On Verification in Logic Database Languages

Francesco Bonchi[1,2], Fosca Giannotti[1] and Dino Pedreschi[2]

[1]CNUCE - CNR Via Alfieri 1, 56010 Ghezzano, Pisa - Italy
e-mail `Giannotti@cnuce.cnr.it`
[2]Department of Computer Science, University of Pisa C.so Italia 40, Pisa - Italy
e-mail {`bonchi,pedre`}`@di.unipi.it`

Abstract. We consider in this paper an extension of Datalog with mechanisms for non-monotonic and non-deterministic reasoning and a simple form of temporal reasoning, which we refer to as Datalog++. First, we show how with this logic database language is possible to express problems in heterogeneous domains, such as operation research and concurrent programming. Second, we provide a methodology for the verification of Datalog++ programs, based on the declarative semantics, which is able to handle both atemporal and temporal properties.

1 Introduction

The name **Datalog++** is used in this paper to refer to Datalog extended with mechanisms supporting:

- *non-monotonic* reasoning, by means of a form of stratified negation w.r.t. the stage arguments, called *XY-stratification* [17];
- *non-deterministic* reasoning, by means of the non-deterministic `choice` construct [12].
- a limited form of *temporal* reasoning, by means of temporal, or *stage*, arguments of relations, ranging over a discrete temporal domain, in the style of Datalog$_{1S}$ [6];

Datalog++, which is essentially a fragment of $\mathcal{LDL}$++ [2], and is advocated in [18, Chap. 10], revealed a highly expressive language, with applications in diverse areas such as AI planning [4], active databases [16], object databases [11], semistructured information management and Web restructuring [11].

In this paper we study a methodology for the development and the verification of Datalog++ programs. Therefore the goal of this paper is twofold: to highlight the expressiveness of Datalog++ programs by providing some realistic examples in diverse programming areas such as operative research and concurrent programming, and to provide the first steps toward a methodology for the verification of Datalog++ programs.

Research in verification in logic programming has focussed on Prolog-like, top-down languages, and relatively little work has addressed the case of deductive databases and Datalog-like languages [1]. However, Datalog-like languages have a purely declarative reading that makes reasoning about programs simpler than the Prolog-like case, and several verification methods developed for Prolog-like language can be adopted for Datalog-like languages [7]. In these methods, the task of proving a given logic program correct reduces to the task of showing that certain appropriate models of the programs coincide with the intended meaning of the program itself. Clearly, the notion of *appropriate model* is not defined once and for all, but depends on the particular language, and on its use of negation. We adhere to this view, and adopt the stable model semantics as the starting point of our investigation, as Datalog++ programs are non monotonic and employ a rather general form of negation.

Also, we are interested in using Datalog++ to specify concurrent programs, and therefore we need to reason about dynamic properties, such as safety and progress. Unlike formalisms such as UNITY [5] or other temporal logic approaches [14], which use some specialized logic to deal with dynamic properties, we show how a set of interesting safety and progress properties can be handled within the mentioned model-theoretic approach. In this sense, dynamic properties can be formalized and proved as properties of the stable models of the program under consideration.

2 Datalog++: Nondeterministic, Nonmonotonic Datalog

In this paper, we use the name Datalog++ to refer to the Datalog extended by two mechanisms for non-monotonic and non-deterministic reasoning, both based on a form of negation. The first mechanism is represented by XY-programs originally introduced in [17]. The language of such programs is $Datalog_{1S}^{\neg}$, which admits negation on body atoms and a unary constructor symbol, used to represent a temporal argument usually called the *stage argument*. A general definition of XY-programs is the following. A set P of rules defining mutually recursive predicates, is an XY-program if it satisfies the following conditions:

1. each recursive predicate has a distinguished stage argument;
2. every recursive rule r is either an X-rule or a Y-rule, where:
   - r is an X-rule when the stage argument in every recursive predicates in r is the same variable,
   - r is a Y-rule when (i) the head of r has a stage argument $J+1$, where J is a variable, (ii) some goal of r has J as its stage argument, and (iii) the remaining recursive goals have either J or $J+1$ as their stage argument.

Intuitively, in the rules of XY-programs, an atom $p(J, \_)$ denotes the extension of relation p at the current stage (present time) J, whereas an atom $p(J+1, \_)$ denotes the extension of relation p at the next stage (future time) $J+1$. By using a different *primed* predicate symbol p' in the $p(J+1, \_)$ atoms, we obtain the so-called primed version of an XY-program. We say that an XY-program is *XY-stratified* if its primed version is a stratified program. Intuitively, if the dependency graph of the primed version has no cycles through negated edges, then it is possible to obtain an ordering on the original rules modulo the stage arguments. As a consequence, an XY-stratified program is also locally stratified, and has therefore a unique perfect model. Such a model can be computed by an iterated fixpoint procedure, which uses the ordering imposed by the primed version of the program: the extension of relations at stage J is computed, before proceeding to stage $J+1$.

Example 1. The following version of the *seminaive* graph-reachability program, discussed in [17], is an example of (XY-stratified) XY-program, which computes all nodes of a graph g reachable from a given node a:

$$\begin{array}{l}
\mathtt{delta}(0,\mathtt{a}).\\
\mathtt{delta}(\mathtt{s}(\mathtt{I}),\mathtt{Y}) \leftarrow \mathtt{delta}(\mathtt{I},\mathtt{X}), \mathtt{g}(\mathtt{X},\mathtt{Y}), \neg\mathtt{all}(\mathtt{I},\mathtt{Y}).\\
\mathtt{all}(\mathtt{I},\mathtt{X}) \leftarrow \mathtt{delta}(\mathtt{I},\mathtt{X}).\\
\mathtt{all}(\mathtt{s}(\mathtt{I}),\mathtt{X}) \leftarrow \mathtt{all}(\mathtt{I},\mathtt{X}), \mathtt{delta}(\mathtt{s}(\mathtt{I}),_).
\end{array}$$

Observe the use of negation, which is not stratified in the usual sense (delta and all are mutually recursive), although it is stratified modulo the stage. In the outcome of the seminaive program, all(n, b) means that b is a node reachable from a in graph g. At each stage i, delta(i, c) means that c is a newly reached node, and i is also the length of the shortest path from a to c; using relation delta to propagate the search avoids unnecessary recomputations.

The second mechanism is represented by *choice goals*, that are used to non-deterministically select subsets of answers to queries, which obey a specified functional dependences (FD) constraint.

Example 2 (Spanning tree). The following program [9] computes the spanning tree starting from the source node a for a graph where an arc from node b to node d is represented by the database fact g(b,d).

$$\begin{array}{l}
\mathtt{st0} : \mathtt{st}(\mathtt{root},\mathtt{a}).\\
\mathtt{st1} : \mathtt{st}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{st}(_,\mathtt{X}), \mathtt{g}(\mathtt{X},\mathtt{Y}), \mathtt{Y} \neq \mathtt{a}, \mathtt{choice}((\mathtt{Y}),(\mathtt{X})).
\end{array}$$

Observe that, when an arc st(b,d) is added by the second rule, the choice goal ensures that no st(X,d) can later be added with $X \neq b$. The condition $X \neq a$ must be added explicitly to prevent it from generating an arc leading back to a because there is no choice goal in the first rule to avoid it.

The semantics of Choice construct is based on the *stable model* semantics of Datalog¬ programs, a concept originating from autoepistemic logic, which was applied to the study of negation in Horn clause languages in [8]. To define the notion of a stable model we need to introduce a transformation H which, given an interpretation I, maps a Datalog¬ program P into a positive Datalog program $H(P,I)$:

$$\begin{aligned}H(P,I) = \{A \leftarrow B_1,\ldots,B_n \mid\ & A \leftarrow B_1,\ldots,B_n,\neg C_1,\ldots,\neg C_m \in ground(P)\\ & \wedge \{C_1,\ldots,C_m\} \cap I = \emptyset\}\end{aligned}$$

Next, we define:

$$S_P(I) = T_{H(P,I)} \uparrow \omega$$

Then, M is said to be a *stable model* of P if $S_P(M) = M$. In general, Datalog¬ programs may have zero, one or many stable models. The multiplicity of stable models can be exploited to give a declarative account of non determinism. We can in fact define the *stable version* of a program P, $SV(P)$, to be the program transformation where all the references to the *choice* atom in a rule $r : H \leftarrow B, choice(X, Y)$ are replaced by the atom $chosen_r(X, Y)$, and define the $chosen_r$ predicate with the following rules:

$$chosen_r(X, Y) \leftarrow B, \neg diffchoice_r(X, Y).$$
$$diffchoice_r(X, Y) \leftarrow chosen(X, Y'), Y \neq Y'.$$

The various stable-models of the transformed program $SV(P)$ thus correspond to the choice models of the original program [15]. For instance, the various stable models of the program of example 2 correspond to the various spanning tree for graph `g`. Choice programs can be efficiently executed by enforcing the FD constraints during the bottom-up computation [12]. Overall semantics of Datalog++, efficient implementation and optimizations are presented in [10].

3 Programming with Datalog++

Although conceived as a deductive database language, Datalog++ revealed a highly expressive language with applications in other domains. The goal of this section is to highlight the expressiveness of Datalog++ programs by providing some examples in diverse programming areas, ranging from typical procedural algorithms to operation research and concurrent programming.

3.1 Euclid's Algorithm for the Greatest Common Divisor

Our first example illustrates a general programming paradigm, where stages used to record various steps in the computation, `choice` is used to non-deterministically select actions at each step, and rules for the *frame axioms* or *inertia axioms* are used to establish the content of a new state. This technique provides a general programming paradigm that has been applied to many problems, such as planning problem [4] and array ordering problem [9].

The following program computes according to Euclid's algorithm, the greatest common divisor of a set of integers stored as facts of the form `number(X)` in the database.

$$\begin{array}{lll}
\mathtt{r0 : candidate_GCD(0,X)} & \leftarrow & \mathtt{number(X).} \\
\mathtt{r1 : aux_GCD(J,X,Y)} & \leftarrow & \mathtt{candidate_GCD(J,X), candidate_GCD(J,Y),} \\
 & & \mathtt{X > Y, choice((J),(X,Y)).} \\
\mathtt{r2 : candidate_GCD(J+1,X)} & \leftarrow & \mathtt{candidate_GCD(J,X), \neg aux_GCD(J,X,Y).} \\
\mathtt{r3 : candidate_GCD(J+1,Z)} & \leftarrow & \mathtt{aux_GCD(J,X,Y), Z = X - Y.} \\
\mathtt{r4 : GCD(X)} & \leftarrow & \mathtt{candidate_GCD(J,X), \neg aux_GCD(J,_,_).}
\end{array}$$

The first rule initializes the staged `candidate_GCD` predicate: at the beginning (stage 0) all integers in the database are candidate to be the greatest common divisor. The second rule nondeterministically selects a pair of candidate integers with different values. Note the use of `choice` to ensure that, at each stage, only one pair of integers out of all the eligible pairs is selected. The third and the fourth rules realize frame axioms: old candidate integers (excluded the greater integer of the selected pair) are copied into the set of candidate integers for the next stage, while a new value, the difference between the two integers of the selected pair, is added. The last rule, according to Euclid's algorithm, simply asserts that when it is no longer possible to select a pair of candidate integers with different values, i.e. when all the candidate integers have the same value, then greatest common divisor has been found.

The objective of this example is to show how it is possible to express typical procedural algorithms in a declarative way. The program we have written is as simple and efficient as its procedural version is. Also, observe that this program computes the greatest common divisor for any set of integers, independently of its cardinality.

3.2 The Ford-Fulkerson Method for the Maximum-Flow Problem

The next program is another example of a typical procedural algorithm in operation research, it is harder than the previous one, but as we show, Datalog++ is naturally geared to cope with graph algorithms. We first introduce some notation.

A *flow-network* $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$. If $(u, v) \notin E$ we assume that $c(u, v) = 0$. We distinguish two vertex in a flow network: a *source* s and a *sink* t. For convenience we assume that every vertex lies on some path from the source to the sink. Let $G = (V, E)$ be a flow-network (with an implied capacity function c). Let s be the source of the network, and let t be the sink. A *flow* in G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the following three properties:

Capacity constraint For all $u, v \in V$, we require $f(u, v) \leq c(u, v)$.
Skew symmetry For all $u, v \in V$, we require $f(u, v) = -f(v, u)$.
Flow conservation For all $u \in V - \{s, t\}$, we require $\sum_{v \in V} f(u, v) = 0$.

The quantity $f(u, v)$, which can be positive or negative, is called the net flow from vertex u to vertex v.

The Ford-Fulkerson method solves the maximum-flow problem. This method is iterative. We start with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value by finding an *augmenting path*, which we can think of as a path from the source s to the sink t along which we can push more flow. We repeat this process until no augmenting path can be found; at that moment the flow has reached its maximum value.

We now show how to realize this algorithm with Datalog++. We are given a directed graph, describing the flow network, whose arcs are stored as facts in the database. For each edge (x, y) with capacity c we have a fact g(x,y,c) in the database. We also have a fact g(y,x,c') where $c' = 0$ in the case that does not exist the edge (y, x). The main predicate is flow(J,X,Y,F), describing the value F of the flow from vertex X to vertex Y at stage J. First of all we need a flow initializing rule:

$$\texttt{flow}(0,\texttt{X},\texttt{Y},0) \leftarrow \texttt{g}(\texttt{X},\texttt{Y},\_).$$

Then we need a predicate describing the *residual capacity network*, that is the graph on which we search a path from the source to the sink along which we can push more flow with respect to the capacity constraint.

$$\begin{aligned}\texttt{residual\_network}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}) \leftarrow\ & \texttt{g}(\texttt{X},\texttt{Y},\texttt{C1}), \texttt{flow}(\texttt{J},\texttt{X},\texttt{Y},\texttt{F}),\\ & \texttt{C} = \texttt{C1} - \texttt{F1}, \texttt{C} > 0.\end{aligned}$$

To the purpose of finding a path from the source to the sink on this graph we use the *spanning tree* program, but each time a new edge is added to the spanning three, its residual capacity is compared to the minimum residual capacity of the edges that lie on that path. Therefore, when we reach the sink we already know the minimum residual capacity on the augmenting path, that is the maximum value of flow we can push along the augmenting path.

$$\begin{aligned}\texttt{augmenting\_path}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}) \leftarrow\ & \texttt{residual\_network}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}),\\ & \texttt{X} = \texttt{source}, \texttt{choice}((\texttt{J},\texttt{Y}),(\texttt{X})).\\ \texttt{augmenting\_path}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}) \leftarrow\ & \texttt{augmenting\_path}(\texttt{J},\_,\texttt{X},\texttt{C}),\\ & \texttt{residual\_network}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C1}),\\ & \texttt{Y} \neq \texttt{source}, \texttt{C} < \texttt{C1}, \texttt{choice}((\texttt{J},\texttt{Y}),(\texttt{X})).\\ \texttt{augmenting\_path}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}) \leftarrow\ & \texttt{augmenting\_path}(\texttt{J},\_,\texttt{X},\texttt{C1}),\\ & \texttt{residual\_network}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}),\\ & \texttt{Y} \neq \texttt{source}, \texttt{C} < \texttt{C1}, \texttt{choice}((\texttt{J},\texttt{Y}),(\texttt{X})).\end{aligned}$$

In this way we compute a spanning tree starting from the source. However what we need is a path from the source to the sink. Therefore we have to spot those edges lying on the augmenting path, i.e. the edges on which we push more flow. To do this we can go backwards from the sink to the source.

$$\begin{aligned}\texttt{flow\_augmentation}(\texttt{J},\texttt{X},\texttt{sink},\texttt{C}) \leftarrow\ & \texttt{augmenting\_path}(\texttt{J},\texttt{X},\texttt{sink},\texttt{C}).\\ \texttt{flow\_augmentation}(\texttt{J},\texttt{Y},\texttt{X},\texttt{C}) \leftarrow\ & \texttt{flow\_augmentation}(\texttt{J},\texttt{X},\_,\texttt{C}),\\ & \texttt{augmenting\_path}(\texttt{J},\texttt{Y},\texttt{X},\_).\end{aligned}$$

Finally, we need some rules to update, according to the algorithm, the flow at each new stage and a final rule to detect the maximum flow when it is no longer possible to find an augmenting path.

$$\texttt{flow}(\texttt{J}+1,\texttt{X},\texttt{Y},\texttt{F}) \leftarrow \texttt{flow}(\texttt{J},\texttt{X},\texttt{Y},\texttt{F1}),\texttt{flow\_augmentation}(\texttt{J},\texttt{X},\texttt{Y},\texttt{C}), \texttt{F} = \texttt{F1} + \texttt{C}.$$

$$\texttt{flow}(\texttt{J}+1,\texttt{X},\texttt{Y},\texttt{F}) \leftarrow \texttt{flow}(\texttt{J},\texttt{X},\texttt{Y},\texttt{F1}),\texttt{flow\_augmentation}(\texttt{J},\texttt{Y},\texttt{X},\texttt{C}), \texttt{F} = \texttt{F1} - \texttt{C}.$$

$$\texttt{flow}(\texttt{J}+1,\texttt{X},\texttt{Y},\texttt{F}) \leftarrow \neg\texttt{flow\_augmentation}(\texttt{J},\texttt{X},\texttt{Y},\_), \neg\texttt{flow\_augmentation}(\texttt{J},\texttt{Y},\texttt{X},\_).$$

$$\texttt{max\_flow} \leftarrow \texttt{flow}(\texttt{J},\texttt{X},\texttt{Y},\texttt{F}),\neg\texttt{augmenting\_path}(\texttt{J},\_,\texttt{sink},\_).$$

3.3 The Readers-Writers Problem

In this last example we show how Datalog++ is able to specify typical concurrent programming problems. We are given a shared resource and some processes. The processes have access to the resource for read or write operations. Any number of reads can proceed concurrently, but a write cannot be executed concurrently with a read or another write. There is no particular priority policy: at each stage it is nondeterministically selected whether to give priority to read or write operations. A process waiting for the right to execute an operation remains in its wait-state until it gets that right and completes its operation. Finally, we assume that all reads and writes complete in a stage. For each process we have in the database a fact of the form `process(X)`, where `X` is the name of the process.

```
want_to(want_to_read).      action(reading).
want_to(want_to_write).     action(writing).

p0 : next_priority(0,X) ← action(X), choice((),(X)).

p1 : aux_priority(J,X) ← next_priority(J,_), action(X), choice((J),(X)).

p2 : next_priority(J+1,X) ← aux_priority(J,X).

s0 : state(0,K,X) ← process(K), want_to(X), choice((K),(X)).

n0 : next_state(J,K,writing) ← state(J,K,want_to_write),
                               next_priority(J,writing),
                               choice((J),(K)).

n1 : next_state(J,K,writing) ← state(J,K,want_to_write),
                               next_priority(J,reading),
                               ¬state(J,_,want_to_read),
                               choice((J),(K)).

n2 : next_state(J,K,reading) ← state(J,K,want_to_read),
                               next_priority(J,reading).

n3 : next_state(J,K,reading) ← state(J,K,want_to_read),
                               next_priority(J,writing),
                               ¬state(J,_,want_to_write).
n4 : next_state(J,K,X)       ← state(J,K,Y), action(Y),
                               want_to(X), choice((J,K),(X)).
```

$$\mathtt{s1} : \mathtt{state}(\mathtt{J}+1, \mathtt{K}, \mathtt{X}) \leftarrow \mathtt{next\_state}(\mathtt{J}, \mathtt{K}, \mathtt{X}).$$

$$\begin{aligned}\mathtt{s2} : \mathtt{state}(\mathtt{J}+1, \mathtt{K}, \mathtt{X}) \leftarrow\ & \mathtt{state}(\mathtt{J}, \mathtt{K}, \mathtt{X}), \mathtt{want\_to}(\mathtt{X}),\\ & \neg\mathtt{next\_state}(\mathtt{J}, \mathtt{K}, \_).\end{aligned}$$

The first three rules nondeterministically assign the priority, at each stage, to the read or to the write operations. The rule `s0` nondeterministically selects a starting state for each process. When the write operations take priority, the rule `n0` gives the right to reach the resource to a process waiting for a write operation. Note the use of `choice` to model *mutual exclusion*: only one process, nondeterministically chosen out of all the waiting-for-writing processes, will be able to write at next stage. When the read operations take priority but there are not any processes waiting for reading, the rule `n1` acts just like the rule `n0`. The rules `n2` and `n3` correspond respectively to the rule `n0` and `n1` but for the processes that want to read. Note that in this case we do not need the `choice` because any number of reads can be executed concurrently.

Another example of Datalog++ program in the concurrent programming area is the *Alternating Bit Protocol* in [3].

4 Verification of Datalog++ Programs

The goal of this section is that of suggesting a methodology for the verification of Datalog++ programs. Our methodology is based on the distinction between *atemporal* and *temporal* predicates. In fact, for temporal predicates we prove correctness by showing that their stable models match their intended semantics. On the other hand, to show the correctness of temporal predicates we define and prove some standard properties based on their temporal structure.

Definition 3 (X and Y predicates). Let P be an XY-program, and let `p` be a predicate of P. We call `p` an *X-predicate* if the rules defining `p` are only X-rules; an *Y-predicate* if at least one rule defining `p` is a Y-rule.

We assimilate to X-predicates also the predicates of non-staged programs. Intuitively using X-predicates we can deduce only facts for the present time (actual stage), while using Y-predicates we can deduce facts for the future time (next stage). It is important to note that all the programs of the previous section share a similar structure described in Figure 1 for the Ford-Fulkerson example. In fact, Datalog++ programs usually have a single Y-predicate which performs stage transition, and many X-predicates which prepare such transition.

4.1 Properties of the `choice` Construct

When `choice` constructs are used in XY-programs we must pay attention to the *choice safety* constraint, which requests the stage argument to appear in the domain of the FD (the right argument of a choice goal). Moreover we restrict the use of `choice` at the X-rules, as we have done in all the programs of the previous

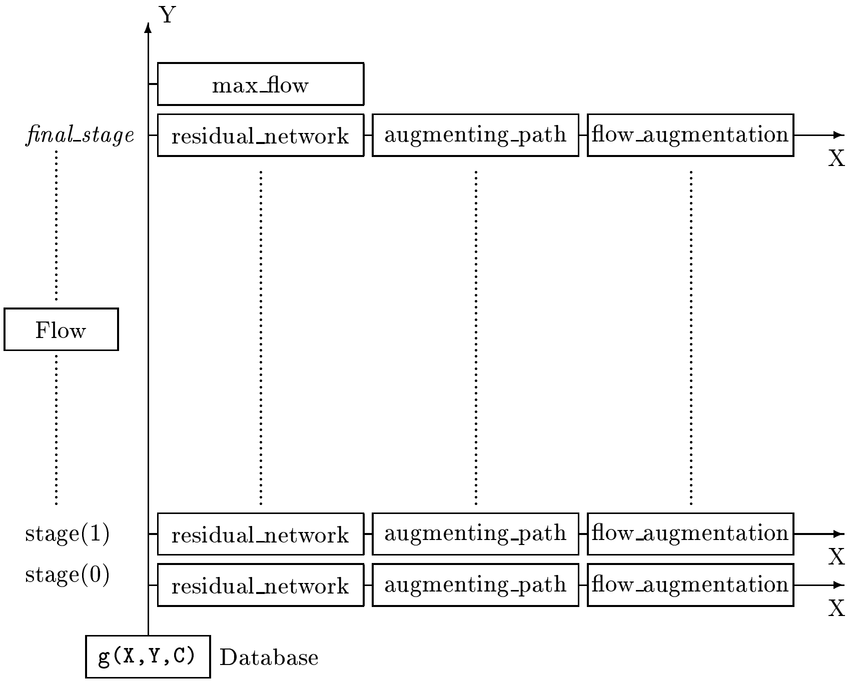

Fig. 1. Structure of the Ford-Fulkerson program

sections. This syntactic restriction, however, does not affect the expressiveness of Datalog++ and makes verification easier.

Now we define a *general choice rule*:

Definition 4 (General `choice` Rule). We define a general `choice` as a rule with the following structure:

$$\mathtt{p}(\mathtt{J}, \alpha, \epsilon) \leftarrow \mathtt{q}(\mathtt{J}, \beta), \mathtt{choice}((\mathtt{J}, \delta), (\epsilon)).$$

where Greek letters are sets of variables, $\mathtt{q}(\mathtt{J}, \beta)$ stands for the conjunction of all the non-choice goals (also the non-staged ones) of the rule, α and δ can be empty sets, $\delta \cap \epsilon = \emptyset$, $\delta, \epsilon \subseteq \beta$.

The stable version of this general `choice` rule is:

$$\begin{aligned}
\mathtt{sv0} : \mathtt{p}(\mathtt{J}, \alpha, \epsilon) &\leftarrow \mathtt{q}(\mathtt{J}, \beta), \mathtt{chosen}(\mathtt{J}, \delta, \epsilon). \\
\mathtt{sv1} : \mathtt{chosen}(\mathtt{J}, \delta, \epsilon) &\leftarrow \mathtt{q}(\mathtt{J}, \beta), \neg\mathtt{diffChoice}(\mathtt{J}, \delta, \epsilon). \\
\mathtt{sv2} : \mathtt{diffChoice}(\mathtt{J}, \delta, \epsilon) &\leftarrow \mathtt{chosen}(\mathtt{J}, \delta, \epsilon'), \epsilon \neq \epsilon'.
\end{aligned}$$

where ϵ' is derived from ϵ by replacing each $A \in \epsilon$ with a new variable A'.

Following property clarifies what kind of functional dependency is enforced by the choice construct in each stable model.

Proposition 5. *Let* $\mathtt{p}(\mathtt{J}, \alpha, \epsilon) \leftarrow \mathtt{q}(\mathtt{J}, \beta), \mathtt{choice}((\mathtt{J}, \delta), (\epsilon))$ *be a general choice rule of a program* P*, and let* M *be a stable model of the stable version of* P*. Then for any integer* $\mathtt{J} \geq 0$ *and for any* α*, there's at most one atom in* M *ground instance of* $\mathtt{p}(\mathtt{J}, \alpha, \epsilon)$*.*

Proof. Suppose that for some $\mathtt{J} \geq 0$ there are in M two atoms $\mathtt{p}(\mathtt{J}, \mathtt{a}, \epsilon 1)$ and $\mathtt{p}(\mathtt{J}, \mathtt{a}, \epsilon 2)$ ground instances of $\mathtt{p}(\mathtt{J}, \mathtt{a}, \epsilon)$, with $\epsilon 1 \neq \epsilon 2$. Since all stable models are supported models [13], then by clause `sv0`, also $\mathtt{chosen}(\mathtt{J}, \delta, \epsilon 1)$ and $\mathtt{chosen}(\mathtt{J}, \delta, \epsilon 1)$ are in M. Hence, by clause `sv1` and since M is supported we have that $\mathtt{diffChoice}(\mathtt{J}, \delta, \epsilon 1)$ and $\mathtt{diffChoice}(\mathtt{J}, \delta, \epsilon 2)$ are not in M. Since $\epsilon 1 \neq \epsilon 2$, it follows from clause `sv2` that either $\mathtt{chosen}(\mathtt{J}, \delta, \epsilon 1)$ or $\mathtt{chosen}(\mathtt{J}, \delta, \epsilon 2)$ is not in M which is a contradiction.

□

4.2 Verification of X-predicates

We are now ready to introduce an overview of the strategy which we follow to prove the correctness of X-predicates.

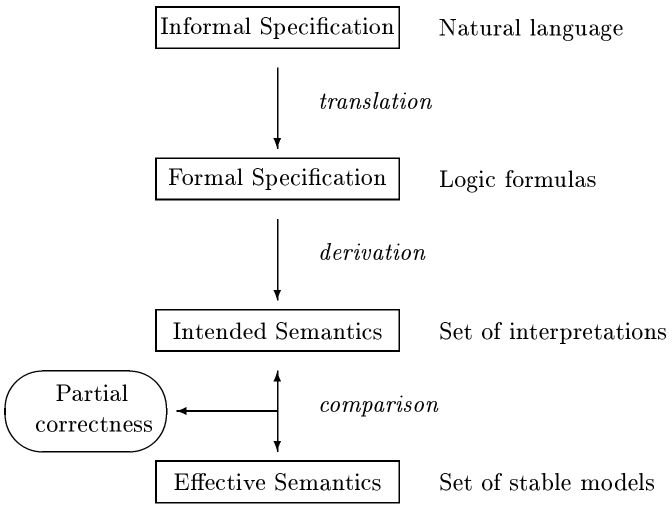

Fig. 2. Our strategy to prove X-predicates partial correctness

We begin by describing in natural language which role in the computation is played by the X-predicate. This description is the *informal specification* of the predicate. The second step consists of translating this description from natural language in a set of logic rules, obtaining the *formal specification*. From this formal specification we derive the *intended semantics* of the predicate. Since the effective semantics of a Datalog++ program is a set of stable models, then, also its intended semantics will be a set of interpretations. Exactly, the set of interpretations which satisfy the logic rules of the formal description.

Definition 6 (Intended Semantics of an X-predicate). Let `p` be an X-predicate, and let S be a set of logic rules defining the formal specification of `p`. The intended semantics of `p` is the set of interpretations that satisfy each rule in S.

$$IS(\mathtt{p}) = \{I | I \in B_{\mathtt{p}} \wedge (\forall r \in S. I \models r)\}$$

We can now define *partial correctness* of an X-predicate.

Definition 7 (Partial correctness of an X-predicate). An X-predicate is said to be partially correct with respect to its intended semantics $IS(\mathtt{p})$ iff, for each stable model M of `p`, there exists an interpretation $I \in IS(\mathtt{p})$ such that $M \subseteq I$.

Example 8 (Partial Correctness of Spanning tree). Consider the *spanning tree* program of example 2 and assume we wish to prove its correctness. First of all we have to give its informal specification in natural language.

Informal Specification: the `st` relation is a maximal subset of the `g` relation. Moreover, `st` is a tree rooted in `a`.

Now we have to translate this informal specification in a set of logic formulas. To be a tree means:

$$st(X, Y) \rightarrow st(\_, X) \vee Y = a. \tag{1}$$

$$st(X, Y) \rightarrow \neg(st(Z, Y) \wedge Z \neq X). \tag{2}$$

To be rooted in `a` means:

$$st(root, a). \tag{3}$$

To be a maximal subset of `g` means:

$$st(X, Y) \rightarrow g(X, Y) \vee Y = a. \tag{4}$$

$$(g(X, Y) \wedge st(\_, X)) \rightarrow st(\_, Y). \tag{5}$$

Intended Semantics: $IS(\mathtt{st}) = \{I | I \subseteq B_{\mathtt{st}} \wedge I \models (1), (2), (3), (4), (5)\}$.

Partial correctness: Let M be a stable model of `st`. Then $\exists I \in IS(\mathtt{st})$ such that $M \subseteq I$.

Proof. Property (1) is trivially satisfied by clause `st0` and by the definition of model; (2) is satisfied by clause `st0`, by the goal `st(_,X)` in the body of clause `st1` and since all stable models are supported; (3) is satisfied by the goal $\mathtt{Y} \neq \mathtt{a}$ and, for proposition 5, by the choice-goal in the body of clause `st1`; (4) is satisfied by clause `st0`, by the goal `g(X,Y)` in the body of clause `st1` and since all stable models are supported; (5) is satisfied because if $(\mathtt{g(X,Y)} \wedge \mathtt{st(\_,X)})$, then each non-choice goal in the body of clause `st1` is satisfied, and then `choice` has to choose.

□

Note how the translation in logic formulas of the informal specification is quite close to the text of the program. This facts hints us a way to exploit the declarative features of Datalog++ programming to give a methodology for program development. Suppose, in fact, we have not yet written the *Spanning tree* program and we wish to develop such program. First we write the informal specification. Then we translate it in logic formulas. At this point the program development become immediate. The logic property (3) trivially gives us the `st0` clause. The properties (1) and (4) suggest us the non-choice goal of clause `st1`. Finally, provided that we recognize in (2) a functional dependence, typically imposed by the `choice` construct, we obtain the program as we have written before.

4.3 *Safety* and *Progress* Properties for Y-predicates

For Y-predicates we do not prove partial correctness by reasoning on models, as we have done for X-predicates. We define and prove some standard properties following the approach of UNITY, a computational model with associated a proof system, introduced by Chandy and Misra in [5]. According to Chandy and Misra, we distinguish between *safety* and *progress* properties. We do not define these terms formally. Intuitively safety properties are invariant of the programs, i.e. properties which are satisfied at each stage of the computation. On the other hand, progress properties regard the temporal development of the computation.

We start introducing two kind of safety properties, one characterizing Y-predicates from a semantics point of view, the other one from a quantitative point of view.

The *Meaning* properties express, at each stage, the matching of the role played by the Y-predicate $\mathtt{p}$ in the computation with its declarative reading $dr(\mathtt{p})$, i.e. the informal specification.

Definition 9 (Final stage). Let $\mathtt{p}$ be a Y-predicate of a program P and M a stable model of P. If for each stage $J : 0 \leq J \leq K$ exists at least one atom of the form `p(J,_)` in M, while it does not exist for $J > K$, then we say K to be the *final_stage* of $\mathtt{p}$ and we write $K = final\_stage(\mathtt{p})$. If such a stage does not exist then $final\_stage(\mathtt{p}) = +\infty$.

Definition 10 (*Meaning*). A *Meaning* property is the matching of an Y-predicate with its declarative reading. Such a property has the following form:

$$\forall J : 0 \leq J \leq final\_stage(\mathtt{p}), \forall \alpha : db(\alpha)\ .\ \mathtt{p}(J, \delta) \equiv dr(\mathtt{p})$$

where α and δ are set of variables such that $\alpha \subseteq \delta$, and db is a relation in the database.

Such kind of properties can be easily proved by induction on the stages as we will show later by providing some examples.

The *HowMany* properties constrain the number of atoms of a given form in every stable model of the program.

Example 11. Recall the program for the *readers-writers problem*. We would like our program to have some desired properties:

1. each process at each stage is in one, and only one, state,
2. two or more write operations can not be executed concurrently,
3. write operations can not be executed concurrently with read operations.

We can express these three properties in terms of number of occurrences of facts of a given form in the stable models:

1. $\forall J, \forall K : process(K) \quad \#\{X | state(J, K, X)\} = 1$
2. $\forall J \quad \#\{K | state(J, K, writing)\} \leq 1$
3. $\forall J \quad \#\{K, W | state(J, K, writing) \wedge state(J, K, reading)\} = 0$

In other words *HowMany* properties are specified using the count aggregate in the stable models or their approximations.

Definition 12 (*HowMany*). Let $\Pi(J, \kappa, \delta, cost)$ be a logic formula formed only by Y-predicates, where J is the stage argument of the Y-predicates, κ is the set of variables on which we count, δ is the set of all the other variables, $cost$ are he constants. We define *HowMany* property the following expression:

$\forall J, K \quad \#\{\kappa | \Pi(J, \kappa, \delta, cost)\} < op > n$

where n is a nonnegative integer, and $< op > \in \{=, \geq, \leq\}$.

We next define two kinds of progress property, the first one for those programs which reach a fixed point, the second one for those programs that do not terminate (such as the *readers-writers problem*).

Definition 13 (*FixedPoint*). Let p be an Y-predicate. We define *FixedPoint* property the following:

$\exists j < +\infty : \; j = final\_stage(\mathtt{p})$.

A useful strategy for showing that a Y-predicate always reaches its final_stage is the following: display a function that strictly decreases at each stage and that it's lower bounded, and map the function to the Y-predicate.

Note that in each program of the previous section we have just one Y-predicate which gives the vertical structure to the program and some X-predicate doing the horizontal work. Therefore, if all X-predicates terminate, when we prove that the unique Y-predicate has a finite final_stage, we have that no new atom can be deduced after the final stage, i.e. the computation has reached a fixed point.

The next kind of properties state that the presence of a given fact in a stable model *ensures* that another fact sooner or later will belong to the approximations of stable model.

Definition 14 (*Ensures*). Let p and q be Y-predicates of a program P. An *Ensures* property is a statement of the form

$\forall \alpha, \; \mathtt{p}(\mathtt{J}, \alpha) \;\; \mathtt{ensures} \;\; \exists \mathtt{I} > \mathtt{J} : \mathtt{q}(\mathtt{I}, \beta)$.

Verification of these kind of properties requires a *fairness assumption* which is not treated here due to lack of space. Examples can be found in [3].

4.4 Application of the Methodology

We now apply the methodology introduced to the GCD program. Verification of the Ford-Fulkerson and other programs can be found in [3].

Example 15 (Properties of the GCD program). Recall the *GCD* program. We wish to prove the following properties:

1. *Meaning:* $\forall J : 0 \leq J \leq final\_stage(\texttt{candidate\_GCD}).\texttt{candidate\_GCD}(\texttt{J},\texttt{X}) \equiv$ "`X` is a nonnegative integer which if inserted in the initial set of number doesn't change the GCD."
2. *FixedPoint:* $\exists j < +\infty : j = final\_stage(\texttt{candidate\_GCD})$.

Proof. We prove, by induction on the stages, the *Meaning* property:

Base case (J=0) Trivial. Since only by rule `r0` we can deduce facts of the form `candidate_GCD(0,X)`, then `X` belongs to the initial set of integers.

Induction step We can deduce facts of the form `candidate_GCD(n+1,X)` by rules `r2` and `r3`. If `candidate_GCD(n+1,X)` is deduced by the rule `r2` then by induction hypothesis `X` satisfies the property. If `candidate_GCD(n+1,X)` is deduced by the rule `r3` then `X` is the difference between two integers `(a,b)` such that `candidate_GCD(n,a)` and `candidate_GCD(n,b)`. Since $GCD(a,b) = GCD(a-b,b)$ if $a > b$, and since `X = a - b` then `X` satisfies the property.

We now prove that `candidate_GCD` reaches its final stage and thus the program terminates. In fact, at each stage we delete from the set of candidates an integer and we add another integer which is less than the previous one. Therefore, at each stage the sum total of the candidate integers strictly decreases but the sum total is lower bounded. Thus the program terminates. In particular, the program terminates when it is no longer possible to find two candidate_integers with differet values. By the *Meaning* property 1, and since $GCD(a,a) = a$, then, when the program reachs the fixed point we find the GCD of the initial set of integers, as stated by rule `r4`.

5 Conclusion and future work

We briefly presented in this paper a methodology for developing Datalog++ programs and reasoning on their correctness. We believe that Datalog++ is a versatile programming language which supports verification principles, firmly coated in logic. Future research includes a thorough formalization of the verification methodology, as well as the development of tools, automatic or semi-automatic, which support verification of Datalog++ programs.

References

1. K. R. Apt. Program verification and Prolog. In E. Borger, editor, *Specification and Validation methods for Programming languages and systems.* Oxford University press, 1994.
2. N. Arni, K. Ong, S. Tsur, and C. Zaniolo. $\mathcal{LDL}$++: A Second Generation Deductive Databases Systems. Technical report, MCC Corporation, 1993.
3. F. Bonchi. Verification of Datalog++ Programs (in Italian). Master's thesis, Department of Computer Science University of Pisa, 1998.
4. A. Brogi, V. S. Subrahmanian, and C. Zaniolo. The Logic of Totally and Partially Ordered Plans: a Deductive Database Approach. *Annals of Mathematics in Artificial Intelligence*, 19:59–96, 1997.
5. K.M. Chandy and J.Misra. *Parrallel Program Design: A Foundation.* Addison-Wesley, 1988.
6. J. Chomicki. Temporal deductive databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design and Implementation*, pages 294–320. Benjamin Cummings, 1993.
7. P. Derensart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.
8. M Gelfond and V. Lifchitz. The Stable Model Semantics for logic programming. In *Proc. of the 5th Int. Conf. on Logic Programming*, pages 1070–1080, 1988.
9. F. Giannotti, S. Greco, D. Saccà, and C. Zaniolo. Programming with non Determinism in Deductive Databases. *Annals of Mathematics in Artificial Intelligence*, 19:97–125, 1997.
10. F. Giannotti, G. Manco, M. Nanni, D. Pedreschi, and F. Turini. Nondeterministic, nonmonotonic logic databases. *To appear in IEEE Transaction on Knowledge and Data Engineering*, 2000.
11. F. Giannotti, G. Manco, and D. Pedreschi. A Deductive Data Model for Representing and Querying Semistructured Data. In *Proc. 5th Int. Conf. on Deductive and Object-Oriented Databases (DOOD97)*, December 1997.
12. F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo. Non-Determinism in Deductive Databases. In *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases (DOOD91)*, volume 566 of *Lecture Notes in Computer Science*, pages 129–146, 1991.
13. V. Marek and V.S. Subrahmanian. The relation between stable, supported, default and auto-epistemic semantics for general logic programs. *Theoretical Computer Science*, 103:365–386, 1992.
14. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18 th. IEEE Symposium on Foundations of Computer Science*, pages 44–57, 1977.
15. D. Saccà and C. Zaniolo. Stable Models and Non-determinism in Logic Programs with Negation. In *Proc. of the ACM Symp. on Principles of Database Systems.*
16. C. Zaniolo. Active Database Rules with Transaction Conscious Stable Model Semantics. In *Proc. 4th Int. Conf. on Deductive and Object-Oriented Databases (DOOD95)*, volume 1013 of *Lecture Notes in Computer Science*, pages 55–72, 1995.
17. C. Zaniolo, N. Arni, and K. Ong. Negation and Aggregates in Recursive Rules: The $\mathcal{LDL}$++ Approach. In *Proc. 3rd Int. Conf. on Deductive and Object-Oriented Databases (DOOD93)*, volume 760 of *Lecture Notes in Computer Science*, 1993.
18. C. Zaniolo, S. Ceri, C. Faloutsos, R.T Snodgrass, V.S. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufman, 1997.