Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

An Operator for Composing Deductive Databases with Theories of Constraints

D. Aquilino, P. Asirelli,

C. Renso e F. Turini

Nota Interna B4-42
Dicembre 1994

An Operator for Composing Deductive Data Bases with Theories of Constraints

D. Aquilino [*] P. Asirelli[†] C. Renso F. Turini[‡]

Abstract

An operation for restricting deductive databases represented as logic programs is introduced. The restrictions are represented in a separate deductive database. The operation is given an abstract semantics in terms of the immediate consequence operator. A transformational implementation is given and its correctness is proved with respect to the abstract semantics.

1 Introduction

Deductive Databases can potentially solve many problems in the field of data and knowledge management. However, in order to make the approach really viable, it is necessary to define an environment at least as rich as the one which has been developed for other kinds of database management systems.

At present, in the database area, a lot of attention is devoted to studying the possibility of combining different databases or, in any case, databases which have been developed within other projects, and that may be resident at different sites. The goal generally can be seen from the point of view of building applications by means of cooperative, interoperable and/or distributed databases. Our claim is that one approach to the above problems is

[*]Intecs Sistemi SpA, Via Gereschi 32, 56125 Pisa, Italy, aquilino@pisa.intecs.it

[†]Istituto di Elaborazione dell'Informazione-CNR, Via S.Maria 46, 56126 Pisa, Italy, asirelli@iei.pi.cnr.it

[‡]Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy, {renso,turini}@di.unipi.it

the definition of an environment that provides a set of basic operators, with a clear formal semantics, for the combination of different databases.

One interesting point is the ability of defining a particular view of the database as a refinement of its relations.

The notion of view we have in mind should allow one, among other possible operations, to impose dynamically a set of restrictions on a database in order to filter elements which do not satisfy certain properties out of the relations of the database. More precisely, we propose an operation that, given a deductive database and a set of restrictions, computes a new deductive database, the relations of which are the original ones properly filtered according to the restrictions.

It is worth noting now that the restrictions are represented as a deductive database itself, i.e. as a collection of deductive rules which establish conditions about the belonging of a tuple to a relation. In other words, the database of restrictions can be seen as an incremental refinement of the original database.

Given that we represent deductive databases as logic programs, our aim is the definition of an operation that, given two logic programs - the original database and the one containing the restrictions - computes a new logic program that behaves as the restricted database.

The following example clarifies the expected behavior of the operation. We have a theory `graph` which defines two extensional relations, `node` and `edge`, that represent a graph, and two intensional relations, `path` and `bidirectional_edge`, with the straightforward meaning.

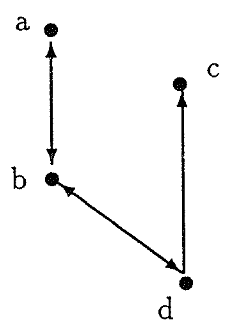
```
Theory Graph

node(a)                   edge(d,c)
node(b)                   edge(b,a)
node(c)                   edge(a,b)
node(d)                   edge(b,d)
                          edge(d,b)

path(X,Y) ← edge(X,Y)
path(X,Y) ← edge(Z,Y),path(X,Z)

bidirectional_edge(X,Y) ← edge(X,Y), edge(Y,X)
```

which is graphically represented as follows



We can now *constrain* the above database by means of rules which impose further restrictions on the relations. For example the rule

node(X) ← path(a,X)

establishes that we accept only nodes that are reachable from the node a.

The following rule establishes that in the constrained graph each node has degree al least two

node(X) ← bidirectional_edge(X,Y), bidirectional_edge(X,Z)

In the following, we show the results of some queries on the theory **graph** w.r.t. the constraints.

The query ← *node*(*b*) satisfies both constraints, while the query ← *node*(*d*) satisfies the first constraint but not the second one.

The approach presented in this paper stems from two separate lines of research:

- the general study of operators on logic programs [7]
- the transformational approach to constraints handling in deductive databases [4]

The study of operators on logic programs consists in the abstract definition of a suite of operations, in the study of their formal properties, in the design of several implementation strategies, and in the application to several problems in the field of knowledge representation and software engineering.

The transformational approach to integrity constraints checking was defined in [4] for databases represented by positive logic programs. There, the

theory of constraints was not a logic program itself but a conjunction of formulas denoting the "if then" part of relations. The assumption to constraints satisfaction was that the constraints should have been logical consequences of the database, i.e. the formulas ought to be true in the minimal model of the database. According to this view, the considered approach defined a new logic program, "modified" by the constraints formulas, so that the minimal model of the resulting database was a model of the theory of constraints as well. The approach of [4] was afterwards extended by [8] to deal with stratified logic program, in view of programming by exception as an approach to default reasoning.
The paper is organized as follows. In Section 2, we describe a complex application of the incremental restriction of deductive databases in the field of building software engineering tools. In Section 3 we present the operation of restriction via its abstract semantics. The abstract semantics is given compositionally in terms of the consequence operators associated to the deductive data base and the database containing the restriction rules. Section 4 discusses an implementation of the operator based on a transformation of the database and the restricting database into a new *constrained* database. We establish also the correctness of the implementation with respect to the abstract semantics. Section 5 presents the comparison of our approach with other approaches in the literature and discusses future work.

2 A Motivating Example

The use of constraints as a restriction in deductive databases [4] has been experimented in Gedblog which is a multi-theory deductive database management system [3]. Gedblog supports the consistent design and prototyping of graphic applications through an incremental development and/or by combining pre-defined theories and constraints. Knowledge is expressed as a deductive database spread into multiple extended logic theories. Each theory entails a piece of application data model and behavior by means of

- facts and rules to express the "general" knowledge;
- integrity constraints formulas to express either "exceptions" to the general knowledge, or general "requirements" of the application being developed;

- transactions to express atomic update operations that can be performed on the knowledge-base.

A very interesting and practical experiment in using Gedblog has been carried out by realising a graphical editor for Oikos process structure [2]. Oikos is an environment that provides a set of functionalities to easily construct process-centered software development environments. In the following a brief description of Oikos is provided. This description will highlight the characteristics that have a direct impact with the realization of the Oikos editor.

In Oikos [1] a software process model is a set of hierarchical entities. Each entity is an instance of one of the Oikos classes and represent a modelling concept. Classes are: Process, Environment, Desk, Cluster, Role and Coordinator.

An entity can be either structured (i.e. formed by other entities) or simple (i.e. a leaf in the model structure). Oikos defines a top-down method to construct process models using two descriptions of the entities: abstract and concrete. The abstract entities are introduced first and then refined in the concrete ones. At the end of the modelling activity an enactable model is obtained by adding to the defined entities the needed details. The method establishes some constraints about the entities and their use as sub-entities during the model refinement.

All entities mentioned above, with the exception of coordinators, have an angelic counterpart, that is, an abstract specification of their contribution to the process activity.

The Oikos Process Editor consists of several databases. There are two kinds of theories:

- the ones representing the Oikos model;

- the ones imposing constraints on the modelling process.

The theory `Oikos-instance` contains a specific instance of a Oikos process and the theory `Oikos-model` describes the types of the entities of the Oikos model and some relations between them (consistency, refinement...). In the following we will refer to the union of this two theories as the Oikos model. The theory `Oikos-constraints` contains all the contraints on the Oikos processes.

The following theory defines the entities of the process and it binds each entity to the corresponding kind. For example the first rule states that the compound entity `slc` is of type `process`.

```
Oikos-instance

   compound(slc,process)
   management(manager,env)
   angel(v_v, ang_env)
   angel(c_e, ang_env)
   part_of(slc,process,v_v,ang_env)
   part_of(slc,process,c_e,ang_env)
   angel(role1,ang_role)
   part_of(slc,process,role1,ang_role)
```

The following clauses define which kinds are concrete:

```
Oikos-model

   concrete_kind(process).
   concrete_kind(env).
   concrete_kind(desk).
   concrete_kind(cluster).
   concrete_kind(role).
   concrete_kind(coord).
```

and which are compound:

```
   compound_kind(process).
   compound_kind(env).
   compound_kind(desk).
   compound_kind(cluster).
```

The following rule states that each concrete entity is necessarily compound:

```
concrete(Name,Kind) <-- compound(Name,Kind)
```

Then there are abstract types:

```
abstract_kind(ang_process).
abstract_kind(ang_env).
abstract_kind(ang_desk).
abstract_kind(ang_cluster).
abstract_kind(ang_role).
```

and management ones:

```
management_kind(process).
management_kind(env).
management_kind(desk).
```

The definition of the consistent predicate states that the entity given as the second argument can be defined inside the entity which appears as the first argument.

```
consistent(process,ang_process).
consistent(process,ang_env).
consistent(process,ang_cluster).
consistent(process,coord).
consistent(env,ang_cluster).
```

⋮

The following clauses states that the second argument is a refinement of the first argument.

```
is_refinement(ang_process,process).
is_refinement(ang_env,env).
```

⋮

We can now discuss a number of constraints on the database that are given as rules restricting the relations of the database.

An Oikos compound entity has compound kind and at least a coordinator:
`Oikos-constraints`

```
1)    compound(Name,Kind) <-
                compound_kind(Kind),
                part_of(Name,Kind,Coord,coord).
```

A Oikos angelic entity has abstract kind and is part of a concrete one:

```
2)    angel(Name,Kind) <-
            part_of(Name1,Kind1,Name,Kind),
            compound_kind(Kind1),
            abstract_kind(Kind).
```

A Oikos entity which has parts is compound and the relationship between kinds is consistent (predicate 'consistent'):

```
3)    part_of(Name1,Kind1,Name2,Kind2) <-
                          compound(Name1,Kind1),
                          consistent(Kind1,Kind2).
```

Name2 may be a refinement of Name1 if Name1 is angelic, Name2 is concrete and the kind of Name2 is a refinement of the kind of Name1 (predicate 'is_refinement'):

```
4)    refinement(Name1,Name2) <-
                  angel(Name1,Kind1),
                  concrete(Name2,Kind2),
                  is_refinement(Kind1,kind2).
```

Each process has a management entity:

```
5)    compound(Name,process) <-
                 management(Name,Name1,Kind),
                 management_kind(Kind).
```

Here we consider some examples of successful and failing queries. Suppose to add a set of new instances to the theory Oikos-instance. Let Oikos-new-inst be the following:

```
Oikos-new-inst

    part_of(desk1, desk, slc, process).
    compound(role1,role).
    refinement(process,desk).
    angel(coord1, coord).
```

The new instance of the Oikos process is the theory `Oikos-instance` ∪ `Oikos-new-inst`. Notice that the query ← *part_of(desk1, desk, slc, process)* succeeds if we evaluate it in the unconstrained Oikos process (`Oikos-instance` ∪ `Oikos-new-inst` ∪ `Oikos-model`), while it fails when we evaluate it in the constrained theory:

(`Oikos-instance` ∪ `Oikos-new-inst` ∪ `Oikos-model`) $/_{IC}$ `Oikos-constraints` where $A/_{IC}B$ denotes the operation of restricting A by means of B.
In fact constraint number 3 constraints the facts of the `part_of` relation. In our case desk is a compound kind (see Oikos-model), but desk in not consistent with process, that is, we cannot have a process inside a desk.

An analogous case is the query ← *compound(role1,role)*. The constraint 1 is not satisfied, because role does not have a compound_kind definition in Oikos-model.

The query ← *refinement(slc,desk1)* does not satisfy the constraint 4 because *slc* is not an angel.
Finally, the query ← *angel(coord1,coord)* fails because of the failure of the constraint 2 which states that coord1 must be a part of a compound_kind and it must also be an abstract_kind.

3 Abstract definition

We are now in the position of giving an abstract definition of the operation $/_{IC}$ that restrict a deductive database by means of a database of constraints. The abstract definition is given via the notion of *immediate consequence operator*. The immediate consequence operator T_P associated to a logic program P is a function that points out which consequences are deducted from a given interpretation by the program P in a single inference step. The abstract definition of an operation on logic programs can then be given in a compositional way by defining the immediate consequence operator of the resulting logic program as a composition of the immediate consequence operators of the argument programs.

We recall now from [7] the basic definition of immediate consequence operator and the definition of two basic operations on logic programs: union and intersection.

For a logic program P, the immediate consequence operator $T(P)$ is a continuous mapping over Herbrand interpretations defined as follows [10]. For any Herbrand interpretation I:

$$A \in T(P)(I) \iff (\exists \bar{B} :\ A \leftarrow \bar{B} \in \mathit{ground}(P)\ \wedge\ \bar{B} \subseteq I)$$

where $\bar{B}$ is a (possibly empty) conjunction of atoms.

The semantics of program operations is given in a compositional way by extending the definition of T with respect to the first argument. We assume that the language in which programs are written is fixed. Namely, the Herbrand base we refer to is determined by a set of function and predicate symbols that include all function and predicate symbols used in the programs being considered.

For any Herbrand interpretation I:

$$\begin{aligned} T(P \cup Q)(I) &= T(P)(I)\ \cup\ T(Q)(I) \\ T(P \cap Q)(I) &= T(P)(I)\ \cap\ T(Q)(I) \end{aligned}$$

The above definition generalises the notion of immediate consequence operator from programs to compositions of programs. The operations of union and intersection of programs directly relate to their set-theoretic equivalent. The set of immediate consequences of the union (resp. intersection) of two programs is the set-theoretic union (resp. intersection) of the sets of immediate consequences of the separate programs.

Let us now go back to the new operator we want to define. Informally, a program P constrained by another program Q is obtained by the union of two parts. One is the intersection of the two theories, that forces the theories to agree during the deduction. But the intersection alone is not enough, because some clauses would be missing in the result. In particular, we miss all the clauses which are defined in P and not constrained by Q. Those are of two kinds: the ones which do not have a definition in Q, and those which have a definition in Q that constrains only a subset of atoms potentially derivable in P. In the last case the clause defining a given predicate r in P is not fully instatiated while the definition of r in Q has at least a ground argument. In order to include both the first and the second kinds of clauses in the resulting program we define a new operator $/_{HIC}$. The definitions of these new operators are given as follows.

Definition 1 $T(P\ /_{IC}\ Q) = \lambda I.\ T(P\ /_{HIC}\ Q)(I) \cup T(P \cap Q)(I)$

Definition 2 $T(P\ /_{HIC}\ Q) = \lambda I.\ T(P_1)(I) \cup (T(P_2)(I) \setminus T(Q)(\mathcal{B}_P))$

where
P_1 contains the predicate definition of P which are not defined in Q
P_2 contains the predicate definition of P which are also defined in Q, such that $P = P_1 \cup P_2$ and $\mathcal{B}_P$ is the Herbrand Base associated to P

We are mainly interested in a transformational definition of the $/_{IC}$ operator. However, although the above definition, based on set-theoretic difference, is easy to understand, it is not easy to implement in a transformational way. Hence, we need to find another abstract definition of the operator, equivalent to the previous one, easier to implement. Of course, we want to show that the two abstract definitions coincide. The idea is to exploit the observation that the operation of set-theoretic difference can be turned into a combination of intersection and complement. In our case, the complement of a program is not defined, so we need first to give an abstract semantics and a trasformational definition for it.

Definition 3 $T(P\ /_{HIC}\ Q) = \lambda I. T(P_1)(I) \cup T(P_2 \cap (Q,P)^c)(I)$

where the semantics of $(Q,P)^c$ is defined as follows

Definition 4 *$T((Q,P)^c) = \lambda I.\ \mathcal{B}_{P_\pi} \setminus T(Q)(\mathcal{B}_P)$*

We denote with $\mathcal{B}_{P_\pi}$ the subset of the Herbrand base of P obtained by selecting those predicate names which have a definition in Q

Theorem 1 *Let P, Q be programs, P_1 and P_2 defined as above, A be an atom. Then*
$A \in T(P_1)(I) \cup T(P_2 \cap (Q,P)^c)(I) \Longleftrightarrow A \in T(P_1)(I) \cup (T(P_2)(I) \setminus T(Q)(\mathcal{B}_P))$

Proof
$A \in (T(P_1)(I) \cup T(P_2 \cap (Q,P)^c)(I))$
$\Longleftrightarrow$ *definition of* P_2
$A \in (T(P_1)(I) \cup T(P \cap (Q,P)^c)(I))$
$\Longleftrightarrow$ *definition of* $\cup$ *and* $\cap$
$A \in (T(P_1)(I) \vee (\ A \in T(P)(I) \wedge A \in T((Q,P)^c)(I)))$
$\Longleftrightarrow$ *definition of* $T((Q,P)^c)(I)$
$A \in T(P_1)(I) \vee (\ A \in T(P)(I) \wedge A \in (\mathcal{B}_\pi \setminus T(Q)(\mathcal{B}_P)))$
$\Longleftrightarrow$ *definition of set difference*
$A \in T(P_1)(I) \vee (\ A \in T(P)(I) \wedge A \in \mathcal{B}_{P_\pi} \wedge A \notin T(Q)(\mathcal{B}_P))$
$\Longleftrightarrow$ *definition of* $\cap$
$A \in T(P_1)(I) \vee (A \in T(P)(I) \cap \mathcal{B}_{P_\pi} \wedge A \notin T(Q)(\mathcal{B}_P))$
$\Longleftrightarrow$ *definition of* P_2 *and* $\mathcal{B}_{P_\pi}$
$A \in T(P_1)(I) \vee (A \in T(P_2)(I) \wedge A \notin T(Q)(\mathcal{B}_P))$
$\Longleftrightarrow$ *definition of set difference*
$A \in T(P_1)(I) \vee (A \in T(P_2)(I) \setminus T(Q)(\mathcal{B}_P))$
$\Longleftrightarrow$ *definition of* $\cup$
$A \in (T(P_1)(I) \cup (T(P_2)(I) \setminus T(Q)(\mathcal{B}_P)))$

$\diamond$

4 Transformational Definition

We address now the problem of providing an implementation for our abstract operator. There are several possibilities for implementing operations on logic programs and they can be classified in two broad categories: an interpretation oriented approach, and a compilation oriented approach. As discussed in [7] the first one can rely upon either metaprogramming techniques or the design of specific abstract machines, whilst the second approach consists in designing proper transformations which map logic programs into logic programs. In this paper we give a transformation oriented implementation of the operator $/_{IC}$. We first recall from [7] the transformational implementation of $\cup$ and $\cap$. We overload the symbols $\cup$ and $\cap$ and we use them also for denoting the transformations.
Given two programs $\mathcal{P}$ and $\mathcal{Q}$, the program corresponding to the union $\mathcal{P} \cup \mathcal{Q}$ is just the set theoretic union of the clauses of the two programs:

$$\mathcal{P} \cup \mathcal{Q} \;=\; \{A \leftarrow B \mid (A \leftarrow B \in \mathcal{P}) \vee (A \leftarrow B \in \mathcal{Q})\}.$$

On the other hand, the implementation of the $\cap$ operation exploits the basic unification mechanism of logic programming. Given two programs $\mathcal{P}$ and $\mathcal{Q}$, a clause $A \leftarrow B$ belongs to $\mathcal{P} \cap \mathcal{Q}$ if there is a clause $A' \leftarrow B'$ in $\mathcal{P}$ and there is a clause $A" \leftarrow B"$ in $\mathcal{Q}$ such that their heads unify, that is $\exists\vartheta : \vartheta = mgu(A', A")$, and $A = (A')\vartheta$, and B is obtained from $(B', B")\vartheta$ by removing duplicated atoms.

$$\begin{aligned}\mathcal{P} \cap \mathcal{Q} \;=\; \{A \leftarrow B \mid\; & (A' \leftarrow B' \in \mathcal{P}) \wedge (A" \leftarrow B" \in \mathcal{Q}) \wedge \\ & \vartheta = mgu(A', A") \wedge (A = A'\vartheta) \wedge (B = B'\vartheta \cup B"\vartheta)\}.\end{aligned}$$

For example, consider the programs

$\mathcal{P}$	$\mathcal{Q}$	$\mathcal{R}$
$r(x,y) \leftarrow s(x)$	$t(x) \leftarrow$	$r(x,y) \leftarrow s(x), t(y)$
$s(f(x)) \leftarrow$		$s(x) \leftarrow$

The program corresponding to the program expression $(\mathcal{P} \cup \mathcal{Q}) \cap \mathcal{R}$ is obtained as follows:

$\mathcal{P} \cup \mathcal{Q}$	$(\mathcal{P} \cup \mathcal{Q}) \cap \mathcal{R}$
$r(x,y) \leftarrow s(x)$	$r(x,y) \leftarrow s(x), t(y)$
$s(f(x)) \leftarrow$	$s(f(x)) \leftarrow$
$t(x) \leftarrow$	

We are now in the position of giving the definition of the constraining operator.

Definition 5 $P /_{IC}\ Q = P_1 \cup (P_2 \cap Q^c) \cup (P \cap Q)$

where
P_1 and P_2 are defined as above and Q^c is defined as follows.

Definition 6 *For each predicate p in Q let C_p be the set of clauses defining it. Then we define a correspondent set of unit clauses $C_p{}'$ in Q^c in the following way:*

- *if C_p is a ground unit clause*
  $p(t_1, \ldots, t_n) \leftarrow$
  then $C_p{}'$ is
  $p(X_1, \ldots, X_n) \leftarrow X_1 \neq t_1$
  $\vdots$
  $p(X_1, \ldots, X_n) \leftarrow X_n \neq t_n$

- *if $C_p = p(X_1, \ldots, X_n)$ is a unit clause where X_i is a variable and $X_i \neq X_j\ \forall\ i,j \in [1, \ldots n]$ then C'_p is the empty set*

- *if $C_p = p(X_1, \ldots, X_n)$ is a unit clause where X_i is a variable $\forall\ i \in [1, \ldots\ n]$ and $X_i = X_j$ for some i,j, then C'_p is*
  $p(X_1, \ldots, X_n) \leftarrow X_i \neq X_j$

- *if $C_p = p(t_1, \ldots, X_n)$ is a unit clause containing both ground and non ground arguments, then C'_p is obtained by applying the rules defined above for each argument, according to its nature.*

- *if $C_p = p(t_1, \ldots, t_n) \leftarrow q(s_1, \ldots, s_m), \ldots, r(v_1, \ldots, v_k)$ is a clause of Q, not necessary ground, then C'_p is obtained by "forgetting" the body $q(s_1, \ldots, s_m), \ldots, r(v_1, \ldots, v_k)$ and applying the method to the remaining unit clause*

- *if C_p is a set of clauses c1, c2, ..., cm defining the predicate p, then $C_p{}'$ is obtained in the following way:*

  $C_p{}' = c1^c \cap\ c2^c, \cap\ \ldots, \cap\ cm^c$

Then Q^c is the set of all the clauses in $C_p{}'$ for each predicate p defined in Q

From now on we will use the ground predicate with two arguments: the first is the Herbrand base with respect to which we instantiate the program given as the second argument which is a program. So, Ground($\mathcal{B}_P$)(Q^c) denotes a set of ground unit clauses which derive all the facts ($\in \mathcal{B}_P$) that cannot unify with the heads of the clauses in Q

Observation 1 *Consider the program Q' obtained from Q taking only the heads of the clauses of Q. If we trasform Q' into a program in Constraint Logic Programming (with '=' constraints), then Q^c is obtained by replacing each '=' symbol occurring in the body, with the $\neq$ symbol.*

Now, we want to show that a ground atom A is an immediate consequence of the trasformed program Q^c with respect to an interpretation I if and only if A belongs to $T((Q,P)^c)(I)$. In order to give the proof, we need a preliminary result

Lemma 1 $A \leftarrow\ \in Ground(\mathcal{B}_P)(Q^c) \Longleftrightarrow A \notin T(Q)(\mathcal{B}_P) \wedge A \in \mathcal{B}_{P_\pi}$

Proof
By contradiction
Suppose A is $p(t_1, \ldots, t_n) \in T(Q)(\mathcal{B}_P)$. This means that it is a ground instance of a head of a clause in Q. The head of the clause in Q can have one of three possible forms:
1) $p(t_1, \ldots, t_n) \leftarrow$
2) $p(X_1, \ldots, X_n) \leftarrow$
3) $p(X_1, \ldots, t_n) \leftarrow$
Then, in Q^c there must be the complement of it. In case 1) in Q^c there must be:
$p(X_1, \ldots, X_n) \leftarrow X_1 \neq t_1, \ldots, X_n \neq t_n, \ldots$
but this is a contradiction because $A \leftarrow\ \in$ Ground($\mathcal{B}_P$)(Q^c) by hypothesis. In case 2) either the variables are all different or some of them are the same. In the former case no new definitions of p are introduced.

In the latter case, if $X_i = X_j$, then in Q^c there will be
$p(X_1, \ldots, X_n) \leftarrow X_i \neq X_j$
that contradicts the hypothesis.
Finally, in case 3), in Q^c there will be:
$p(X_1, \ldots, X_n) \leftarrow X_n \neq t_n$,
and this is a contradiction because $A \leftarrow \in \text{Ground}(\mathcal{B}_P)(Q^c)$ by hypothesis.

$\diamond$

The following theorem establishes the correctness of the *complement* operation with respect to the natural interpretation in terms of set-difference with respect to the universe.

Theorem 2 *Given a program Q, let R be the program obtained from the transformation* Q^c*. Then* $T(R)(I) = \mathcal{B}_{P_\pi} \setminus T(Q)(\mathcal{B}_P)$

Proof
Let A an atom
$A \in T(R)(I)$
$\Longleftrightarrow$ *definition of* T
$\exists\, \mathbf{B} \mid A \leftarrow \mathbf{B} \in \text{Ground}(\mathcal{B}_R)(R) \wedge \mathbf{B} \subseteq I$
$\Longleftrightarrow$ *R contains only facts*
$A \leftarrow \in \text{Ground}(\mathcal{B}_R)(R)$
$\Longleftrightarrow$ *lemma 1*
$A \notin T(Q)(\mathcal{B}_P) \wedge A \in \mathcal{B}_{P_\pi}$
$\Longleftrightarrow$ *definition of* $\setminus$
$A \in (\mathcal{B}_{P_\pi} \setminus T(Q)(\mathcal{B}_P))$

$\diamond$

We are, at last, in the position of proving the correctness of the transformation oriented implementation of $/_{IC}$ with respect to its abstract semantics.

Theorem 3 *Let P, Q be programs and P_1, P_2 defined as in the previous section. Then*

$$P /_{IC}\ Q = P_1 \cup (P_2 \cap Q^c) \cup (P \cap Q) \text{ is correct w.r.t. } T(P /_{IC}\ Q)$$

Proof

The correctness of union and intersection was proved in [6]. The correctness of the ^{c} operator is established by theorem 2.

$\diamond$

The following example clarifies the trasformational approach

Example 1 *Let us consider the programs*

P	Q
$A(a,a) \leftarrow$	$A(a,y) \leftarrow C(a,y)$
$A(a,b) \leftarrow$	
$B(b,b) \leftarrow$	
$B(c,c) \leftarrow$	
$C(b,a) \leftarrow$	
$A(x,y) \leftarrow B(x,y)$	

We split P into P_1 and P_2 as stated in section 3

P_1	P_2
$B(b,b) \leftarrow$	$A(a,a) \leftarrow$
$B(c,c) \leftarrow$	$A(a,b) \leftarrow$
$C(b,a) \leftarrow$	$A(x,y) \leftarrow B(x,y)$

The complement of Q, built according to the definition 6, is

Q^c

$A(x,y) \leftarrow x \neq a$

The first step is to compute the theory obtained from the intersection of P_2 and Q^c. Namely, we want to obtain those rules the head of which has a predicate defined in Q, but are not constrained by Q. This is the case in which Q constraints a specific instantiation of rules of P. So,

$$\begin{array}{l} P_2 \cap Q^c \\ A(x,y) \leftarrow x \neq a,\ B(x,y) \end{array}$$

The second step is to build $P \cap Q$, that is, all the clauses of P which agree with the constraints.

$$\begin{array}{l} P \cap Q \\ A(a,a) \leftarrow C(a,a) \\ A(a,b) \leftarrow C(a,b) \\ A(a,y) \leftarrow B(a,y), C(a,y) \end{array}$$

Thus, the transformed program which satisfies the constraints is

$$\begin{array}{l} P_1 \cup (P \cap Q^c) \cup (P \cap Q) \\ A(x,y) \leftarrow x \neq a,\ B(x,y) \\ A(a,a) \leftarrow C(a,a) \\ A(a,b) \leftarrow C(a,b) \\ A(a,y) \leftarrow B(a,y), C(a,y) \\ B(b,b) \leftarrow \\ B(c,c) \leftarrow \\ C(b,a) \leftarrow \end{array}$$

5 Conclusions

We have presented an operation for the definition of restricting views of deductive databases, represented as logic programs. The approach allows for the definition and use of databases as theories of constraints for another existing database. The main advantage of the approach is that the database and the theories of constraints can be defined separately by different users and at different stages of the development of an application. The example of Section 2 illustrates exactly this point: deductive databases supporting software engineering tools are incrementally restricted by means of theories of constraints.

The operation of restriction has been given an abstract semantics. The semantics has been given compositionally, in terms of the immediate consequence operators associated to the deductive database to restrict and the databases containing the restriction rules. An implementation of the operation has been presented. Such an implementation is based on a transformation of the database and the restricting database into a new constrained database. The correctness of the implementation with respect to the abstract semantics has been proved.

The example od Section 2 is part of a complex application developed in [4]. The transformed program resulting from the implementation of operation $/_{IC}$ is also obtainable by the so called *Modified Program Approach* of [4] supported by GEDBLOG [3]. The novelty is that the transformation is now associated to an operation on logic program with a clear formal semantics. We consider this as step towards the design of a semantically well founded environment for prototyping and developing deductive databases. The development can be carried on by using the operation of these paper and other ones to put together theories developed either elsewhere, or by different users, or resident in different sites, i.e. a formal basis to study applications obtained by means of cooperative, interoperable and/or distributed databases.

A transformational approach for constraints and default reasoning is described in [8]. Although the transformation is similar to the one proposed here and in GEDBLOG, no formalization in terms of operation on logic programs is given. On the other hand in [8] also normal stratified logic programs are dealt with. We are currently looking at a similar extension, in which the operations are extended to handle normal programs. From an abstract viewpoint, the definition of the semantics is given by referring to Fitting's immediate consequence operator [9]. From an implementational viewpoint we are considering both a transformational approach based on intensional negation [5] and an interpretation oriented approach.

References

[1] V. Ambriola and C. Montangero. Oikos: Constructing process-centered sdes, software process modelling and technology. In A. Finkelstein,

J. Kramer, and B. Nuseibeh, editors, *Research Study Press.* J. Wiley and sons, 1994.

[2] D. Apuzzo, D. Aquilino, and P. Asirelli. A Declarative Approach to the Design and Realization of Graphic Interfaces. Technical Report B4-39, October 1994, IEI-CNR Internal Report, 1991. submitted for publication.

[3] P. Asirelli, D. Di Grande, P. Inverardi, and F. Nicodemi. Graphics by a logic database management system. *Journal of the Visual languages and Computing*, 1994. to appear.

[4] P. Asirelli, M. De Santis, and M. Martelli. Integrity constraints in logic databases. *Journal of Logic Programming*, 3:221,232, 1985.

[5] R. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. A trasformational approach to negation in logic programming. *Journal of Logic Programming*, 2:201–228, 1990.

[6] A. Brogi. *Program Construction in Computational Logic.* PhD thesis, University of Pisa, March 1993.

[7] A. Brogi, A. Chiarelli, P. Mancarella, V. Mazzotta, D. Pedreschi, C. Renso, and F. Turini. Implementations of program composition operations. In *International Conference on Programming Languages Implementation and Logic programming.* Springer-Verlag, 1994.

[8] R.A. Kowalski and F. Sadri. Logic programs with exceptions. In DHD Warren and P. Szeredi, editors, *7th International Conference on Logic Programming, Proceedings*, pages 598–613. The MIT Press, Cambridge, Mass., 1990.

[9] Fitting M. A kripke-kleene semantics for general logic programs. *Journal of Logic Programming*, 2:295–312, 1985.

[10] van Emden M. H. and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.