Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 216 (2008) 79–92
www.elsevier.com/locate/entcs

A New Proposal for Debugging Datalog Programs[1]

R. Caballero[a,2] Y. García-Ruiz[a,3] F. Sáenz-Pérez[b,4]

[a] *Departamento de Sistemas Informáticos y Computación*
Universidad Complutense de Madrid
Madrid, Spain

[b] *Departamento de Ingeniería del Software e Inteligencia Artificial*
Universidad Complutense de Madrid
Madrid, Spain

Abstract

In this paper, we propose to apply declarative debugging to Datalog programs. Our approach relies on program semantics rather than on the computation mechanism. The debugging process starts when the user detects an unexpected answer. By asking questions about the intended semantics, the debugger looks for incorrect program relations. While usual declarative debuggers for other languages are based on computation trees, we show that graphs are more convenient structures for representing Datalog computations. The theoretical framework is complemented by the implementation of a debugger for the deductive database system DES, a publicly available open-source project.

Keywords: Declarative Debugging, Datalog Programs.

1 Introduction

The declarative programming paradigm is targeted to raise the semantic level of programs, therefore isolating them from the computation model. Thus, programmers are intended to focus on a higher semantic level rather than on the level corresponding to the underlying computation procedures.

Deductive database languages such as Datalog [9], which inherit the declarative nature of the Logic Programming language Prolog [15], increase the gap between the program semantics and the computations because the computation model of Datalog is much more intricate than that of Prolog. The Prolog computation model is

[1] This work has been funded by the projects TIN2005-09207-C03-03 and S-0505/TIC/0407.

[2] Email: rafa@sip.ucm.es

[3] Email: ygarciar@fdi.ucm.es

[4] Email: fernan@sip.ucm.es

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2008.06.035

based on the SLD resolution principle [7], which deals with SLD computation trees, whereas Datalog computation model is based on a number of proposals, ranging from interpreters [17] to compilation to Prolog using magic sets [3]. The first proposal needs some sort of fixpoint computations to solve queries, which is not the computation model the user might have in mind. The second proposal makes things even worse, in the sense that the program is transformed before applying SLD resolution. This semantic gap between program semantics and program execution makes debugging Datalog programs a hard task if one tries to use existing tools for debugging in a quite different level the user thinks about (for instance, using a trace debugger in the level of the transformed program).

Our approach to debug Datalog programs is anchored to the semantic level, which is a natural requirement every user imposes to development systems. We propose a novel way of applying *declarative debugging*, also called Algorithmic Debugging (a term first coined in the logic programming field by E.H. Shapiro [12]) to Datalog programs, allowing to debug queries and diagnose *missing* (an expected tuple is not computed) as well as *wrong* (a given computed tuple should not be computed) answers with the same tool.

What a Datalog programmer would find useful is to catch program rules or relations which are responsible for a mismatch between the intended semantics of a query and its actual computed semantics. Our system, by means of a question-answering procedure which starts when the user detects an unexpected answer for some query, looks for those errors pointing to the program fragment responsible for the incorrectness. For this procedure, we propose to use *computation graphs* as a novel data structure for declarative debugging of Datalog programs. We find that these graphs are more convenient for modeling program computations, instead of computation trees, which have been typically used in declarative debuggers for other languages (e.g., Prolog [12], Java [5] and $\mathcal{T}oy$ [4]).

With this aim we have implemented a working prototype for DES [11], a Datalog system publicly available as an open-source project which was released in 2004, and it has been mainly used in several universities for teaching deductive databases since then. The current version with debugging capabilities can be downloaded and tested on almost any platform.

The few existing proposals for debugging Datalog programs are usually based on "imperative" debugging, that try to follow the computation model to find bugs. These proposals are mainly based on forests of proof trees [2,18,14], which makes debugging a trace-based task not so amenable to users. To our knowledge, the very first work on this setting is due to [10], but a variant of SLD resolution is used by the user to look for program errors, therefore imposing to traverse at least as many trees as particular answers are obtained for any query. In a database framework, where the answer can contain many individual values, this makes the task of debugging quite cumbersome. In our setting, we deal with the set of values for a query as a single entity, therefore reducing the complexity of the debugging task.

This paper is organized as follows. Section 2 introduces some definitions. Our proposal of computation graph is defined in 3. Section 4 discusses how the com-

putation graph can be used for debugging. Section 5 presents the debugger tool integrated in the system DES. Finally, Section 6 summarizes conclusions from this work and points out some future work.

2 Datalog Preliminaries

Definitions for Datalog mainly come from the field of Logic Programming. In this section, we only introduce the concepts which are needed in our setting, referring the reader to [7] for a more general presentation of Logic Programming.

We consider (recursive) Datalog programs with stratified negation [1,17], i.e., normal logic programs without function symbols. Stratification is imposed to ensure a clear semantics when negation is involved, and function symbols are not allowed in order to guarantee termination of computations, a natural requirement with respect to a database user.

A *term* is either a variable or a constant symbol. An *atom* is $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol and t_i are terms, $1 \leq i \leq n$, which can also be written as $p(\bar{t}_n)$. A *literal* is either an atom or a negated atom. A *positive* literal is an atom, and a *negative* literal is a negated atom. A negated atom is syntactically constructed as $not(A)$, where A is an atom. The atom contained in a literal L will be denoted as *atom(L)*. A *rule* R is an expression of the form $A : -L_1, \ldots, L_n$, where A is an atom and L_i are literals. All of the variables in a rule are assumed to be universally quantified. Concerning the rule R, A is referred to as the *head* of R, $L_1, \ldots, L_n$ as its *body*, and L_i as *subgoals*. Commas in bodies stand for conjunctions. A *fact* is a rule with empty body and *ground* head. The symbol $: -$ is usually dropped in this case. A *Datalog program* is a finite set of Datalog facts and rules. In order to fit with database notation, the term *relation* is used *in lieu* of *predicate*. A database *relation* is, therefore, a set of rules with the same predicate symbol and arity. If a relation is only defined with facts, it is called an *extensional-database* relation (EDB), whereas it is otherwise called an *intensional-database* relation (IDB). A *query* (term preferred in a deductive database context) or *goal* (term preferred in a logic programming context) is a literal (i.e., an atom or a negated atom) which can be solved by a Datalog system with respect to a given program. Analogously to literals, we say that a *positive* query is an atom, and a *negative* query is a negated atom. In contrast to facts, queries may contain variables.

Substitutions are defined as usual in logic programming. *Subst* denotes the set of all the substitutions. We also assume the existence of a composition operation between substitutions defined in the usual way and fulfilling the property $(s\theta)\sigma = s(\sigma \cdot \theta)$ for all $\sigma, \theta \in Subst$. Two formulae φ, φ' are *variants* if $\varphi = \varphi'\theta$ with θ a renaming. We use the notation *fresh(φ)* to represent a renaming of the formula φ which replaces all its variables by new variables.

Datalog programs resemble Prolog programs as the program (adapted from [19]) in Figure 1 suggests, which will be used as a running example for the rest of the paper. In Datalog programs, variables start with uppercase letters whereas constants start with lowercase letters (e.g., `X` and `nil`, respectively, in the example).

```
% Pairs of non-consecutive elements in br
between(X,Z) :-       br(X), br(Y), br(Z), X<Y, Y<Z.

% Consecutive elements in the sequence, starting at nil
next(X,Y) :-          br(X), br(Y), X<Y, not(between(X,Y)).
next(nil,X) :-        br(X), not(has_preceding(X)).

% Values having preceding values in the sequence
has_preceding(X) :-   br(X), br(Y), Y>X. % error: it should be Y<X

% Values in an even position of the sequence, including nil
even(nil).
even(Y) :-            odd(X), next(X,Y).

% Values in an odd position of the sequence
odd(Y) :-             even(X), next(X,Y).

% Succeeds if the cardinality of the sequence is even
br_is_even :-         even(X), not(next(X,Y)).

% Succeeds if the cardinality of the sequence is odd
br_is_odd :-          odd(X), not(next(X,Y)).

% Sequence
br(a).
br(b).
```

Fig. 1. Example of Datalog program

Predicate symbols start with lowercase letters (e.g., `between`). Code remarks start with `%` and apply up to the end of line.

The example program is intended to compute the parity of a given base relation `br(X)`, i.e., it can determine whether the number of elements in the relation (cardinality) is even or odd by means of the relations `br_is_even`, and `br_is_odd`, respectively. The relation `next` defines an ascending chain of elements in `br` based on their textual ordering, where the first link of the chain connects the distinguished node `nil` to the first element in `br`. The relations `even` and `odd` define the even, resp. odd, elements in the chain. Finally the relation `has_preceding` defines the elements in `br` such that there are previous elements to a given one (the first element in the chain has no preceding elements). The rule defining this relation includes an intended error (fourth rule in the example) which will be used in forthcoming sections to show how it is catched by the declarative debugger. The symbol $<$ denotes a built-in relation checking if some element is less than another w.r.t. the predefined term ordering. Observe that relations `br_is_even`, and `br_is_odd` are not *range restricted* because variable Y occurs only in a negative literal, and that therefore the program does not fulfill the usual *safety conditions* [17]. In fact, our setting does not enforces the use of safe programs. Only the stratification requirement is needed for the correctness of the debugging technique.

The semantics (i.e., the "meaning") of a Datalog program can be given by either the model-theoretic, proof-theoretic or fixpoint semantics. We focus on the

model-theoretic semantics. In particular, we consider *Herbrand interpretations* and *Herbrand models*, i.e., Herbrand interpretations that make every Herbrand instance of the program rules logically true formulae.

Given a Herbrand model $\mathcal{M}$, we define the meaning of a query Q in the context of a program P as the set:

$$Q_{\mathcal{M}} = \{Q\theta \in \mathcal{M}\}$$

with $\theta \in Subst$. We call $Q_{\mathcal{M}}$ the *answer* for Q w.r.t. P.

In Logic Programming languages such as Prolog, the least Herbrand model cannot be actually computed, because programs in general are non-terminating. Also the use of *negation as failure* contributes to the lack of completeness of Prolog computations. However, due to the use of stratified programs in Datalog, the existence of a so-called *supported model* $\mathcal{M}$ is ensured [1]. $\mathcal{M}$ is a minimal Herbrand model of program P that can be actually computed by a Datalog system. Thus, we assume that our Datalog system implementation yields the value $Q_{\mathcal{M}}$ for any query Q.

Additionally, we use the term *intended interpretation* represented by $\mathcal{I}$, to denote the model the user has in mind for the program. The *intended* answer for a query Q is accordingly defined as:

$$Q_{\mathcal{I}} = \{Q\theta \in \mathcal{I}\}$$

Then, the user can focus on queries and compare the intended interpretation to the minimal Herbrand model actually computed. We therefore speak of *validity* of a computed query w.r.t. the intended model of a program when:

$$Q_{\mathcal{M}} = Q_{\mathcal{I}}$$

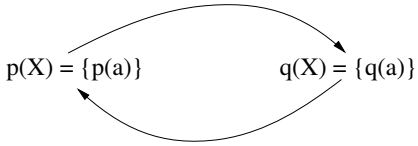
If given a program P we find some query Q s.t. $Q_{\mathcal{M}} \neq Q_{\mathcal{I}}$, we have that P is an *incorrect program*, which must include one or more *incorrectly defined relations*.

3 Computation Graphs

In this section, we define a suitable structure for representing Datalog computations. Usually in logic programming languages such as Prolog, the computations are represented through some tree structure such as the SLD-tree [7]. In the case of Datalog, we claim that a tree is not a convenient structure due to the different treatment of recursive programs. For instance consider the program:

```
p(a).
r(b).
p(X) :- q(X),r(X).
q(X) :- p(X).
```

In Prolog, the SLD-tree for the goal `p(X)` will contain an infinite branch, representing a non-terminating computation. However in Datalog the same goal is terminating and returns the finite answer: {`p(a)`} because the computation mechanism detects the repetition of the subgoal `p(x)` and avoids the infinite loop. Thus, our computation structure must represent finitely these situations, which can be achieved by using a graph:



The graph contains the two subgoals occurred during the computation together with their respective answers. It also indicates that `p(X)` and `q(X)` are mutually dependent. We will call such graph the *computation graph* for the goal w.r.t. the program. Observe that this graph is different from the *predicate dependency graph* [19] of the Datalog program, which show the connections between the relations from a static point of view. In the example, the dependence graph will include a vertex for the relation `r` connected to the vertex for `p`. However, our computation graph depends on the initial goal and the subgoals that occur during the computation and hence does not include any vertex for `r`.

The computation graph (CG in short) is a directed graph which only has one connected component, and that can contain cycles in the case of queries involving recursive predicates.

Each vertex of the CG contains all the information necessary for detecting its validity w.r.t. the intended model of the program. Hence the information stored at each vertex of the CG is the following:

- The query Q.
- The answer for the query.

Now we describe formally how the debugger builds a CG.

Definition 3.1 (**Computation Graph**) Given a Datalog program P and a query Q the computation graph of Q w.r.t. P is defined as the value *cg(P,Q)* defined as follows:

cg(P,Q)

Input:

- P: a stratified Datalog program.
- Q: a program query. It must be either of the form $p(\bar{a}_n)$ or $not(p(\bar{a}_n))$, with $p(\bar{a}_n)$ an atom and p a relation defined in P.

Output: a directed graph (V, E), where V is a set of vertices of the form $(Q = Q_{\mathcal{M}})$ and $E = \{(u, v) | u, v \in V\}$ is a set of edges.

Steps:

1 The first vertex v of the graph is associated to the initial query Q and is defined as $v := (p(\bar{a}_n) = \Pi)$, with $\Pi := p(\bar{a}_n)_{\mathcal{M}}$. The set Π can be obtained directly using the system to obtain the answer for the query $p(\bar{a}_n)$.

2 $V := \{v\}$, $E := \emptyset$.

3 Let A be an auxiliary set containing the vertices that must be unfolded in order to build the graph. Initially $A := \{v\}$.

4 While $A \neq \emptyset$ do:

i) Select any vertex u in A. $A := A \setminus \{u\}$. The vertex u must be of the form $(q(\bar{b}_k) = \Pi_q)$ for some q, Π_q, $\bar{b}_k$.

ii) Consider all the rules (disregarding facts) defining q: $R_{q_1}, \ldots, R_{q_s}$. For each R_{q_i}, $1 \leq i \leq s$:

a) N := *newVertices*($q(\bar{b}_k)$,*fresh*(R_{q_i}))

b) For each new vertex $(Q = Q_{\mathcal{M}}) \in N$ check whether exists already a vertex $(Q' = Q'_{\mathcal{M}}) \in V$ such that Q and Q' are variants. There are two possibilities:

- There exists such $(Q' = Q'_{\mathcal{M}})$. Then, $E := E \cup \{(u, (Q' = Q'_{\mathcal{M}}))\}$. That is, if the vertex already exists we simply add a new edge from u.
- Otherwise, $V := V \cup \{(Q = Q_{\mathcal{M}})\}$, $A := A \cup \{(Q = Q_{\mathcal{M}})\}$, and $E := E \cup \{(u, (Q = Q_{\mathcal{M}}))\}$.

newVertices(A,R)

Input:

- A: An atom of the form $q(\bar{b}_k)$.
- R: A program rule of the form $q_i(\bar{t}_n) :- L_1, \ldots, L_m$.

Output: a set S of vertices.

Steps:

1 If $\theta := m.g.u.(q(\bar{b}_k), q_i(\bar{t}_n))$ does not exist return $\emptyset$.

2 Otherwise, for each literal L_j in the right-hand side of rule R consider the next two possibilities:

i) $j = 1$. Then $v_1 := (atom(L_1)\theta = \Pi_1)$, where $\Pi_1 := (atom(L_1)\theta)_{\mathcal{M}}$. $S := \{v_1\}$. v_1 is the vertex associated with the first literal L_1 .

ii) If $j > 1$, let $\delta_1, \ldots, \delta_r$ be all the substitutions such that:

$$L_h\theta\delta_1 \in \mathcal{M}, \ldots, L_h\theta\delta_r \in \mathcal{M} \quad , \; h = 1, \ldots, j-1$$

This means that the first $j-1$ literals have succeeded for each substitution $\theta\delta_1, \ldots, \theta\delta_r$. Then the following new r vertices associated with the literal L_j are defined

$$v_{j_1} := (\ atom(L_j)\theta\delta_1 = \Pi_{j_1}) \qquad \ldots \qquad v_{j_r} := (\ atom(L_j)\theta\delta_r = \Pi_{j_r})$$

with $\Pi_{j_1} := (atom(L_j)\theta\delta_1)_{\mathcal{M}}, \ \ldots, \ \Pi_{j_r} := (atom(L_j)\theta\delta_r)_{\mathcal{M}}$.

Finally set $S := S \cup \{v_{j_1}, \ldots, v_{j_r}\}$, stating that we have created r new nodes.

End of Definition

Observe that if a relation p is only defined by facts (i.e., $s = 0$ at step 4.*ii*) of the *cg* algorithm), the CG only contains one vertex, and has no edges. A CG of the query `br_is_even` w.r.t the program in Figure 1 is presented in Figure 2 (ignore the bounded and colored vertices at the moment). For instance the vertex `even(X) = {`

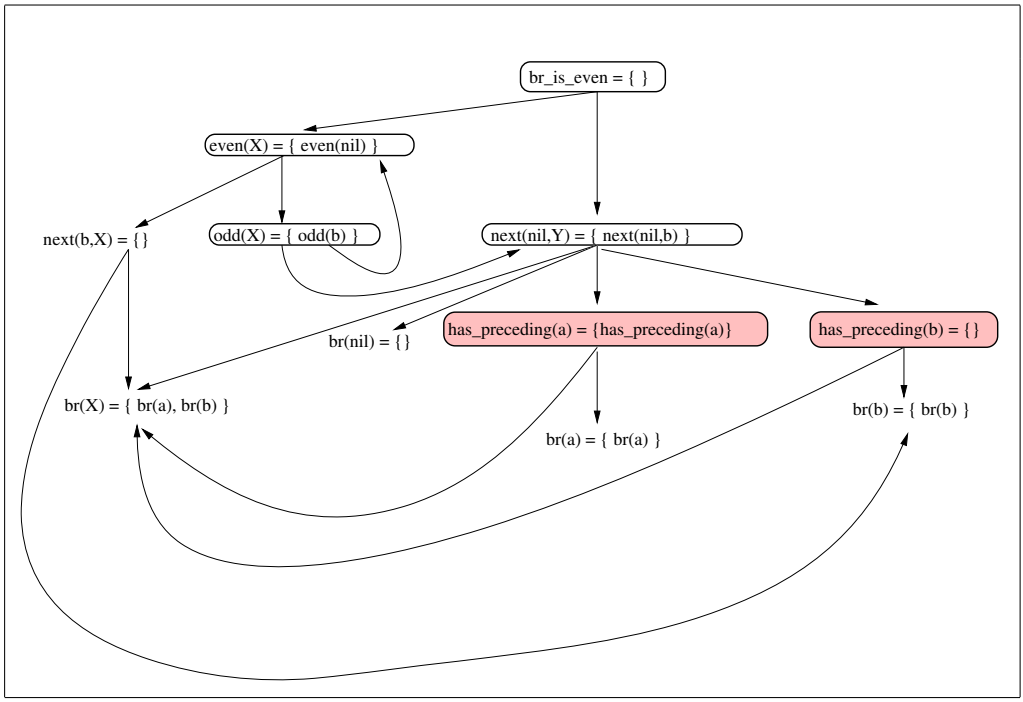


Fig. 2. Example of Computation Graph

`even(nil) }` includes the answer `even(nil)` for the query `even(X)`. The relation `even` is defined through one fact and one rule. The fact does not appear in the figure (they do not create new subgoals), and the rule yields two new vertices, one for each literal in its right-hand side. The first one, namely `odd(X)` , includes an answer with one result. This result has an associated substitution $\delta_1 = \{X \mapsto b\}$. The second vertex corresponds to `next(X,Y)`δ_1, which produces an empty answer. Notice that there is no edge from vertex `next(b,X) = { }` to any vertex corresponding to the relation `between`, although `between` occurs in the rhs of the first rule defining `next`. The reason is that there is not exist a substitution δ such that

$$br(b)\delta \in \mathcal{M} \ \wedge \ br(Y)\delta \in \mathcal{M} \ \wedge \ (b < Y)\delta \in \mathcal{M}$$

i.e. the first three literals for this clause do not succeed for any substitution δ. Also observe that there is no vertex corresponding to the symbol $<$ in the graph, because it is a primitive built-in relation and it is hence assumed correct by the debugger.

4 Declarative Debugging with CG's

In this section, we show how the computation graph is used by the debugger in order to detect incorrectly defined relations.

The debugging process starts when the user finds some unexpected answer for a query, i.e., some *initial symptom*. For instance, in the case of the program in Figure 1, the user expects that the answer for the query `br_is_even` should be

{br_is_even}, because the relation br contains two elements: a and b. However, the answer returned by the system is { }, which means that the corresponding goal was unsuccessful. Therefore, the user will start the debugger. The debugger proceeds by following the stages:

(i) First the system ensures that the computation for the goal is up to date. Then, it generates a suitable *computation graph* that represents the computation. In the case of our running example and the query br_is_even, the CG is displayed in Figure 2. This first phase is automatically performed by the tool.

(ii) Second, the CG obtained in the previous phase is traversed asking to the user about the validity of some vertices looking for a *buggy* vertex. A vertex is called buggy when it is non-valid but all its immediate descendants are valid. A buggy vertex always corresponds to an incorrectly defined relation, which is pointed out by the debugger as the cause of the error.

Therefore, the debugger will ask the user questions about the validity of certain vertices of the CG w.r.t. to the intended interpretation $\mathcal{I}$ of the program P. For instance, the intended interpretation of the program of the Figure 2 is:

$$\mathcal{I} = \{\ \textit{br\_is\_even, even(nil), even(b), odd(a), next(nil,a), next(a,b),}$$
$$\textit{has\_preceding(b), br(a), br(b)}\ \}$$

The debugger assumes that the user knows whether an instance of a query is in $\mathcal{I}$, i.e., that the user can determine the answer for any query. For instance, a possible question could be *Is odd(X)={odd(b)} valid?* The question must be understood as *Is {odd(b)} the expected answer for the query odd(X)?*. The answer will be no because the expected answer for a query *odd(X)* is {*odd(a)*}, because *a* is the only element of the sequence *a,b* that is in an odd position.

We can distinguish two reasons for detecting that a computed answer $Q_{\mathcal{M}}$ for a query Q is incorrect:

(i) There exists $\sigma \in \mathit{Subst}$ s.t. $Q\sigma \in \mathcal{M}$ but $Q\sigma \notin \mathcal{I}$. Then $Q_{\mathcal{M}}$ is called a *wrong answer*. For instance, the answer {*has_preceding(a)*} for the query *has_preceding(a)* in the CG of Figure 2 contains a wrong answer, because *a* has no preceding value in the intended interpretation.

(ii) There exists $\sigma \in \mathit{Subst}$ s.t. $Q\sigma \in \mathcal{I}$ but $Q\sigma \notin \mathcal{M}$. Then $Q_{\mathcal{M}}$ is called a *missing answer*. For instance, the vertex *even(X)*={*even(nil)*} in the Figure 2 contains a missing answer because for $\sigma = \{X \mapsto b\}$ we have $\mathit{even}(X)\sigma = \mathit{even}(b)$, and $\mathit{even}(b) \in \mathcal{I}$ but $\mathit{even}(b) \notin \mathcal{M}$.

Observe that the two errors can exist at the same time: $\mathit{odd}(X) = \{\mathit{odd}(b)\}$ is both a wrong answer ($\mathit{odd}(b)$ should not be in the answer) and a missing answer ($\mathit{odd}(a)$ should be in the answer). Declarative debuggers usually require the user to distinguish both types of errors in order to initiate the debugging process. These types of errors can require even different types of different computation structures. An advantage of our approach is that this distinction is not needed, and that the same structure, the CG is valid for both types of errors.

In fact, our approach does not match the general scheme proposed by L. Naish in [8] for declarative debugging, because it is based on a graph rather than on a tree. However, some of the basic results are still valid. For instance the *correctness* of the technique:

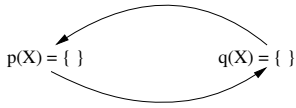
Theorem 4.1 *Any buggy vertex in a CG corresponds to an incorrectly defined relation.*

The informal reasoning behind this result is easy: since a buggy vertex is an incorrect vertex, this means that it contains an incorrect answer for its associated query. Also, in the graph construction algorithm, it can be checked that the immediate descendants of the vertex are the subqueries whose result were needed to produce such incorrect answers. But, if the subqueries returned correct values, the error must come from the relation itself, which is therefore incorrectly defined.

In the CG of the Figure 2 the incorrect vertices are surrounded by a box, while the two buggy vertices are contained in a colored box. Both buggy vertices correspond to the relation `has_preceding`, which corresponds to the only incorrectly defined relation of the program of Figure 1.

Another nice property of the general scheme based on computation trees is completeness: every computation tree with an incorrect node contains a buggy node. Unfortunately, this result does not hold in our setting. Consider for instance the program:

```
p(X) :- q(X).
q(X) :- p(X).
```



The CG, displayed at the right of the program will contain two vertices, both displaying the empty answer. Imagine also that either p or q is an incorrectly defined relation, because the user forgot to include in the program either the fact *p(a)* or *q(a)*. In either of theses cases $\mathcal{I} = \{p(a), q(a)\}$. Then, we will have a CG with two incorrect vertices and with no buggy vertex. In this case, our debugger will not point out any relation, but a *set of connected relations* as the cause of the error. Fortunately, this situation that leads to less informative diagnosis, is not usual in common Datalog programs.

5 Implementation

The Datalog Educational System (DES) is an open-source free Prolog-based implementation of a basic deductive database with stratified negation with Datalog as a query language. The system is implemented on top of Prolog and can be used from several Prolog interpreters (Ciao Prolog, GNU Prolog, SICStus Prolog, and SWI Prolog) running on several operating systems (OSs). Moreover, executables for several OSs (Windows 98 and later, and SunOS/Solaris) are also provided. It was aimed to have a simple, interactive, multiplatform, and affordable system for students and researchers. DES 1.3.0 is the current release, which enjoys full recursive evaluation with memoization techniques and stratified negation. DES is

implemented with the seminar ideas found in [6,16], that deal with termination issues of Prolog programs. These ideas have been already used in the deductive database community. Our implementation uses the concept of *extension table* for achieving a top-down driven bottom-up approach. In its current form, it can be seen as an extension of the work in [6] in the sense that, in addition, we deal with negation and undefined (although incomplete) information (cfr. [11] for further details about undefinedness). Once a query is computed, the extension table holds the computed meaning of the program restricted to the query, i.e., only the meanings of needed relations for computing the meaning of the query are computed.

We have implemented a debugger tool in the DES system based on the ideas presented in previous sections. Next, we describe some of its features.

As we have seen in Section 4, the debugging process consists of two phases. During the first phase, the tool builds a CG for the initial query Q w.r.t. the program P. This phase, in turn, can be divided into two parts:

(i) The debugger uses the system DES in order to produce the *extension table* for Q w.r.t. P.

(ii) The CG is built following the description given in Section 3. The answers Π at each vertex are obtained from the extension table. This corresponds to assuming that the system computes the supported Herbrand model $\mathcal{M}$. As we explained in Section 1, this is possible due to the requirements of stratification imposed to our Datalog programs.

The second phase consists of traversing the CG in order to find either a buggy vertex or a set of related incorrect vertices. The vertex associated to the initial query Q is marked automatically as *non-valid* by the debugger. The rest of the vertices are marked initially as *unknown*. In order to minimize the number of questions asked by a declarative debugger, several traversing strategies have been studied [4,13]. However, these strategies are only valid for declarative debuggers based on trees and not on graphs and new strategies are still to be investigated for this structure. Nevertheless, the currently implemented strategy already contains some ideas of how to minimize the number of questions in a CG:

- It firstly asks about the validity of vertices that are not part of a cycle, in order to find a buggy vertex if it exists. Only when this is no longer possible the vertices that are part of cycles are visited.
- Each time the user indicates that a vertex ($Q = \Pi$) is valid, i.e., the validity of the answer Π for the subquery Q is ensured, the tool changes to *valid* all the vertices with associated queries of the form $Q\theta$, with $\theta \in Subst$.
- Each time the user indicates that a vertex ($Q = \Pi$) is non-valid, the tool changes to *non-valid* all the vertices with associated queries Q', with $Q = Q'\theta$, $\theta \in Subst$.

The two last items help to reduce the number of questions deducing automatically the validity/non-validity of some vertices from the validity/non-validity of others. For instance, in Figure 2, the validity of the vertices containing *br(a)*={*br(a)*} and *br(nil)*={} can be deduced automatically from the validity of the vertex *br(X)*=

{*br(b),br(a)* }. The soundness of these deductions is established by the following proposition:

Proposition 5.1 *Let Q be a query, let $Q_{\mathcal{I}}$ be the answer of Q w.r.t. the intended interpretation $\mathcal{I}$, and let θ be a ground substitution, $\theta \in Subst$. Then:*

(i) *If $Q = Q_{\mathcal{I}}$ is valid, then $Q\theta = Q\theta_{\mathcal{I}}$ is valid.*

(ii) *If $Q = Q'\theta$ for some Q' and $Q = Q_{\mathcal{I}}$ is non-valid, then $Q' = Q'_{\mathcal{I}}$ is non-valid.*

Proof.

(i) If $Q = Q_{\mathcal{I}}$ is valid, then for all $\sigma \in Subst$, $Q\sigma \in \mathcal{I} \iff Q\sigma \in \mathcal{M}$. Thus, for all σ', $(Q\theta)\sigma' \in \mathcal{I} \iff (Q\theta)\sigma' \in \mathcal{M}$ simply considering $\sigma = (\sigma' \cdot \theta)$.

(ii) If $Q = Q'\theta$ for some Q' and $Q = Q_{\mathcal{I}}$ is non-valid, then there exists $\sigma \in Subst$ s.t. one of the two possibilities hold:
   (a) $Q\sigma \in \mathcal{M}$ but $Q\sigma \notin \mathcal{I}$.
   (b) $Q\sigma \in \mathcal{I}$ but $Q\sigma \notin \mathcal{M}$.

   Hence, $Q_{\mathcal{M}}$ is either a wrong or a missing answer. By defining $\sigma' = (\sigma \cdot \theta)$ we have that $Q'\sigma'$ is also a wrong or missing answer.

□

A debugger session for the query `br_is_even` of our running example:

```
DES> /debug br_is_even
Debugger started ...
Is br(b) = {br(b)}  valid(v)/non-valid(n) [v]? v
Is has_preceding(b) = {}  valid(v)/non-valid(n) [v]? n
Is br(X) = {br(b),br(a)}  valid(v)/non-valid(n) [v]? v

! Error in relation:  has_preceding/1
! Witness query: has_preceding(b) = { }
```

In this particular case, only three questions are necessary to find out that the relation *has_preceding* is incorrectly defined.

6 Conclusions and Future Work

In the previous sections we have presented a framework for the declarative debugging of Datalog programs. The proposed technique finds incorrect relation definitions in Datalog programs by comparing the results of the computations to the intended interpretation of each relation, which is assumed to be known by the user. Thus, our technique relies on the program semantics for debugging, disregarding the implementation issues. In Datalog this is not only an advantage; it is almost a necessity. The Datalog computations are based on operational features or program transformations that make the execution very difficult to follow and understand using the normal trace facilities included in logic languages such as Prolog.

We have also defined a suitable structure for representing the computations, the computation graphs. This represents a novelty w.r.t. the traditional presentation of

declarative debugging, which is based on trees rather than on graphs. Nevertheless, declarative debugging using computation graphs lacks some of the nice properties of usual declarative debugging such as completeness. We have shown that indeed in some Datalog programs is not possible to point out a single relation as the cause of an unexpected computation result, and that in those programs the debugger can only detect sets of mutually dependent relations as possible error sources, meaning that one or more of these relations are incorrectly defined. However, these situations are not common in practice.

It is important to emphasize that the debugger can be used for diagnosing errors starting either from a wrong or from a missing answer. Since Datalog programs are terminating, we can claim that the presented technique covers *all* the possible errors that produce unexpected answers in Datalog programs. This makes a difference w.r.t. other declarative debuggers that are limited to a particular kind of errors (i.e., only missing or only wrong answers). The ideas have been implemented in a working prototype included as part of the Datalog system DES.

As future work we plan to represent graphically the computation graph. This will help the user to find the error more easily, inspecting the graph and choosing freely the more convenient vertices to start the debugging process. Another improvement can be obtained by allowing the user to provide more informative answers. For instance, if the debugger knows that an answer is not only non-valid but *wrong*, i.e., it contains an unexpected atom, it can use this information to skip some questions; in particular, the questions involving children with empty answers, which are always valid w.r.t. wrong answers.

References

[1] Apt, K. R., H. A. Blair and A. Walker, *Towards a theory of declarative knowledge* (1988), pp. 89–148.

[2] Arora, T., R. Ramakrishnan, W. G. Roth, P. Seshadri and D. Srivastava, *Explaining program execution in deductive systems*, in: *Deductive and Object-Oriented Databases*, 1993, pp. 101–119.

[3] Beeri, C. and R. Ramakrishnan, *On the power of magic*, 1987, pp. 269–284.

[4] Caballero, R., *A declarative debugger of incorrect answers for constraint functional-logic programs*, in: *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming* (2005), pp. 8–13.

[5] Caballero, R., C. Hermanns and H. Kuchen, *Algorithmic Debugging of Java Programs*, Electronics Notes in Theorical Computer Science (2007), in Press.

[6] Dietrich, S. W., *Extension tables: Memo relations in logic programming.*, in: *SLP*, 1987, pp. 264–272.

[7] Lloyd, J., "Foundations of Logic Programming," Springer Verlag, 1984.

[8] Naish, L., *A Declarative Debugging Scheme*, Journal of Functional and Logic Programming **3** (1997).

[9] Ramakrishnan, R. and J. Ullman, *A survey of research on Deductive Databases*, The Journal of Logic Programming **23** (1993), pp. 125–149.

[10] Russo, F. and M. Sancassani, *A declarative debugging environment for Datalog*, in: *Proceedings of the First Russian Conference on Logic Programming* (1992), pp. 433–441.

[11] Sáenz-Pérez, F., *Datalog Educational System. User's Manual*, Technical Report 139-04, Facultad de Informática, Universidad Complutense de Madrid (2004), `http://des.sourceforge.net/`.

[12] Shapiro, E., "Algorithmic Program Debugging," ACM Distiguished Dissertation, MIT Press, 1982.

[13] Silva, J., *A Comparative Study of Algorithmic Debugging Strategies*, in: *Proc. of International Symposium on Logic-based Program Synthesis and Transformation LOPSTR 2006*, 2007, pp. 134–140.

[14] Specht, G., *Generating explanation trees even for negations in deductive database systems*, in: *Proceedings of the 5th Workshop on Logic Programming Environments*, Vancouver, Canada, 1993.

[15] Sterling, L. and E. Shapiro, "The art of Prolog: advanced programming techniques," MIT Press, Cambridge, MA, USA, 1986.

[16] Tamaki, H. and T. Sato, *Old resolution with tabulation*, in: *Proceedings on Third international conference on logic programming* (1986), pp. 84–98.

[17] Ullman, J., "Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)," Computer Science Press, 1995.

[18] Wieland, C., *Two explanation facilities for the deductive database management system DeDEx.*, in: H. Kangassalo, editor, *ER* (1990), pp. 189–203.

[19] Zaniolo, C., S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian and R. Zicari, "Advanced Database Systems," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.