N>)` and `yes_cnt(Tx_id, count<part>)` are two aggregate relations, with the first maintaining the number of participants, and the second counting the number of received `"yes"` votes. We considerably simplify the 2PC protocol by disregarding failures and timeouts actions, since our goal is not an exhaustive exposition of the 2PC.

```
\\Initialization at coordinator
  r1: Kc(transaction(Tx_id,State)@next:-
        transaction(Tx_id,State),
        ¬del_transaction(Tx_id,State)).
  r2: Kc(log(Tx_id,State)@next:-log(Tx_id,State)).
  r3: Kc(part_cnt(count<N>):-participants(N)).
  r4: Kc(transaction(Tx_id,State):-log(Tx_id,State)).
  r5: Kcparticipants(P1).
  r6: K_Cparticipants(P2).

\\Initialization at participants
  r7: K_P(transaction(Tx_id,State)@next:-
          transaction(Tx_id,State),
          ¬del_transaction(Tx_id,State)).
  r8: K_P(log(Tx_id,State)@next:-log(Tx_id,State)).

\\Decision Phase at coordinator
  r9: K_C(yes_cnt(Tx_id,count<Part>):-
        vote(Vote,Tx_id,Part),Vote == "yes").
  r10: K_C(log(Tx_id,"commit")@next:-part_cnt(C),
        yes_cnt(Tx_id,C1),C==C1,State=="vote-req",
        transaction(Tx_id,State)).
  r11: K_C(log(Tx_id,"abort"):-vote(Vote,Tx_id,Part),
        Vote == "no",transaction(Tx_id,State),
        State =="vote-req").

\\Voting Phase at participants
  r12: K_P(log(Tx_id,"prepare"):-State=="vote-req",
       K_ctransaction(Tx_id,State)).
  r13: K_P(log("abort",Tx_id):-log(Tx_id,State),
        State=="prepare",db_status(Vote),Vote=="no").

\\Decision Phase at participants
  r14: K_P(log(Tx_id,"commit"):-log(Tx_id,State_l),
        State_l=="prepare",State_t=="commit",
        K_ctransaction(Tx_id,State_t)).
  r15: K_P(log(Tx_id,"abort"):-log(Tx_id,State_l),
        State_l=="prepare",State_t=="abort",
        K_ctransaction(Tx_id,State_t)).

\\Communication
  r16: K_Xtransaction(Tx_id, State):-K_Cparticipants(X),
       K_Ctransaction(Tx_id,State).
  r17: K_Cvote(Vote,Tx_id,"part1"):-K_P1log(Tx_id,State),
       State=="prepare",K_P1db_status(Vote).
  r18: K_Cvote(Vote,Tx_id,"part2"):-K_P2log(Tx_id,State),
       State=="prepare",K_P2db_status(Vote).
```

Listing 5: Two Phase Commit Protocol

In the above example, for simplicity we wrote K_{P} as 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 code, which are almost a faithful translation of algorithms specified in pseudocode [12, 14].

5. KNOWLOG SEMANTICS

The first step towards the definition of Knowlog's semantics is the specification of the *reified* version of Knowlog. For this purpose, we augment **dom** with a new set of constants $\triangle$ containing the modal symbols. We also assume a new set of variables O that will range over the just defined set of modal 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 built-in relations `K` and $\oplus$. A tuple over the $\mathbf{R}^{TK}$ schema will have the form $(k, s, t_1, ..., t_n)$ where $k \in O \cup \triangle$ identifies the knowledge accumulator term, $s \in S \cup \mathbb{N}$ and $t_1, ..., t_n \in \mathbf{var} \cup \mathbf{dom}$. Conversely, tuples 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., `K`$_{\text{S}}$`K`$_{\text{R}}$`input("value")`, the reified version will be `input (Y,n,"value"),Y = K`$_{\text{S}}$ $\oplus$ `K`$_{\text{R}}$ for example at time-step n. The operator $\oplus$ is hence employed to concatenate epistemic operators. Instead, in case the operator employes a variable for identifying nodes, we need to introduce in $\mathbf{R}^{TK}$ relation `K(X,Y)` in order to help us in the effort of building the knowledge accumulator term. 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 `K`$_{\text{X}}$`transaction(Tx_id,State),participants(X)` will be `K(X,Y),transaction(Y,n,Tx_id,State),participants(X)`.

What about the modal context? We already mentioned that the modal context is used to identify the node 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. For a discussion on Knowlog's operational semantics, we refer to the technical report [13].

6. RELATED WORKS

In the last few years a renew interest in Datalog is arising, especially pushed by new emerging trends such as packets routing [17], overlay networks [18], network provenance [24] and web data management [1]. Our work, in particular is motivated by [12] where the author discusses how Datalog$^\neg$ programs are interestingly suited to express logically distributed systems and their properties. Knowlog, as pointed out multiple times in the paper, is heavily based on Dedalus [5] and Statelog [19]. The notion of knowledge applied to multi-agent systems was first discussed in [11] but we take cue from the comprehensive exposition of [10].

In literature many approaches exist for enhancing logic programming with modal operators. A survey is presented in [21]. To be brief, two classes can be identified based on the employed methodology: *direct approaches* and *translation approaches*. The first directly handles modalities, while the second translates modal programs in classical logical programs. We embrace the first approach.

We found in literature just a couple of languages that share some principles with Knowlog. One is Datalog^K and is introduced in [15]. Datalog^K is basically Datalog$^\neg$ augmented with knowledge operators $\mathbf{K}$ and $\mathbf{P}$ in order to express data replication and data fragmentation among distributed databases. They define the model-theoretic, the operational semantics and a proof that these semantics coincides. Datalog^K is only able to manage static systems since no notion of time or state is used in the language. Moreover, knowledge operators are defined using a modal structure (W, V) where the set of possible worlds W contains the different sites where data can be stored, and not global states as in our case, therefore loosing the ability to express the behavior of a distributed system. Another approach similar to our, is introduced in [7] where answer set programming (ASP) is used to reason about multi-agent systems following the translational approach. On the contrary, our aim is to build a real distributed system where, exploiting the direct approach, each agent is able to directly reason on the system and then act accordingly.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented Knowlog, a programming language based on Datalog$^{\neg}$ leveraged with a notion of time and the modal operator K. Through Knowlog, reasoning about states of knowledge in distributed systems can be performed, therefore lightening 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 nodes 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 [10]. The successive step will be the definition in Knowlog of weaken forms of common knowledge such as *eventual* common knowledge. Since we are dealing with "state of knowledge", another interesting point that we are going to investigate will be the specification of how nodes "learn" [9].

8. REFERENCES

[1] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. 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, pages 293–304, New York, NY, USA, 2011. ACM.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[3] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I do declare: consensus in a logic language. *Operating Systems Review*, 43(4):25–30, 2009.

[4] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[5] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. In *Datalog2.0*, pages 262–281, 2010.

[6] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 283–292, New York, NY, USA, 2011. ACM.

[7] C. Baral, G. Gelfond, T. C. Son, and E. Pontelli. Using answer set programming to model multi-agent scenarios involving agents' knowledge about other's knowledge. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, AAMAS '10, pages 259–266, Richland, SC, 2010.

[8] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[9] K. M. Chandy and J. Misra. How processes learn. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, PODC '85, pages 204–214, New York, NY, USA, 1985. ACM.

[10] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge.* MIT Press, Cambridge, MA, USA, 2003.

[11] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 50–61, New York, NY, USA, 1984. ACM.

[12] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[13] M. Interlandi. Knowlog^{k}: A declarative language for reasoning about knowledge in distributed systems. `http://www.dbgroup.unimo.it/TechnicalReport/interlandi2012.pdf`, March 2012. Technical Report.

[14] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, May 1994.

[15] M. Levene and G. Loizou. A modal logic formalism for distributed and parallel knowledge bases. *Parallel Algorithms Appl.*, 1(1):11–27, 1993.

[16] J. Lloyd. *Foundations of logic programming.* Symbolic computation: Artificial intelligence. Springer, 1987.

[17] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 97–108, New York, NY, USA, 2006. ACM.

[18] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 75–90, New York, NY, USA, 2005. ACM.

[19] B. Ludäscher. *Integration of Active and Deductive Database Rules*, volume 45 of *DISDBIS*. Infix Verlag, St. Augustin, Germany, 1998.

[20] L. A. Nguyen. Foundations of modal deductive databases. *Fundam. Inf.*, 79:85–135, January 2007.

[21] M. A. Orgun and W. Ma. An overview of temporal and modal logic programming. In *Proceedings of the First International Conference on Temporal Logic*, ICTL '94, pages 445–479, London, UK, 1994. Springer-Verlag.

[22] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *J. Log. Program.*, 23(2):125–149, 1995.

[23] C. Zaniolo. *Advanced database systems.* Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, 1997.

[24] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 615–626, New York, NY, USA, 2010. ACM.