HAL
open science

Datalog extensions for database queries and updates

Serge Abiteboul, V. Vianu

To cite this version:

Serge Abiteboul, V. Vianu. Datalog extensions for database queries and updates. [Research Report] RR-0900, INRIA. 1988. inria-00075656

HAL Id: inria-00075656

https://inria.hal.science/inria-00075656v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.(1)39 63 55 11

Rapports de Recherche

N° 900

DATALOG EXTENSIONS FOR DATABASE QUERIES AND UPDATES

Programme 4

Serge ABITEBOUL
Victor VIANU

Septembre 1988

RR.0900

DATALOG EXTENSIONS FOR DATABASE QUERIES AND UPDATES

EXTENSIONS DE DATALOG POUR LES REQUETES ET MISES-A-JOUR DANS LES BASES DE DONNEES

Serge Abiteboul
I.N.R.I.A.
Domaine de Voluceau-Rocquencourt
78153 Le Chesnay
FRANCE
(serge@inria.inria.fr)

Victor Vianu[1]
CSE Department
University of California at San Diego
La Jolla, CA 92093
USA
(vianu@sdcsvax.ucsd.edu)

Abstract:

Deterministic and non-deterministic extensions of Datalog with fixpoint semantics are proposed, and their expressive power characterized. It is argued that fixpoint semantics provides an elegant way to overcome the limited expressive power available with purely declarative semantics. The Datalog extensions range from complete languages to languages capturing interesting complexity classes of queries and updates: NPTIME and NPSPACE in the non-deterministic case, and the fixpoint queries and *while* queries in the deterministic case. The connection between the Datalog extensions and explicitly procedural languages, as well as fixpoint extensions of first-order logic, is also investigated.

Résumé:

Des extensions déterministiques et non-déterministiques de Datalog avec des sémantiques de point fixe sont proposées et leur puissance d'expression caractérisée. Il est montré qu'une sémantique de point fixe procure une manière élégante de surpasser les limitations de puissance disponible dans des langages purement déclaratifs. Les langages obtenus vont de langages complets à des langages correspondant à des classes de complexité intéressantes: NPTIME et NPSPACE dans le cas non-déterministe, et les requêtes de point fixe ou les requêtes *while* dans le cas déterministe. Le lien entre ces extensions de Datalog et des langages procéduraux ou des extensions par des opérateurs de point fixe de la logique du premier ordre est aussi étudié.

[1] This work was performed in part while the author was visiting at I.N.R.I.A. The author was supported in part by the National Science Foundation under grant number IST-8511538.

PAPIER RECUPERE ET RECYCLE

DATALOG EXTENSIONS FOR DATABASE QUERIES AND UPDATES

Serge Abiteboul
I.N.R.I.A.
Domaine de Voluceau-Rocquencourt
78153 Le Chesnay
FRANCE
(serge@inria.inria.fr)

Victor Vianu[1]
CSE Department,
University of California at San Diego
La Jolla, CA 92093
USA
(vianu@sdcsvax.ucsd.edu)

INTRODUCTION

The use of the logic programming paradigm in the context of data and knowledge bases has been a primary focus of research in the last few years. Recently, much of that research revolved around attempts to develop extensions of Datalog with increased expressive power, providing forms of non-monotonic reasoning (see [Ap, Ka, U]). In this paper, we propose a variety of extensions of Datalog with fixpoint semantics. We argue that fixpoint semantics provide an elegant way to overcome the limited expressive power available with purely declarative semantics. The focus of the results is on the expressive power of the languages, and on understanding the functionality and interactions of the programming primitives used. In particular, we highlight the connection between the Datalog extensions with fixpoint semantics and explicitly procedural languages.

The most popular extension of Datalog to date provides semantics to a class of Datalog programs with negation, satisfying a syntactic criteria called "stratification" [ABW, CH2, N, VG, BF, P]. Stratified Datalog constitutes a departure from traditional declarative semantics, which are based exclusively on model theory. In contrast, the semantics of stratified Datalog specifies in effect an *order* of evaluation of the rules, which results in the computation of a particular model called "perfect", which is one of several models of the corresponding sentences. Indeed, it appears unlikely that purely declarative semantics can provide languages considerably more powerful than Datalog. Thus, purely declarative semantics for expressive languages are likely to be rather complicated and artificial. The Datalog extensions that we propose, like stratified Datalog, provide

semantics which involve a procedural component, in that the intended model is specified as the result of computing a fixpoint associated with the program. We continue to call these languages "declarative" because of the lack of explicit control.

The Datalog extensions that we define can be viewed both as query languages and as update languages (to unify the discussion, we refer to queries or updates as *database transformations* and to the languages as *database languages*). We consider a family of *non-deterministic* languages and a family of *deterministic* languages. The choice between non-deterministic and deterministic semantics results from the decision to consider one possible application of a rule at a time, or to fire in *parallel* all rules that apply. Within each family, we provide *complete* languages and restrictions of the complete languages which correspond to interesting complexity classes of transformations. Completeness is achieved by providing a mechanism for introducing "invented" values into the database. (With fixed schemas and without the ability to increase the active domain, only transformations whose computation requires polynomial space can be computed.) Intuitively, a variable occurring in the head of a rule and not in the body is interpreted as an invented value. We also consider negations in heads of rules, interpreted as deletions. This allows invalidating a previously asserted fact, which is a key aspect of database updates. Interesting classes of transformations captured by the Datalog extensions include the NPTIME and NPSPACE transformations (in the non-deterministic case), and the set of fixpoint queries and the transformations corresponding to the *while* language of [Ch] (in the deterministic case).

We next illustrate informally the semantics and main primitives used in the various languages. We start with determinism versus non-determinism. In the case of Datalog, a program can be evaluated indifferently by applying one rule at a time, or by firing all rules that apply at once, until a fixpoint is reached. When negation is allowed in bodies of rules, this is no longer the case. The choice of firing all rules simultaneously results in deterministic semantics, whereas firing one rule at a time yields non-determinism. Indeed, consider the program P_1:

$$R(x) \leftarrow S(x), \neg T(x)$$
$$T(x) \leftarrow S(x), \neg R(x).$$

Suppose that we apply this program to the instance I such that $I(S)=\{1,2,3\}$ and $I(R)=I(T)=\emptyset$. With the non-deterministic semantics, we fire rules one at a time, in a non-deterministic manner. We can derive several fixpoints, for instance $I(S)=\{1,2,3\}$, $I(R)=\{1\}$, $I(T)=\{2,3\}$, and $I(S)=\{1,2,3\}$, $I(R)=\{1,2\}$, $I(T)=\{3\}$. Each of the fixpoints is a model for the set of rules. Other models can be obtained as well. With the deterministic semantics, all possible applications of the rules are fired simultaneously. Then P_1 applied to the same instance I yields the unique result $I(S)=I(R)=I(T)=\{1,2,3\}$, obtained in a single stage. This too is a model for the set of rules. Note

that this particular model cannot be obtained with the non-deterministic semantics.

Each of the deterministic and non-deterministic semantics is best suited to particular types of applications. For instance, consider the well-known *game of life*, which is usually described by several rules which are applied simultaneously to generate consecutive states. The parallel firing of rules associated with the deterministic semantics is naturally suited for such an application (see Example 4.6). Non-deterministic semantics resulting from firing rules one at a time is natural in other applications, such as production systems [DE] or tutoring systems [SCG]. Non-deterministic updates are discussed in [Ab,MS2,MW]. Of course, deterministic and non-deterministic semantics coincide for Datalog. It is an interesting problem to find conditions under which this holds for other languages.

Without deletions, both the deterministic and non-deterministic semantics are *inflationary*, i.e. the database grows continuously throughout the computation. We also consider non-inflationary languages by allowing negative literals in the heads of rules and interpreting them as deletions. This is particularly well-suited for update languages, where the ability to retract a previously asserted fact is crucial. To illustrate negations in heads of rules, consider the program P_2 (G represents a graph):

$$\neg G(x,y) \leftarrow G(x,y), G(y,x)$$

With a non-deterministic semantics, P_2 brings G to a "triangular" form by removing non-deterministically *one* edge <x,y> or <y,x> for each pair of edges <x,y>, <y,x> of G. With the deterministic semantics, P_2 removes from G *both* <x,y> and <y,x> for each pair of edges <x,y> and <y,x> in G. Note that a "triangularization" of G cannot be achieved by *any* deterministic program. We lastly illustrate the use of invented values. Consider the following "unsafe" program P_3 with non-deterministic semantics:

$$S(x,y), T(x,y,z) \leftarrow R(x,y), \neg S(x,y).$$

Variable z, which occurs only in the head, allows to "invent" new values. Suppose that K is an instance such that K(R)={[1,2],[3,4]} and K(S)=K(T)=∅. Then one possible computation derives S(1,2) and T(1,2,171); then S(3,4) and T(3,4,5); this yields a model of the program. Intuitively, each tuple in R is copied in S and T, and "marked" in T with an invented value.

The semantics of the Datalog extensions involve an implicit procedural element (the computation of the fixpoint). It turns out that this limited procedurality is sometimes sufficient to simulate languages with explicit, powerful control mechanisms like composition and iteration. This issue is intimately connected with the expressive power of the Datalog extensions, and is highlighted throughout the paper. To illustrate the techniques involved in simulation of control,

consider Datalog programs with negations in bodies of rules and a deterministic fixpoint semantics (Datalog$^{\neg}$). (This language was independently proposed in [AV3,KP].) Suppose that we wish to compose two programs Q_1 and Q_2. The problem is to inhibit Q_2 until after Q_1 has been evaluated. If Q_1 is recursion-free, Q_2 must be inhibited for a number of steps which is constant (input independent). It is easy to do this using several 0-ary control predicates which are fired in sequence. If Q_1 is recursive the above technique does not apply, since Q_2 must be inhibited for a number of steps that is data dependent. However, we will show how this can be achieved using a "differential" technique which detects when the computation of the fixpoint of Q_1 is completed, by keeping track of the facts newly inserted at consecutive iterations. This technique can be extended to show that Datalog$^{\neg}$ has the same power as a procedural language with composition and iteration, and that it yields the fixpoint queries.

The non-deterministic semantics generally provides weaker control capability, due to the non-deterministic firing of rules. The techniques used in the deterministic case must be augmented with means to correct erroneous non-deterministic choices. In this context, deletions and invented values contribute to the ability to simulate control.

To understand the issue of control, we consider the procedural languages studied in [AV1, AV2], which are essentially the Datalog extensions with added explicit control. The procedural languages use the elementary operations of tuple insertion and deletion, composition, a *while* construct, and a *with new* construct for inventing new values. The connection between the Datalog extensions and their procedural counterparts is underscored throughout the paper. In particular, most proofs concerning expressive power of the Datalog extensions involve simulations of the procedural languages. Thus, it becomes apparent which of the Datalog extensions can simulate their procedural counterparts. In particular, the connection between various features of the Datalog extensions (like invented values, or deletions) and the ability to simulate explicit control, becomes clear. Such results are of practical importance, since they suggest how explicit control mechanisms can be used in conjunction with declarative languages. This added flexibility would result in programs whose semantics are clearer to users than programs where intricate control is encoded using weak implicit control.

Similar insights are obtained by examining the connections between the safe Datalog extensions and fixpoint extensions of first-order logic. (Indeed, the fixpoint logics are similar to the procedural languages: composition is analogous to nesting, and iteration is similar to an application of the fixpoint operator.) We first consider first-order logic augmented with an inflationary fixpoint operator (FO+IFP), which is known to define the fixpoint queries [GS]. We show the equivalence of Datalog$^{\neg}$ and FO+IFP. The simulation of the inflationary fixpoint logic by Datalog$^{\neg}$ provides as

a side effect some results on the inflationary logic: the collapse of the hierarchy based on the depth of nesting of the fixpoint operators (which was a known result of [GS]), and a normal form. We also provide other fixpoint extensions of first-order logic for the non-deterministic and non-inflationary cases. The non-inflationary fixpoint logic is obtained using a partially defined fixpoint operator PFP instead of the classical inflationary operator IFP. The non-deterministic one uses an operator *witness* which yields formulas with *several* possible interpretations for a given structure. Results like the collapse of the hierarchy and existence of normal forms are also exhibited.

The paper consists of six sections. Section 1 contains preliminaries, including a review of the procedural languages investigated in [AV1,AV2]. (This makes the paper self contained.) In Sections 2 and 3, the non-deterministic and deterministic extensions of Datalog are presented, and their expressive power investigated; in particular, the connection with the procedural languages is exhibited. The variations of the Datalog extensions allowing negations in heads of rules are studied in Section 4. The connection between the Datalog extensions and fixpoint extensions of first-order logic is studied in Section 5. Section 6 contains conclusions, including a summary of the results.

The various languages are summarized at the end of the paper in Figure 1. Their expressive power is given in Figure 2, and connections with procedural languages and fixpoint logics in Figure 3.

1. BACKGROUND

We start by reviewing some basic database terminology and notation. We also recall the procedural languages introduced in [AV1,AV2] and results on their respective power.

1.1 Preliminaries

We assume that the reader is familiar with the basic concepts and terminology of relational database theory (see [U]). We also refer to [Ka] for a survey of the field. We review here some database terminology and notation.

We assume the existence of three infinite and pairwise disjoint sets of symbols: the set **att** of *attributes*, the set **dom** of *constants*, and the set **var** of *variables*. A *relational schema* is a finite set of attributes. A *tuple* over a relational schema R is a mapping from R into **dom** $\cup$ **var**. A *constant tuple* over a relational schema R is a mapping from R into **dom**. An *instance* over a relation schema R is a finite set of constant tuples over R. A *database schema* is a finite set of

relational schemas. An *instance* I over a database schema **R** is a mapping from **R** such that for each R in **R**, I[R] is an instance over R. In general, we use A, B, C,... for attributes, a, b, c,... or 0, 1, 2,... for constants, x, y, z,... for variables, and $\vec{r}$, $\vec{s}$,... for tuples. We usually denote relational schemas by R, S,..., database schemas by **R**, **S**,..., and database instances by I, J,.... The set of all instances over a schema **R** is denoted by inst(**R**). The set of all constants occurring in an instance I is denoted by const(I).

We view updates and queries as transformations of database instances into other database instances. While in some cases, the transformations considered will be mappings, in others, non-determinism is allowed, and the transformation is described as a relation between database instances. In a database context, such transformations are not arbitrary. Clearly, the relation between instances has to be at least recursively enumerable (r.e.). It is usually required that instances over a *fixed* schema be related to instances over another *fixed* schema. Finally, it is also required that constants (except perhaps for a fixed number) be uninterpreted. The last property has been introduced in [H, AU, CH1] under different names and with minor differences. We use here the terminology of [H], where the property is called C-genericity. Formally, let **R** and **S** be database schemas, and C a <u>finite</u> set of constants.

- A subset τ of inst(**R**) $\times$ inst(**S**) is *C-generic* iff for each bijection ρ over **dom** which is the identity on C, $(I,J) \in \tau$ iff $(\rho(I),\rho(J)) \in \tau$.
- A mapping τ from inst(**R**) to inst(**S**) is *C-generic* iff its graph is C-generic.

We now define three important classes of transformations. Let **R** and **S** be database schemas.

- A *(non-deterministic) database transformation* (from **R** to **S**) is a subset of inst(**R**) $\times$ inst(**S**) which is r.e., and C-generic for some finite C.
- A *finitely non-deterministic database transformation* (from **R** to **S**) is a non-deterministic database transformation τ such that for each instance I over **R**, the set $\{J \mid (I,J) \in \tau\}$ is finite.
- A *deterministic database transformation* (from **R** to **S**) is a mapping from inst(**R**) into inst(**S**) which is partial recursive, and C-generic for some finite C.

For conciseness, we usually refer to database transformations simply as "transformations".

Note that, although not allowed by traditional database systems, non-deterministic updates or queries arise quite naturally (e.g., "Find one cafe' at the intersection of Blvd. St. Michel and Blvd. St. Germain"). To illustrate the three kinds of transformations defined above, consider the

following operations: t_1 = insert(5), t_2 = delete_random_tuple, and t_3 = insert_random_tuple. The first transformation is deterministic. The second one is finitely non-deterministic but not deterministic. The last one is not finitely non-deterministic.

With the three classes of transformations defined above, we can introduce completeness criteria for languages based on their capability to express transformations in these classes. Informally, a *language* L specifies a set of programs. The semantics of a program in L is given by a transformation called the *effect* of the program, and denoted by $eff_L(t)$. Note that $eff_L(t)$ is a mapping if the transformation is deterministic, and a relation otherwise. Now, a language L is *non-deterministic complete* if it defines precisely the set of non-deterministic database transformations; similarly for *finitely non-deterministic complete* and *deterministic complete*. The concept of completeness is originally from [CH1]. The precise variations used here were introduced in [AV2].

We will refer to the following well-known database languages:

- Relational calculus [C]: this is a first-order calculus without function symbols;
- Datalog: a Datalog program consists of a set of rules (function-free Horn clauses) of the form $R(\vec{r}) \leftarrow \phi$, where ϕ is a list of positive atoms;
- *FP* [CH2]: fixpoint formulas are obtained from the first-order constructors ($\exists$,$\forall$,$\wedge$,$\vee$,$\neg$) and a fixpoint constructor (that can be applied on "positive" formulas).
- *while*: the *while* language (which is the same as *LE* of [Ch] and *RQ* of [CH2]) uses ranked variables that can hold relations of fixed arity. The constructors of the language are composition, assignment of a relational algebra expression to a variable, a *while* construct which permits to iterate a program until a variable holds an empty relation.

Throughout the paper, we will refer to complexity classes of transformations. We use as complexity measures the time (space) used by a Turing Machine to produce a standard encoding of the output instance starting from an encoding of the input instance. The measures are functions of the size of the input instance. Note that the complexity of a transformation is not expressed in terms of the corresponding recognition problem. Thus, for each Turing Machine complexity class C there is a corresponding complexity class of transformations denoted DB-C. In particular, the class of database transformations which can be computed by a deterministic Turing machine in polynomial time is denoted DB-PTIME. (Similarly, for DB-PSPACE.) The analog for non-deterministic transformations are DB-NPTIME and DB-NPSPACE. Note that Savitch's theorem (PSPACE=NPSPACE) is not relevant in this context, since a PSPACE transformation is

deterministic by definition, whereas NPSPACE contains non-deterministic transformations. (So, DB-NPSPACE≠DB-PSPACE.)

1.2 The TL and detTL languages

We now review the procedural languages studied in [AV1,AV2]. We do this in some detail, since many proofs of the paper consist of simulations of these procedural languages.

In all the procedural languages, two basic operations (denoted, respectively, by $i_R(\vec{r})$ and $d_R(\vec{r})$) allow the insertion or deletion of a tuple $\vec{r}$ in a relation R. The content of a relation R can be completely deleted by the operation $erase_R$. Three constructs are used besides the basic operations: composition (denoted by ";"), a construct to perform iterations (*while*), and finally one to randomly assign a new domain value to a domain variable (*with new*). The last construct permits the introduction in the database of values which were not originally part of the active domain. (Recall that we assume an infinite domain.)

A *while* statement is of the form: *while <condition> do t done*. Domain variables are used in the body of a *while* statement, and in its condition. The semantics attached to the *while* is based on valuations of the variables to domain values. Two semantics can be given to a *while* statement: a deterministic one and a non-deterministic one. With the non-deterministic semantics, a valuation satisfying the condition is non-deterministically chosen. The program in the body of the *while* is then applied for that valuation. This is iterated until no valuation satisfying the condition can be found. Clearly, the choice of valuation introduces non-determinism. With the deterministic semantics, <u>all</u> valuations satisfying the condition are considered simultaneously. The program of the body is applied for each such valuation. The result of one iteration is then the <u>union</u> of the results for each valuation. This is iterated until no valuation satisfying the condition can be found.

We now present the syntax and semantics of the language TL more formally. Let **R** be a database schema. A *condition* over a database schema **R** is an expression $Q_1 \wedge \cdots \wedge Q_n$ where each Q_i is an atomic formula over **R**, i.e., an expression of the form $R(\vec{r})$, $\neg R(\vec{r})$, $x = y$ or $x \neq y$, for some $R \in \mathbf{R}$, some tuple $\vec{r}$ over R (possibly with variables), and x in **var**, y in (**var** $\cup$ **dom**). Intuitively, "parameterized" programs are programs with "free" variables (not bound to any condition). Due to space limitations, we do not define here formally the notion of free variable of a program. A *parametrized program* in **TL** over a database schema **R** is an expression obtained as follows:

(1) for R in **R**, and for each tuple $\vec{r}$ over R (possibly with variables), $i_R(\vec{r})$, $d_R(\vec{r})$, and $erase_R$ are parameterized programs, and

(2) if t, t' are parameterized programs over **R**, Q a condition over **R**, and z is a free variable in t, then *(t;t')*, *while Q do t done*, and *with new z do t done* are parameterized programs over **R**.

A *program* t in (**TL**) is a parameterized program with no free variables.

The semantics of a TL program t is defined as a binary relation between database instances, denoted by $eff_{TL}(t)$. Intuitively, insertions, deletions and erase are interpreted in the natural way. The interpretation of the program (t;t') is the product of the binary relations corresponding to t and t'. Next, (I,J) is in eff_{TL}(*while Q do t done*) if there is a sequence of instances $I=I_0,...,I_n=J$ such that

- for each i ($i < n$), there exists a valuation v such that $I_i \models vQ$ and $<I_i, I_{i+1}>$ is in $eff_{TL}(vt)$, and
- for all valuations v, $J \not\models vQ$.

Finally, (I,J) is in eff_{TL}(*with new z do t done*) iff (I,J) is in $eff_{TL}(vt)$, where v is a valuation of z such that v(z) is not in const(I) $\cup$ const(t).

When a program is applied to a given database, its effect is often interpreted by identifying some relations as input relations, and other relations as output relations. In addition to semantically significant input and output relations, the programs may use "temporary" relations. Thus, it appears useful to also define the effect of a program with respect to specified input and output database schemas. Given a program t and an input-output (i-o) schema <**R**,**S**>, t transforms instances over **R** into instances over **S** as follows: relations which are not in the input schema are assumed to be empty before the program is executed; after the program is run, the relations in the output schema must contain the desired result. (The content of the other relations is immaterial.) The effect thereby obtained is denoted by eff(**R**,**S**,t).

Our definition of the language TL allows programs to produce domain values not present in the original database or in the program, and which are thus "unsafe". We recall here the definition of safety of [AV1], and two syntactic criteria to guarantee safety introduced there. A program t is *safe* with respect to an i-o schema <**R**,**S**> iff for each (I,J) in eff(**R**,**S**,t), the set of constants appearing in J is included in those of I together with those of t. To achieve safety, the obvious restriction is to completely forbid the invention of values. This yields *strongly-safe TL* (STL), the language obtained from TL by removing the "with new" construct. An alternative is to allow the use of invented values in the columns corresponding to some attributes that occur only in

temporary relations but not in relations of the i-o schema. This can be done in the following way. A program has a safe-attribute-set θ if

- θ contains all the attributes of the input or output relations; and
- each occurrence of a variable in an insertion into a θ-column is "positively bound" by some value in a θ-column (see [AV2]).

A program is weakly-safe if it has a safe-attribute-set. The corresponding language is called *weakly-safe TL (WTL)*. The connection between the various notions of safety is studied in [AV2].

We next consider the deterministic counterpart of TL (detTL), and its safe restrictions. TL programs may be non-deterministic for two reasons: (a) they produce values which are arbitrary, and (b) the choice of a valuation in a *while* loop is arbitrary. In order to obtain determinism, we use a different semantics of the *while*, and also require weak safety. A *detTL program* is a weakly-safe TL program. (Syntactically, there is no difference between WTL and detTL.) The semantics of insertions, deletions and *with new* is like in TL. A statement *while <cond> do t done* has the following semantics. One iteration is obtained by considering in "parallel" <u>all</u> the satisfying valuations of *<cond>* in the active domain of the database, applying t for all these valuations, and taking the <u>union</u> of the results. This is iterated until *<cond>* is not satisfied by any valuation. We also require that two "branches" that are viewed as realized in parallel not invent the same new value. With these restrictions, it can be shown that if t is a detTL program over some i-o schema <**R**,**S**>, then eff(**R**,**S**,t) is a C-generic <u>mapping</u>.

As in the case of TL, we can require strong safety of programs in detTL. The language thereby obtained from detTL by disallowing the "with new" construct is called *strongly-safe deterministic TL* (SdetTL).

We next recall results on the power of the languages TL, detTL, WTL, STL, and SdetTL.

Theorem 1.1 [AV2]:

(a) TL is non-deterministic complete.

(b) WTL is finitely non-deterministic complete.

(c) detTL is deterministic complete.

(d) STL computes DB-NPSPACE.

(e) A transformation is computable by SdetTL iff it is computable by the *while* language (extended with constants). □

Finally, we mention another result (that we shall use) concerning a trade-off between negation and deletion in some of the procedural languages. Specifically, under certain conditions, deletions can be simulated using negation and invented values. For each procedural language L discussed above, we denote by L^+ the language restricted by disallowing deletions.

Theorem 1.2 [AV2]: Each TL (detTL) program over disjoint i-o schemas is equivalent to a TL^+ ($detTL^+$) program. □

2 NON-DETERMINISTIC LANGUAGES

In this section, we define several non-deterministic extensions of Datalog which allow for negations in bodies of rules. We consider both safe and unsafe variations and study the expressive power of the languages that are obtained.

We first introduce the language DL. (Its procedural analog is TL.) The syntax of DL is the syntax of Datalog extended by allowing (i) negative literals in the body, (ii) more than one positive literal in the head and (iii) variables appearing in the head but not in the body. More precisely, we have:

Definition: A *Declarative Language (DL)* program is a finite set of rules of the form

$$A_1,\ldots,A_q \leftarrow B_1,\ldots,B_n$$

($q > 0$, $n \geq 0$) where each A_i is a positive literal of the form $Q(x_1,\ldots,x_m)$ ($m \geq 0$), and each B_i is a positive literal of the form $Q(x_1,\ldots,x_m)$ ($m \geq 0$) or $x_1 = x_2$ (the x_i's are domain variables or constants), or the negation of such a literal.

The expression $A_1,\ldots,A_q$ is called the *head*, and $B_1,\ldots,B_n$, the *body*.

If P is a program, sch(P) denotes the database schema consisting of all relation schemas occurring in P. The semantics of a program P is given by the *effect* eff(P) of P, which is a binary relation on inst(sch(P)). Intuitively, DL programs work as follows. New facts are inferred by repeated instantiations of rules in the program, in any order. Variables occurring in the head but not in the body play the role of the *with new* construct in TL. Thus, instantiations of such variables must be *new* constants, i.e. constants not appearing in the current database instance or in the program. Facts are added to the database until no additional facts can be inferred, and no facts are ever deleted. (In this respect the semantics is inflationary, since the database grows continuously.) Of course, there is no guarantee of termination. The random assignment of new values to

variables occurring only in the head, as well as the arbitrary choice of consecutive instantiations, introduce non-determinism. The semantics of DL is formalized next.

Definition: The *effect* $eff_{DL}(P)$ (eff(P) when DL is understood) of a DL program P is the binary relation over inst(sch(P)) consisting of all pairs (I,J) for which there is a sequence $I=I_0,...,I_p=J$ such that

(1) for each k, $0 \leq k < p$, there is a rule $A_1,...,A_q \leftarrow B_1,...,B_n$ in P, and a valuation v of the variables such that

(1-a) vx belongs to $const(I_k) \cup const(P)$ iff x is a variable occurring in the body,

(1-b) $I_k \models vB_1 \wedge..\wedge vB_n$,

(1-c) $I_{k+1} = i_1...i_q(I_k)$ where for each j, $1 \leq j \leq q$, $i_j = i_Q(vx_1,...,vx_m)$ if $A_j = Q(x_1,...,x_m)$; and

(2) for each extension $I_0,...,I_{p+1}$ of $I_0,...,I_p$ satisfying (1), $I_{p+1} = I_p$.

To distinguish a set of input relations and a set of output relations among relations in sch(P), we sometimes say that a program P is *over* an i-o schema <**R**,**S**>, where **R** and **S** are subsets of sch(P) (not necessarily disjoint). The effect eff(**R**,**S**,P) of a program P w.r.t. the i-o schema <**R**,**S**> is the binary relation from inst(**R**) to inst(**S**), induced by $eff_{DL}(P)$ as follows. A pair (I,J) is in eff(**R**,**S**,P) if for some pair (I',J') in $eff_{DL}(P)$, $I'|_R = I$, I' is empty outside **R**, and $J'|_S = J$.

We also consider weakly-safe and strongly-safe versions of DL (denoted WDL and SDL, respectively). A DL rule is *strongly-safe* if each variable in the head occurs in the body. A program in DL is strongly-safe if all its rules are strongly-safe. Note that this restriction is analogous to disallowing the *with new* construct to obtain strong safety in procedural languages.

Weak safety is defined relative to a given i-o schema, as for TL, using the auxiliary concept of "safe-attribute-set". Let P be a DL program over some i-o schema <**R**,**S**> and θ a set of attributes. Then θ is a *safe-attribute-set* w.r.t. <**R**,**S**> if

- θ contains all attributes occurring in **R** or **S**, and
- for each rule r in P and each variable x, if x occurs in the head in some column A in θ, then x occurs in a positive literal in the body in a column in θ.

A program is *weakly-safe* w.r.t. <**R**,**S**> if it has a safe-attribute-set w.r.t. <**R**,**S**>. Thus, "invented" values are allowed only in particular columns of the temporary relations (i.e., not input

or output relations) in sch(P).

Note that by definition, a program P is strongly-safe iff the set of all attributes occurring in sch(P) is a safe-attribute-set for P. Therefore, each strongly-safe program is also weakly safe.

We briefly show that checking if a program is weakly-safe can be performed in polynomial time. This is done by computing a minimal set of "unsafe" columns, and checking empty intersection with the input and output schemas. The minimal set of unsafe columns is computed using the notion of "migration" of a variable. Consider a program P. A variable x *migrates* (in P) from a set Γ of columns to column D iff for some rule <head> $\leftarrow$ <body> in P, (a) x occurs in <head> in column D, and (b) all occurrences of x in positive literals in <body> are in columns in Γ. A PTIME algorithm to check for weak safety is as follows. First, the attributes where new values are inserted are marked as unsafe. Then, if some x migrates from the set of unsafe columns to a column D, then D is added to the list of unsafe columns. This is iterated until a *minimal* set of unsafe attributes is found. If this set does not intersect the input and output schemas, its complement is a safe-attribute-set; otherwise, the program is not weakly safe.

Before turning to the study of the power of the languages, we make two remarks on the above semantics: one on the connection with logic, and the other on alternative semantics.

Definition: To each rule $A_1,...,A_q \leftarrow B_1,...,B_n$ in P, we associate the first-order sentence

$$\forall\ \vec{x}\ \exists\ \vec{y}\ (\ B_1 \wedge .. \wedge B_n \Rightarrow A_1 \wedge .. \wedge A_q\ )$$

where $\vec{x}$ is the vector of the variables occurring in the body, and $\vec{y}$ is the vector of the variables occurring only in the head. For a program P, let $\Sigma(P)$ be the set of sentences associated with the rules in P.

Remark: given a program P, if $\langle I,J\rangle \in eff_{DL}(P)$ then J is a model of $\Sigma(P)$ containing I.

Remark: The non-standard part of the above definition concerns the handling of new values, which allow the active domain to grow. It should be noted that variables occurring in bodies of rules are valuated *within* the active domain while the others are valuated *outside* the active domain. Variations could be considered such as: variables occurring in heads alone are interpreted freely (i.e., within or outside the active domain); or, all variables are interpreted freely. Such variations are minor, and do not affect the expressive power of the language. A more significant departure would be to fire a rule only if the corresponding sentence is not satisfied. However, we conjecture that the language obtained with this semantics has the same power as DL.

We now study the power of the languages DL, WDL, SDL. It turns out that as long as invented values are provided, the declarative languages have roughly the power of their procedural counterparts. To prove this result, one has to simulate in a declarative language the control mechanisms of procedural languages, using invented values. We also use the results stating that the expressive power of TL is basically not affected if deletions are disallowed (Theorem 1.2).

2.1 The unsafe non-deterministic languages - DL and WDL

We first consider the unsafe case. Our main result is the non-deterministic completeness of DL. To prove it, we use the following definition and notation:

Definition: Let χ be a DL (resp. TL) program and $\vec{x}$ a vector of variables. The expression $\chi(\vec{x})$ is called a DL (resp. TL) *procedure*. If v is a valuation of the variables in $\vec{x}$, then the program obtained by replacing in χ each variable in $\vec{x}$ by its value is denoted $\chi(v\vec{x})$. Two procedures $\chi(\vec{x})$ and $\beta(\vec{x})$ (over <**R**,**S**>) are *equivalent with respect to* <**R**,**S**> iff for each valuation v of the variables in $\vec{x}$, eff(**R**,**S**,$\chi(v\vec{x})$) = eff(**R**,**S**,$\beta(v\vec{x})$).

Notation: For each relational schema Q, let $\overline{Q}$ be a new relational schema with arity($\overline{Q}$) = arity(Q)+1. If ($\neg$) $Q(x_1,...,x_p)$ is a literal and z a variable not occurring in it, then ($\neg$) $Q(x_1,...,x_p)[z]$ denotes the literal ($\neg$) $\overline{Q}(x_1,...,x_p,z)$. For each program P and variable z not occurring in P, P[z] denotes the program obtained from P by replacing each literal A by A[z].
Let P be a program and $B_1,...,B_q$ a list of literals. Then P // $B_1,...,B_p$ is the program obtained by appending $B_1,...,B_n$ to the bodies of all rules in P.

To illustrate the previous notation, consider the program P consisting of the following two rules:

$S(x,y) \leftarrow R(x,y)$

$S(x,y) \leftarrow R(x,z), S(z,y).$

Then P[z] // $\neg$ T(x,w,y) is:

$\overline{S}(x,y,z) \leftarrow \overline{R}(x,y,z), \neg T(x,w,y)$

$\overline{S}(x,y,z) \leftarrow \overline{R}(x,z,z), \overline{S}(z,y,z), \neg T(x,w,y).$

Now we have:

Theorem 2.1: For disjoint input and output schemas[2]
(i) DL is non-deterministic complete; and
(ii) WDL is finitely non-deterministic complete.

Proof: We only provide a proof of (i), since the proof of (ii) is a variation of it. In view of Theorem 1.1, it clearly suffices to show that:

(a) the effect of a DL program is a transformation, and

(b) for each TL procedure $t(\vec{x})$ over <**R**,**S**>, where **R** and **S** are disjoint, there is a DL procedure $prog_t(\vec{x})$ over <**R**,**S**> equivalent to $t(\vec{x})$ w.r.t. <**R**,**S**>.

The proof of (a) is straightforward. We next show (b). The main difficulty is the simulation of the explicit control of the procedural language TL. Intuitively, the non-deterministic nature of DL provides very weak control, which makes it hard to simulate the steps of a TL program in the right order, and to delay the simulation of one step until after the simulation of the previous step has been completed. The simulation is achieved by keeping a *trace* of the computation in auxiliary predicates; errors in the simulation are *eventually* detected by rules which observe the traces. When an error is detected, the program enters an infinite loop. Thus, infinite loops make up for the weak control capability of DL. Note that, even if a TL program always terminates, the DL program simulating it may have non-terminating computations. We later discuss several ways to avoid this drawback using additional constructs (e.g., universal quantification in bodies of rules) which increase the control capability of DL.

We prove (b) by induction on $|t|$ (the length of t). By Theorem 1.2, we may assume without loss of generality that the only elementary instructions of t are insertions, since we assume disjoint input and output schemas. If t consists of a single insert $i_R(\vec{x})$, the DL program "$R(\vec{x}) \leftarrow$" has the desired property. Now suppose that for some n, (b) holds for all t with $|t| < n$. Let $t(\vec{x})$ be a TL procedure over some <**R**,**S**> such that **R** and **S** are disjoint and $|t| = n$. Three cases occur:

(i) $t(\vec{x})$ = with new z do $s(\vec{x},z)$ done,

(ii) $t(\vec{x})$ = while <cond> do $s(\vec{x},\vec{y})$ done, and

(iii) $t = t_1;t_2$.

Since the third case is simpler than the second and requires similar techniques, we only consider the first two cases.

[2] This technical condition is forced by the absence of deletions in DL.

Consider (i). By the induction hypothesis, there is a procedure $prog_s(\vec{x},z)$ equivalent to $s(\vec{x},z)$ w.r.t. <**R**,**S**>. Let *INVENTED* and *NEW* be two new 0-ary, respectively unary, predicates. Let $prog_t$ be the program:

INVENTED , NEW(z) $\leftarrow \neg$ INVENTED

$prog_s$ // NEW(z).

It is easy to see that $prog_t(\vec{x})$ and $t(\vec{x})$ are equivalent w.r.t. <**R**,**S**>.

Now consider (ii). Let $t(\vec{x})$ = while <cond> do $s(\vec{x},\vec{y})$ done. Here $\vec{y}$ is the vector of variables, other than those in $\vec{x}$, occurring in <cond>, i.e., the vector of variables which determine the choice of the valuation for each iteration of s. By the induction hypothesis, there is a procedure $prog_s(\vec{x},\vec{y})$ equivalent to $s(\vec{x},\vec{y})$. Clearly, we would like to use the procedure $prog_s$. However, during an execution of t, s may be called several times. Therefore, stamps (which are invented values) are used to distinguish the different iterations of s performed during a computation. We next describe a program $prog_t$ which simulates t.

The following relations are used:

- five unary predicates $STEP_1$, $STEP_2$, $STEP_3$, $STEP_4$ and DONE;
- for each predicate R used by $prog_s$, two new predicates $\hat{R}$ and $\overline{R}$ such that arity($\hat{R}$) = arity($\overline{R}$)= arity(R) + 1 ($\overline{R}$ corresponds to an extension R[z] of R);
- a binary predicate NextStamp; and
- a predicate COND whose arity is the number of variables occurring in $\vec{y}$ plus one.

The relation NextStamp holds the stamps used to mark the iterations, in order. (The first stamp is always 0.) A tuple COND($\vec{y}$,z) indicates that at iteration z, the valuation $\vec{y}$ of the variables in the condition was chosen. At the end of the computation, a tuple [$\vec{y}$,z] in relation $\hat{R}$ indicates that $\vec{y}$ is in R at the beginning of iteration z; and a tuple [$\vec{y}$,z] in relation $\overline{R}$ indicates that $\vec{y}$ is in R at the end of iteration z. Relations $\hat{R}$ and $\overline{R}$ are used to check that the simulation of the control of t is correct. If the flow of control of t is erroneously simulated at any point, this is detected eventually by rules inspecting $\hat{R}$ and $\overline{R}$, and the program is prevented from terminating.

The program $prog_t(\vec{x})$ is presented next. It is separated in two parts: the first allows the simulation of t for some sequence of *correct* non-deterministic choices; the second detects non-deterministic choices made in the first part which correspond to incorrect computations. In the following, $\vec{u}$ denotes a vector of distinct variables not occurring in $prog_s$ or in $\vec{y}$, and w, z are two

distinct variables not occurring in $\vec{u}$, $\vec{y}$ or $prog_s$.

Rules for valid computations

An iteration consists of three steps (1) copy, (2) choice of a valuation satisfying the condition of the loop, and (3) simulation of the body of the loop.

(0) <u>start-up of first iteration:</u>

$$STEP_1(0) \leftarrow .$$

(1-a) <u>copy for first iteration:</u> for each R in **R**,

$$\hat{R}(\vec{u},0), \overline{R}(\vec{u},0) \leftarrow STEP_1(0), \neg\, STEP_2(0), R(\vec{u}).$$

(1-b) <u>copy for other iterations:</u> for each R occurring in $prog_s$,

$$\hat{R}(\vec{u},z), \overline{R}(\vec{u},z) \leftarrow STEP_1(z), \neg\, STEP_2(z), \overline{R}(\vec{u},w), NextStamp(w,z).$$

<u>switch from (1) to (2):</u>

$$STEP_2(z) \leftarrow STEP_1(z), \neg\, STEP_2(z).$$

(2) <u>choice of a valuation and switch from (2) to (3):</u>

$$COND(\vec{y},z), STEP_3(z) \leftarrow STEP_2(z), \neg\, STEP_3(z), \text{<cond>}[z], \neg\, DONE(0).$$

(3) <u>simulation of the body of the loop for the valuation chosen in (2):</u>

$$prog_s[z] \;//\; STEP_3(z), \neg\, STEP_4(z), COND(\vec{y},z).$$

<u>switch from (3) to (1-b) - start next iteration</u>

$$NextStamp(w,z), STEP_4(w), STEP_1(z) \leftarrow STEP_3(w), \neg\, STEP_4(w)$$

(4-a) <u>Termination:</u>

$$DONE(0) \leftarrow STEP_2(z), \neg\, STEP_3(z)$$

(4-b) <u>Copy of the result:</u> for each S in **S**,

$$S(\vec{u}) \leftarrow DONE(0), STEP_2(z), \neg\, STEP_3(z), \overline{S}(\vec{u},z).$$

Error handling rules

A rule

$$\text{ERROR}(z) \leftarrow \text{ERROR}(x)$$

is used to prevent termination of the program if an error is detected. Note that, with this rule, the program does not terminate as long as a single value is inserted in the ERROR relation. The following rules are used to detect erroneous transfer of control from (1) to (2) (i.e., before the copy stage was completed), from (3) to (1-b) (i.e., before the simulation of the body of the loop was completed), and premature termination at (4-a) (i.e., before the condition of the loop becomes false). Note that the copying of the result in (4-b) is always completed, so this needs not be checked.

- (check 1-a) $\text{ERROR}(0) \leftarrow \text{DONE}(0), R(\vec{u}), \neg \hat{R}(\vec{u},0),$
- (check 1-b) $\text{ERROR}(0) \leftarrow \text{DONE}(0), \text{NextStamp}(w,z), \overline{R}(\vec{u},w), \neg \hat{R}(\vec{u},z).$
- (check 3)

  $$\text{ERROR}(0) \leftarrow \text{DONE}(0), \langle\text{body}\rangle, \text{COND}(\vec{y},z), \neg \alpha$$

  for each <u>safe</u> rule <head> ← <body> in $\text{prog}_s[z]$ such that α is a literal in the head, and

  $$\text{ERROR}(0) \leftarrow \text{DONE}(0), \langle\text{body}\rangle, \text{COND}(\vec{y},z)$$

  for each <u>unsafe</u> rule <head> ← <body> in $\text{prog}_s[z]$.

  The distinction between safe and unsafe rules comes from the fact that a safe rule is saturated if all tuples that can be derived using it are already in the database, whereas an unsafe rule is saturated only if it is not applicable.

- (check 4-a) $\text{ERROR}(0) \leftarrow \text{DONE}(0), \text{STEP}_2(z), \neg \text{STEP}_3(z), \langle\text{cond}\rangle[z].$

To conclude, we sketch a proof that $\text{prog}_t(\vec{x})$ is equivalent to $t(\vec{x})$. Let v be a valuation of $\vec{x}$ and $\vec{a} = v(\vec{x})$. We need to show that $\text{eff}(\mathbf{R},\mathbf{S},t(\vec{a})) = \text{eff}(\mathbf{R},\mathbf{S},\text{prog}_t(\vec{a}))$.

Let (I,I') be in $\text{eff}(\mathbf{R},\mathbf{S},t(\vec{a}))$. By definition, there is a pair of instances (J,J') in $\text{eff}(t(\vec{a}))$ such that $J|_{\mathbf{R}} = I$, J is empty outside **R**, and $J'|_{\mathbf{S}} = I'$. Note that $\text{sch}(s) \subset \text{sch}(t)$; if eff is the effect of a program over sch(s), let $\overline{\text{eff}}$ be the effect extended to sch(t) by leaving unchanged all relations in sch(t) - sch(s). By the semantics of the *while* construct, there are valuations $v_1,\ldots,v_n$ of the variables in $\vec{y}$ and instances $J_0=J,\ldots,J_n=J'$ over sch(t) such that for each i, $(0<i\leq n)$,

$$(J_{i-1},J_i) \in \overline{\text{eff}}(s(\vec{a},v_i\vec{y}))$$

Now consider the DL program $\text{prog}_t(\vec{a})$. First apply (0), then (1-a), then the following instance of (2):

$$\text{COND}(v_1\vec{y},0),\ \text{STEP}_3(0) \leftarrow \text{STEP}_2(0),\ \neg\ \text{STEP}_3(0),\ v_1\text{<cond>}[0],\ \neg\text{DONE}(0).$$

Note that $\hat{R}$ and $\overline{R}$ now contain copies of J_0 marked with the stamp 0. Since $\text{prog}_s(\vec{x},\vec{y})$ and $s(\vec{x},\vec{y})$ are equivalent, $(J_0,J_1) \in \overline{\text{eff}}(s(\vec{a},v_1\vec{y})) = \overline{\text{eff}}(\text{sch}(s), \text{sch}(s),\text{prog}_s(\vec{a},v_1\vec{y}))$. Thus, by construction, some applications of rules in (3) lead to J_1 marked with 0 in $\overline{R}$. The switch from (3) to (1-b) can be fired and a new stamp α is created. Using (1-b), the database state is copied in $\hat{R}$ and $\overline{R}$ with stamp α and the computation can proceed. Finally, a copy of J_n is obtained and (4) fired. It is easy to see that a fixpoint is reached. Clearly, this demonstrates that (J,J') is in eff(sch(t), sch(t), $\text{prog}_t(\vec{a})$), so (I,I') is in eff(**R**,**S**,$\text{prog}_t(\vec{a})$). Thus, eff(**R**,**S**,$t(\vec{a})$) $\subseteq$ eff(**R**,**S**,$\text{prog}_t(\vec{a})$).

Conversely, let (I,I') be in eff(**R**,**S**,$\text{prog}_t(\vec{a})$). Consider a terminating computation of $\text{prog}_t(\vec{a})$ leading from I to I'. Observe first the consecutive states of STEP_1, STEP_2, STEP_3, STEP_4, DONE throughout the computation. The facts derived are, in sequence:

$\text{STEP}_1(0)$, $\text{STEP}_2(0)$, $\text{STEP}_3(0)$, $\text{STEP}_4(0)$,

$\text{STEP}_1(c_1)$, $\text{STEP}_2(c_1)$, $\text{STEP}_3(c_1)$, $\text{STEP}_4(c_1)$,

. . .

$\text{STEP}_1(c_n)$, $\text{STEP}_2(c_n)$,

DONE(0)

for some distinct values (stamps) $c_1,\ldots,c_n$. Observe also that the relation NextStamp contains the tuples $[0,c_1]$ and $[c_i,c_{i+1}]$ for $1 \le i < n$. It is now easy to see that this corresponds to n+1 iterations. Since the program terminates, the error-handling rules were never fired. Thus, the copy stages were all completed, and so were the simulations of s by prog_s. Therefore (I,I') $\in$ eff(**R**,**S**,$t(\vec{a})$) and eff(**R**,**S**,$\text{prog}_t(\vec{a})$) $\subseteq$ eff(**R**,**S**,$t(\vec{a})$). $\square$

The simulation of TL programs by DL programs in the previous proof has the serious drawback of using non-terminating computations to simulate flow of control. In particular, even if a given TL program always terminates, the corresponding DL program simulating it may have non-terminating computations. We next show that this is a limitation of DL which cannot be circumvented without additional constructs. Specifically, we show that there are TL programs which always terminate, and cannot be simulated by *any* always terminating DL program.

Proposition 2.2: Let **R** = {R(A),S(AB)}, **T** = {T(A)}. Then there is a TL program which always terminates on all inputs and computes R - π_A(S) in T. Each DL program with the same effect has some non-terminating computations.

Proof: The following TL program always terminates on all inputs and computes R - π_A(S) in T (without deletions):

while S(x,y), ¬ Q(x) do i_Q(x) done;

while R(x), ¬ Q(x), ¬ T(x) do i_T(x) done.

Let P be a DL program over <**R**,**T**> computing R - π_A(S). Let I be the instance defined by I(R) = {[0]}, and I(S) = ∅. Consider a computation of P on input I which terminates. In this computation, an instantiation r_1 of a rule in P is fired first, then r_2,..., finally some r_n. Observe that since 0 ∈ I(R) - π_A(I(S)), (*) T(0) is derived in $r_1 ... r_n$. Let m be a constant not occurring in $\{r_1,...,r_n\}$. Let J be the instance defined by J(R) = {[0]}, and J(S) = {[0,m]}. It is easy to see that $r_1 ... r_n$ is the beginning of a computation of P on input J. (Since m does not occur in $\{r_1,...,r_n\}$, any positive or negative ground literal that is used in $r_1,...,r_n$ remains valid.) Therefore, by (*), there is a computation $\rho = r_1 ... r_n ...$ of P on input J, which infers the fact T(0). Since 0 is not in J(R) - π_A(J(S)), ρ is infinite. □

We next exhibit two extensions of DL (DL⊥ and DL∀) which can be used to increase its control capability and thus enable simulation of TL without the use of non-terminating computations. Similar extensions will be used in the context of the strongly-safe language SDL, discussed in the next subsection. Intuitively, in DL⊥, an "inconsistent" symbol ⊥ can occur as head of a rule. The idea is that if such a symbol is derived, this particular computation is abandoned. In DL∀ , universal quantification is allowed in bodies of rules. We first present DL⊥ and DL∀ .

DL with inconsistent symbol: DL⊥

The language DL is extended with the symbol ⊥ that can occur only as a literal in the head of rules. A pair (I,J) is in the effect of a DL⊥ program iff J is obtained by a computation where ⊥ is not derived.

DL with universal quantification: DL∀

The language DL is extended to allow rules of the form:

$$A_1,...,A_q \leftarrow \forall\ \vec{x}\ B_1,...,B_n,$$

where $\vec{x}$ is a sequence of variables occurring *only* in the body of the rule. Let $\vec{y}$ be the vector of the variables occurring in $B_1,...,B_n$ and not in $\vec{x}$, and v be a valuation of $\vec{y}$. The rule is fired with valuation v if for each extension $\bar{v}$ of v to the variables in $\vec{x}$ (which valuates variables in $\vec{x}$ in the active domain), $\bar{v}B_1 \wedge..\wedge \bar{v}B_n$ holds.

To illustrate these two languages, we show how to compute the query of Proposition 2.2 with DL$\forall$ or DL$\perp$ programs which always terminate.

Example 2.3: The mapping R - π_A(S) is computed by the following TL$\forall$ program:

$$T(x) \leftarrow \forall\ y\ R(x), \neg\ S(x,y).$$

The mapping R - π_A(S) is computed by the following TL$\perp$ program:

PROJ(x) $\leftarrow \neg$ done_with_proj, S(x,y)

done_with_proj $\leftarrow$

$\perp \leftarrow$ done_with_proj, S(x,y), $\neg$ PROJ(x)

T(x) $\leftarrow$ done_with_proj, R(x), $\neg$ PROJ(x).

More generally, the following can be shown:

Fact: Each TL program t over some i-o schema with disjoint input and output can be simulated by a DL$\forall$ (resp., DL$\perp$) program P without introducing additional non-terminating computations. In particular, if t always terminates, then P always terminates. $\square$

Intuitively, in DL$\forall$·, one can check that a stage is completed (using $\forall$) before proceeding to the next one; and in DL$\perp$, a detected error leads to the derivation of $\perp$ instead of leading to an infinite loop. In Section 4, we consider a last extension of DL - DL* - obtained by allowing negations in heads of rules, interpreted as deletions. As we shall see, DL* can also simulate TL programs without introducing additional loops, but yet with another technique: a log of the updates is kept, and when an error is detected backtracking is performed (using deletes) to the point in the computation where the error occurred. The DL language could also be extended to allow set-valued entries in relations in the spirit of the complex objects of [ABe]. In this context again, TL programs could be simulated without using additional non-terminating computations: intuitively, the universal quantification construct of DL$\forall$ can be simulated using set equality. Finally, we note

that DL itself can simulate TL without non-terminating computations on *ordered* databases, i.e. databases where a successor function on the active domain of the database is provided. Again, universal quantification can be simulated by an exhaustive search performed using the successor function.

2.2 The strongly-safe non-deterministic language - SDL

The symmetry between procedural and declarative languages breaks down in the strongly-safe case. Indeed, the strongly-safe declarative language - SDL - is weaker than its procedural counterpart - STL. By Theorem 1.1, STL expresses exactly the DB-NPSPACE transformations. This is in part due to deletions, which are allowed in STL. Clearly, one cannot expect the same power from a language with inflationary semantics. (As we shall see in Section 4, SDL extended with deletions in heads of rules - SDL* - does have the power of STL.) Indeed, it is easy to see that each SDL transformation is in DB-NPTIME. It turns out that there are simple DB-NPTIME transformations that cannot be expressed in SDL even for disjoint input and output schemas. We show this next, and then show that SDL augmented with the $\perp$ or $\forall$ constructs defined in the previous subsection computes exactly DB-NPTIME. The precise characterization of the power of SDL itself is still open.

We first consider SDL and show, in particular, that it is strictly weaker that DB-NPTIME even over disjoint input and output schemas.

Proposition 2.4: For disjoint input and output schemas, the set of transformations computable by SDL strictly contains the datalog mappings and is strictly included in DB-NPTIME.

Proof: The strict containment of the Datalog mappings is obvious. The inclusion in DB-NPTIME follows from the "inflationary" character of SDL and the fact that only a number of tuples polynomial in the size of the database can be derived. The proof that the inclusion is strict is similar to the proof of Proposition 2.2: by the same argument, one shows that $R - \pi_A(S)$ cannot be computed by an SDL program. $\square$

Note that from the previous proof, SDL is not "closed under composition" since $R - \pi_A(S)$ can be obtained as the composition of the mappings defined by the following two rules:

$Q(x) \leftarrow S(x,y)$, and

$T(x) \leftarrow R(x), \neg Q(x).$

As seen above, there are very simple transformations that SDL cannot compute. We now show how this weakness of SDL can be corrected. More precisely, we consider the languages SDL$\perp$ and SDL$\forall$ obtained from SDL by adding, respectively the $\perp$ and $\forall$ constructs defined earlier. We will show that both languages compute exactly DB-NPTIME. To prove this, we will study first the power of SDL on "ordered" databases, mentioned informally at the end of the previous subsection.

Definition: An *ordered database instance* is an instance over a database schema containing a binary relation NEXT and two unary relations MIN and MAX, such that

(i) every value occurring in the instance also occurs in NEXT, and

(ii) NEXT = $\{ [a_i,a_{i+1}] \mid 1 \leq i < n \}$ for some finite sequence $a_1,a_2,\ldots,a_n$ of distinct values , MIN = $\{[a_1]\}$, and MAX = $\{[a_n]\}$.

The effect of a program on ordered instances is the restriction of the effect to ordered instances only.

Different variations of the notion of ordered database can be considered. One might use an order relation ($\leq$) instead of NEXT and/or not know in advance the minimum and maximum elements of the order. Proposition 2.7 below may not be true for all these variations.

Our first result concerns ordered instances. We will show that, on ordered instances, SDL computes the DB-NPTIME transformations. To prove it, we use one lemma concerning STL and one dealing with the simulation of STL by SDL. Recall that a TL program *runs in NPTIME* if on each input, each computation stops after a number of steps polynomial in the size of the input, a step being the execution of a tuple insertion or deletion (regardless of whether the tuple is actually inserted/deleted.) A similar definition can be given for DL, a step in a DL execution being a tuple insertion.

Now we have:

Lemma 2.5: Let STL$^+$ be the language consisting of all STL programs without deletions or erases. For disjoint i-o schemas, the following are equivalent: (i) τ is a DB-NPTIME transformation, and (ii) τ is the effect of an STL$^+$ program.

Proof: (Sketch) By its inflationary nature, every STL^+ program runs in NPTIME, so (ii) implies (i). Conversely, let τ be a transformation in DB-NPTIME over disjoint i-o schemas. By Theorem 1.1, STL computes the DB-NPSPACE transformations. By inspection of the proof of this result in [AV2], one can see that the simulation by STL of a Turing machine corresponding to a DB-NPTIME transformation also runs in NPTIME. Thus, there is an STL program computing τ, which runs in NPTIME. Next, by Theorem 1.2, there is an equivalent TL program t without deletions or erases (but with invented values). By construction, the TL program also runs in NPTIME. In particular, every computation uses a number of invented values bounded by a polynomial, say n^k, in the size n of the active domain of the input database. Finally, we will show that the invented values are not necessary, since the n^k invented values can be simulated using k-tuples formed using the n constants available in the active domain.

Two k-ary relations, $VALUE_k$ and OLD_k are used to keep track of the current invented value. (The last value used is in $VALUE_k$ - OLD_k.) The simulation proceeds as follows:

(i) prefix t by a program constructing non-deterministically in a binary relation NEXT an order of the active domain of the database, and storing the minimum (maximum) elements min (max) in two unary relations MIN (respectively, MAX).

(ii) to simulate t without invented values, each relation R of t is extended to accommodate the k-tuples representing invented values. Let m be the value in MIN. For each R in t, let R_k be a new relation with $arity(R_k) = arity(R) \cdot (k+1)$. Intuitively, each attribute A of R corresponds to k + 1 attributes $A, A_1,...,A_k$ in R_k. A tuple u over R is represented by a tuple u_k over R_k as follows: for each A

- if u(A) is not an invented value, $u_k(A) = u(A)$ and for each i, $u_k(A_i) = m$.
- if u(A) is an invented value simulated by the k-tuple the tuple $\vec{a}_k$, $u_k(A) = m$ and $u_k(A_1...A_k) = \vec{a}_k$.

(The k-tuple with all values m is not used to represent an invented value, but as a marker for non-invented values.)

To simulate t, the R_k corresponding to each R in the input schema is first constructed. Then t is simulated with k-tuples in place of invented values. Each time a new invented value is needed, the successor (in the lexicographic order induced on k-tuples by NEXT) of the current invented value is computed and used in place of the invented value. $VALUE_k$ and OLD_k are updated so that $VALUE_k$ - OLD_k contains the value currently used. Finally, each output relation S is decoded from the corresponding S_k.

Thus, t can be simulated without invented values, so τ is computed by an STL^+ program. □

Lemma 2.6: For each STL^+ program over disjoint i-o schemas, there is an SDL program equivalent to it on ordered instances.

Proof: (Sketch) Let t be an STL^+ program over disjoint i-o schemas. Since t is in particular a TL program, by Theorem 2.1, there is a DL program P equivalent to it. However, the DL program has non-terminating computations, and uses invented values in the simulation. To prove the lemma, we show that, on ordered database instances:

(i) the construction of the DL program corresponding to t can be modified so that the control of t is simulated without introducing non-terminating computations, and

(ii) invented values are not needed.

We first show (i). Consider the construction of the DL program P corresponding to t, in the proof of Theorem 2.1. To avoid using non-terminating computations in the simulation, we have to make the switch from (1) to (2) and (3) to (1) more complex. Intuitively, a switch is enabled if the previous stage is completed, i.e. a fixed-point has been reached for the subprogram corresponding to the previous stage. We show how to enforce the correct switch from (3) to (1). (The other is treated similarly.) Let $\{r_i \mid i \text{ in } [1..n]\}$ be the set of rules of (3). For each i, let k_i be the number of variables occurring in r_i. Intuitively, we will first construct, for each i, an ordered list of all possible valuations of k_i variables in the active domain. To check that a fixpoint for $\{r_i \mid i \text{ in } [1..n]\}$ has been reached, one starts by verifying that r_1 can no longer be applied and produce new tuples. This is done by checking all possible valuations for the variables in r_1, in order; r_1 is fired if possible, and the process re-starts. If r_1 cannot be fired for any valuation, the process continues for r_2. Eventually the process terminates at r_n. Then stage (3) is completed and the switch to (1) is enabled. Note that, since the process can be re-started several times, the different trials must be distinguished using an invented value. However, termination is guaranteed after polynomially many re-starts, since at each trial at least one new tuple with values from the active domain of the input database is inserted.

We omit the details and only describe the construction of the ordered list of valuations for k given variables.

The ordered list of valuations for k variables is constructed in a relation $NEXT_k$ of arity 2k. The list is constructed starting from the given relations NEXT, MIN, and MAX, which order the active domain of the input database. We use the rules:

$$NEXT_k(\vec{x}_{k-j-1},y,\vec{M}_j,\vec{x}_{k-j-1},z,\vec{m}_j) \leftarrow NEXT(y,z),\ NEXT_k(\vec{y}_k,\vec{x}_{k-j-1},y,\vec{M}_j),\ MIN(m),\ MAX(M)$$

for each j, $0 \le j < k$, where

- $\vec{y}_k$ is a vector of k distinct,
- $\vec{x}_{k-j-1}$ is a vector of k-j-1 distinct variables,
- M and m are two new distinct variables and $\vec{M}_j$ (resp., $\vec{m}_j$) is a j-vector where all entries are M (resp., m).

Note that it is easy to detect when $NEXT_k$ has been completely computed: this occurs when "$NEXT_k(\vec{b}_k,\vec{M}_k)$, MAX(M)" holds for some k-vector $\vec{b}_k$. This can be used to delay the remainder of the computation until after $NEXT_k$ has been constructed.

Finally, consider (ii). Note that the DL program obtained in (i) always terminates, and uses a number of invented values polynomial in the size of the input database. The simulation of the polynomial number of invented values using vectors of values from the active domain is similar to that in the proof of Lemma 2.5, with the distinction that relations NEXT, MIN, and MAX are provided and thus do not have to be constructed. □

From Lemmas 2.5 and 2.6, we have:

Proposition 2.7: On ordered instances and for disjoint input and output schemas, SDL computes exactly the DB-NPTIME transformations. □

Finally, we use the above result to show that SDL augmented with the construct $\perp$ or $\forall$ computes all DB-NPTIME transformations over disjoint i-o schemas.

Theorem 2.8: Let <**R**,**S**> be an i-o schema with **R** and **S** disjoint, and τ a transformation from inst(**R**) to inst(**S**). The following are equivalent:

- τ is a DB-NPTIME transformation,
- τ is the effect of an SDL$\perp$ program over <**R**,**S**>, and
- τ is the effect of an SDL$\forall$ program over <**R**,**S**>.

Proof: In view of Proposition 2.7, it suffices to show that SDL$\perp$ and SDL$\forall$ programs can construct relations NEXT, MIN, MAX provided in ordered databases, and delay the firing of other rules until after these relations have been constructed.

First consider SDL$\forall$. For simplicity, we assume that the active domain of the input database does not consist of a single value (the special case of the one-constant domain is dealt with by minor additions to the program that we exhibit). The program uses two unary predicates, *ordered*

and *old*, and one 0-ary predicate *start*. Intuitively, a value in the active domain is in *ordered* if its rank in the total order that is constructed has been chosen. During the construction of NEXT, a unique value is in *ordered* - *old*: the "greatest" ordered value so far.

For each input predicate R, and each i, $1 \leq i \leq$ arity(R), the program contains the following rules:

(1) $MIN(x_i)$, $ordered(x_i)$, start $\leftarrow P(...x_i...)$, $\neg$start

(2) $NEXT(z,x_i)$, $ordered(x_i)$, old(z) $\leftarrow P(...x_i...)$, $\neg$ $ordered(x_i)$, ordered(z), $\neg$old(z)

(3) MAX(z) $\leftarrow \forall$ y [ordered(y), ordered(z), $\neg$old(z)].

Note that rule (3) is fired only when all y have been ordered. Thus the condition MAX(m) signals the completion of the ordering procedure and can be used to delay the remainder of the computation.

Now consider SDL$\underline{\perp}$. Consider the program obtained by replacing in the above program rule (3) by the two rules:

(3-a) MAX(z), old(z) $\leftarrow$ ordered(z), $\neg$old(z)

(3-b) $\underline{\perp} \leftarrow$ MAX(z), $P(...x_i...)$, $\neg ordered(x_i)$

Again, condition MAX(m) signals the completion of the computation. However, there is a possibility that rule (3-a) was fired too early, i.e. before all values were "ordered". If such is the case, this is detected eventually by rule (3-b) which invalidates the computation. Thus, correct computations yield complete orderings in NEXT. $\square$

3. DETERMINISTIC LANGUAGES

In this section, we define several deterministic counterparts of the non-deterministic languages investigated in the previous section. These languages, detDL and SdetDL, are declarative analogs of the procedural languages detTL and SdetTL. In particular, SdetDL (which we also denote Datalog$^{\neg}$) is of particular interest since it is a strongly safe, deterministic language which provides a new, appealing semantics for Datalog with negation.

3.1 A deterministic-complete language

In this subsection, we discuss the syntax and semantics of the deterministic language detDL, and prove that it is deterministic complete. One could use the same syntax for the deterministic languages as for their non-deterministic counterparts. However, due to the deterministic semantics

that we shall describe, the syntax can be simplified without loss of power (this is explained further after the definition of the deterministic semantics). Specifically, no multiple heads and no explicit equality are required in the deterministic case. In order to obtain deterministic semantics, we also place the syntactic restriction that programs be weakly safe (so, invented values are not allowed in the result). Thus, the syntax of detDL is a special case of the syntax of WDL. More precisely, we have:

Definition: A *Deterministic Declarative Language (detDL)* program over an i-o schema **<R, S>** is a WDL program over **<R, S>** such that:

(i) all heads of rules consist of single literals, and

(ii) equality does not occur in the program.

Note that the above syntax is that of Datalog augmented with negation in bodies of rules, and variables which may occur in heads of rules without occurring in their bodies.

We next discuss the semantics for detDL programs. The semantics is analogous to that of detTL. As in the procedural case, we use a "parallel" saturation semantics. Specifically, each iteration of the program adds all facts which can be inferred by considering *simultaneously* all possible instantiations of the rules. However, for each instantiation of the variables in the body of a rule, only *one* extension assigning invented values to the variables occurring only in the head of the rule is considered. Furthermore, the simultaneous instantiations must observe the restriction that the values "invented" by the different instantiations are distinct. This is repeated until no new facts can be added to the database. (Termination is not guaranteed). Clearly, the above semantics retains a non-deterministic aspect, due to the arbitrary choice of invented values. However, the non-determinism affects only temporary relations. To define the semantics of detDL, we use the auxiliary notion of "valuation-set", which concerns instantiations of the variables in rules.

Definition: Let P be a detDL program and I an instance over sch(P). A *valuation-set for* P and I is a mapping Γ such that

(1) the domain of Γ is the set of all pairs (r,v) where $r = A \leftarrow B_1,\ldots,B_n$ is a rule in P, and v a valuation of the variables occurring in the body of r within the current active domain, such that $I \models vB_1 \wedge .. \wedge vB_n$;

(2) for each (r,v) in the domain of Γ, $\Gamma(r,v)$ is a valuation which extends v to the variables occurring only in the head of r, such that these variables are mapped to distinct values not occurring in I or P; and

(3) for each (r,v), (r',v') in the domain of Γ, (r',v') $\neq$ (r,v), and each variable z (resp. z') occurring in the head of r (resp. r') and not in its body, $\Gamma(r,v)(z) \neq \Gamma(r',v')(z')$.

Intuitively, each valuation-set for P and I provides *one* extension for each possible instantiation v of the variables in the body of a rule of P, such that variables occurring in heads of rules alone are assigned *distinct new* values.

We now have:

Definition: The *effect* eff(P) of a detDL program P is the binary relation on inst(sch(P)) defined by: <I,J> is in eff(P) iff there exists a sequence $I_0,\ldots,I_{p+1}$ of instances over inst(sch(P)) such that

(a) $I_0 = I$, $I_{p+1} = I_p = J$, and

(b) for each $i \leq p$, there is a valuation-set Γ_i for P and I_i such that

$$I_{i+1} = I_i \cup \{\Gamma_i(r,v)(A) \mid (r,v) \in \mathrm{dom}(\Gamma_i), A \text{ is the head of } r\}.$$

The effect eff(**R**,**S**,P) of a detDL program P over an i-o schema <**R** , **S**> is the binary relation from inst(**R**) to inst(**S**) such that (I,J) $\in$ eff(**R**,**S**,P) if there exists <I',J'> in eff(P) such that I' agrees with I on **R** and is empty everywhere else, and J' agrees with J on **S**.

The following fact can easily be verified:

Fact: The *effect* eff(**R**,**S**,P) of a detDL program P over an i-o schema <**R** , **S**> is a mapping, that is, the semantics is indeed deterministic.

Intuitively, the determinism is due to the fact that the different possible valuation-sets for given P and I are isomorphic and agree on the active domain. Furthermore, the relations of the i-o schema always contain just constants from the active domain. Thus, the choice of one valuation-set over another does not affect the final result, although it does introduce the appearance of non-determinism in the computation. We shall denote by eff(**R**,**S**,P)(I) the image of an instance I.

By definition of the semantics of detDL, the evaluation of a detDL program P on an instance I consists of several iterations (each of which consists of firing simultaneously all rules with all valid valuations of the bodies). One such iteration is also referred to as a *stage* in the evaluation of P on I. In particular, each terminating evaluation has a last stage, which is the last iteration resulting in the addition of new tuples in the database.

Remark: For each detDL program P over an i-o schema <**R**,**S**>, let $\Sigma(P)$ be the set of sentences associated with P, (as in Definition, Section 2). If $J = eff_{detDL}(\mathbf{R},\mathbf{S},P)(I)$, then J is the restriction to S of a model of $\Sigma(P)$.

The definition of detDL *procedures* is analogous to that of DL procedures.

A strongly safe restriction of detDL is obtained by requiring that all variables occurring in the head of a rule also occur in the body. The resulting language is *strongly-safe deterministic DL* (SdetDL). The same language has been independently proposed in [KP] under the name Datalog$^{\neg}$.

Remark: As noted earlier, under deterministic semantics, the syntax used for DL is no more powerful than the simpler syntax used for detDL. To see that explicit equality can be eliminated, note that a predicate $EQ = \{<c,c> \mid c$ occurs in the database or in the program$\}$ can be constructed in a first stage; the computation of the program can be easily delayed until after EQ has been constructed. Next, multiple-head rules of the form

$$A_1,\ldots,A_k \leftarrow B_1,\ldots,B_n$$

can be simulated by k single-head rules

$$A_i \leftarrow B_1,\ldots,B_n$$

if no variable corresponding to an invented value occurs several times in the head. If such a variable exists, then the invented value is first created using a single-head rule and the firing of all other rules is delayed by one step, at which time the invented value can be used as a regular database value. Note that the above simulations cannot be carried out with the non-deterministic semantics. □

We now study the expressive power of detDL.

Theorem 3.1: For disjoint input and output schemas, detDL is deterministic complete.

Proof: As in in the case of DL (see Theorem 2.1), the proof of completeness for detDL involves a simulation of its procedural counterpart detTL, which is known to be deterministic complete. Furthermore, we will use the fact that detTL remains deterministic complete over disjoint input and output schemas when deletions are disallowed (see Theorem 1.2).

We will show that every detTL program (over disjoint input and output schemas and without deletions) can be simulated by a detDL program. In order to prove this, we will show by induction the following:

(*) for each detTL procedure $t(\vec{x})$ without deletions, there exists a detDL procedure $prog_t(\vec{x})$ such that:

(1) $sch(t) \subset sch(prog_t)$, $sch(prog_t)$ contains a special 0-ary predicate $done_t$,

(2) for each valuation v of the variables in $\vec{x}$, $eff(t(v\vec{x}))$ is the restriction of $eff(prog_t(v\vec{x}))$ to sch(t), and

(3) for each instance I over sch(t), $done_t$ becomes true only at the last stage of the evaluation of $prog_t$ on I.

The main difference with the simulation of TL by DL (Theorem 2.1) is that, due to the deterministic semantics which allows more accurate timing, simulation of control can be achieved without introducing additional non-terminating computations. We now provide the simulation. For conciseness, we allow rules with several literals in heads, which can be easily converted to rules with single-literal heads (see earlier remark).

Let $t(\vec{x})$ be a detTL procedure without deletions. The base case is obvious and is omitted. Suppose the induction hypothesis holds for each detTL procedure of length less than $t(\vec{x})$. There are three cases to consider:

(i) $t(\vec{x}) = t'(\vec{y});t''(\vec{z})$. Then $prog_t$ is the following:

$prog_{t'}$

$prog_{t''}$ // $done_{t'}$

$done_t \leftarrow done_{t''}$.

(ii) $t(\vec{x})$ = with new z do $s(\vec{y})$ done. Then $prog_t$ consists of:

INVENTED , NEW(z) $\leftarrow$ ¬INVENTED

$prog_s$ // NEW(z)

$done_t \leftarrow done_s$.

where INVENTED and NEW are two new predicates.

(iii) $t(\vec{x})$ = while <cond>$(\vec{y})$ do $s(\vec{x},\vec{y})$.

Here $\vec{y}$ contains the variables in <cond> but not in $\vec{x}$. By the induction hypothesis, there exists a detDL procedure $prog_s(\vec{x},\vec{y})$ corresponding to $s(\vec{x},\vec{y})$ and satisfying (1)-(3). The detDL procedure $prog_t(\vec{x})$ corresponding to $t(\vec{x})$ is given below. Informally, an iteration (identified by a stamp α) is simulated as follows: first, all valuations $\vec{y}$ satisfying <cond> at the beginning of iteration α are

stored in relation COND($\vec{y}$,α); next, program $prog_s$ is run for each such valuation $\vec{y}$. The end of the iteration α is detected using three relations, *try-end-iteration, maybe-end-iteration,* and *not-end-iteration.* Each time the simulation of s for a given valuation $\vec{y}$ is completed, a trial marked with β (recorded in *try-end-iteration*(α,β)) is carried out to check if the entire iteration has been completed, i.e. the simulations of s on *all* valuations $\vec{y}$ in COND($\vec{y}$,α) have been completed. If for some valuation, the simulation has not yet been completed, not-end-iteration(α,β) becomes true. Thus, the end of the iteration is detected when, for a trial β, not-end-iteration(α,β) does not become true. At that point, the results of the iteration for the different valuations are collected and the condition of the loop re-evaluated. Finally, the end of the simulation of the loop is detected by a technique similar to that for detecting the end of an iteration, using three relations *try-end-loop, maybe-end-loop, not-end-loop.* When the end of the simulation of the loop is detected, the 0-ary predicate $done_t$ becomes true. We assume without loss of generality that the auxiliary predicates used in $prog_t$ are new, i.e. they do not occur in $prog_s$. The description of $prog_t$ follows.

<u>initialization:</u> find all valuations $\vec{y}$ satisfying <cond>($\vec{y}$) and prepare input for iteration 0 (i.e., mark all relations with the valuations $\vec{y}$ and stamp 0).

initialized, COND($\vec{y}$, 0) $\leftarrow$ <cond>($\vec{y}$), $\neg$initialized

initialized, $\overline{R}$($\vec{z}$, $\vec{y}$, 0) $\leftarrow$ R($\vec{z}$), <cond>($\vec{y}$), $\neg$initialized

for each R in sch($prog_s$)

<u>iteration α:</u> call $prog_s$ with stamp α for each $\vec{y}$; when the computation of $prog_s$ for $\vec{y}$ terminates, check whether iteration α is over.

$prog_s$[$\vec{y}$, α] // COND($\vec{y}$, α), $\neg$OLD(α)

try-end-iteration(α,β) $\leftarrow$ $\overline{done_s}$($\vec{y}$, α), $\neg$OLD(α)

maybe-end-iteration(α,β) $\leftarrow$ try-end-iteration(α,β)

not-end-iteration(α,β) $\leftarrow$ try-end-iteration(α,β), COND($\vec{z}$,α), $\neg\overline{done_s}$($\vec{z}$, α)

done-iteration(α) $\leftarrow$ maybe-end-iteration(α,β), $\neg$not-end-iteration(α,β)

<u>loop control:</u> at the end of an iteration, collect the results and check whether this is the end of the loop.

R($\vec{z}$) $\leftarrow$ done-iteration(α), $\neg$OLD(α), $\overline{R}$($\vec{z}$, $\vec{y}$, α) for each R in sch($prog_s$),

try-end-loop(α) $\leftarrow$ done-iteration(α), $\neg$OLD(α)

maybe-end-loop(α) $\leftarrow$ try-end-loop(α)

not-end-loop(α) $\leftarrow$ <cond>($\vec{y}$), try-end-loop(α)

$done_t \leftarrow$ maybe-end-loop(α), $\neg$not-end-loop(α)

<u>prepare next iteration:</u> if the end of the loop was not detected, prepare the next iteration γ. In particular, record valuations $\vec{y}$ satisfying <cond>($\vec{y}$) at beginning of iteration γ and prepare input for the iteration (i.e., mark all relations with the valuations $\vec{y}$ and stamp γ).

OLD(α), new-iteration(γ) $\leftarrow$ not-end-loop(α), $\neg$OLD(α)

COND($\vec{y}$, γ) $\leftarrow$ <cond>($\vec{y}$), new-iteration(γ), $\neg$done-iteration(γ)

$\overline{R}(\vec{z}, \vec{y}, \gamma) \leftarrow \neg done_t$, R($\vec{z}$), <cond>($\vec{y}$), new-iteration($\gamma$), $\neg$done-iteration(γ)

By inspection, one can verify that (a) $prog_t$ is weakly safe (invented values are entered in particular columns) and (b) $prog_t$ satisfies (1)-(3). Thus, the induction is complete. □

As seen above, the simulation of detTL by detDL can be achieved without the use of infinite loops for control, unlike the simulation of TL by DL programs. Intuitively, this is due to the additional control available in detDL which comes from the fact that *all* ground instances of rules (modulo invented values) must be applied at the same time.

The strongly-safe restriction of detDL is SdetDL (Datalog$^{\neg}$). The semantics is defined as for detDL, although some simplifications are possible due to the restricted form of the language. Indeed, we will see in Section 5 that the semantics of SdetDL can be defined easily using an inflationary fixpoint operator. We will show there that the transformations corresponding to SdetDL programs are the fixpoint queries (*FP*). As a side effect, we obtain results on inflationary fixpoint logic. We postpone the study of SdetDL to Section 5, where the connection between our languages and fixpoint extensions of first-order logic is discussed at length.

The results on the simulation of procedural languages by extensions of Datalog provide, as a side effect, a normal form for the procedural languages. This is described next.

Remark 3.2: We have shown that the procedural languages TL and detTL can be simulated by DL and detDL, respectively. Conversely, it is easy to see that each DL (detDL) program can be simulated by a TL (detTL) program consisting of one main *while* loop whose body contains only insertions, deletions, and statements of the form *if* <*cond*> *then* <*body*>, where <*body*> contains only insertions and deletions. (The *if-then* construct is used in [AV2]; we did not consider it here since it does not modify the power of TL or detTL.) This provides a normal form for (det)TL

programs: each (det)TL program is equivalent to a (det)TL program consisting of a *while* statement with no inner *while*, but possibly with inner *if-then* statements. Note that normal forms also hold for the safe languages STL and SdetTL (which are shown in Section 4 to be equivalent to Datalog extensions with deletions).

4. NEGATIONS IN HEADS OF RULES

In this section, we consider extensions of the languages considered so far, which allow for negative literals in heads of rules. The negative literals are interpreted as tuple deletions. We consider such extensions for two reasons. First, a construct allowing explicit deletions appears desirable in an update language. Thus, such a construct provides variations of our Datalog extensions, which are more update oriented. Second, the extended languages are interesting with respect to expressive power. Indeed, with the deletion construct, the Datalog extensions have precisely the same expressive power as their procedural counterparts.

We first consider extensions DL* and SDL* of the non-deterministic languages (DL and SDL), then extensions detDL* and SdetDL* of the deterministic ones (detDL and SdetDL).

4.1 Non-deterministic languages with negations in heads

In this section, we show the non-deterministic completeness of DL*. We also prove that SDL* yields the DB-NPSPACE transformations, i.e., has the same expressive power as STL.

We now consider the extension DL* of DL, which allows negative literals in heads of rules. Earlier, we proved that DL is non-deterministic complete for *disjoint* i-o schemas. We will show that (not surprisingly) DL* is non-deterministic complete for all i-o schemas. First, we present the syntax of DL*.

Definition: A *Declarative Language* (DL*)* program is a finite set of rules of the form

$$A_1,\ldots,A_q \leftarrow B_1,\ldots,B_n$$

($q > 0$, $n \geq 0$) where each A_i is a <u>positive or negative</u> literal of the form ($\neg$) $Q(x_1,\ldots,x_m)$ ($m \geq 0$), and each B_i is a positive literal of the form $Q(x_1,\ldots,x_m)$ ($m \geq 0$) or $x_1 = x_2$ (the x_i's are domain variables or constants), or the negation of such a literal.

The semantics of DL* is similar to that of DL. As for DL, rules are fired non-deterministically. When a rule is fired, a positive literal in the head is interpreted as an insertion

and a negative one as a deletion. Additionally, the ground rule which is fired must be "consistent", i.e., it may not require both the insertion and deletion of the same tuple. Formally, the definition of eff_{DL*} is like that of eff_{DL} except that (1-c) is replaced by:

(1-*) $I_{k+1} = \varepsilon_1 \ldots \varepsilon_q(I_k)$ where for each j, $1 \leq j \leq q$, $\varepsilon_j = i_Q(vx_1,\ldots,vx_m)$ if $A_j = Q(x_1,\ldots,x_m)$; $\varepsilon_j = d_Q(vx_1,\ldots,vx_m)$ if $A_j = \neg Q(x_1,\ldots,x_m)$; and
if $Q(x_1,\ldots,x_m)$ and $\neg Q(y_1,\ldots,y_m)$ both occur in $A_1,\ldots,A_q$, $vx_j \neq vy_j$ for some j.

Remark: The above definition does allow a negative literal in the head of a rule to contain some variable not occurring in the body of the rule. However, such a negative literal is vacuous, since it can never result in a deletion being performed.

Remark: For each DL* program P, let $\Sigma(P)$ be the set of sentences associated with P (as in Definition, Section 2, extended in the obvious way to allow for negative literals in the heads). If $\langle I,J\rangle \in eff_{DL*}(P)$, then J is a model of $\Sigma(P)$.

Now we have:

Theorem 4.1: DL* is non-deterministic complete.

Proof: Clearly, the effect of every DL* program is a transformation. Conversely, let τ be a transformation over an i-o schema $\langle \mathbf{R},\mathbf{S}\rangle$ (with **R**, **S** not necessarily disjoint). For each attribute A occurring in **S**, let A_c be a new attribute. Let $\mathbf{S}_c$ be obtained from **S** by substituting each A by A_c. For each instance J over **S**, let J_c be the corresponding instance over $\mathbf{S}_c$. Consider the transformation τ_c over $\langle \mathbf{R},\mathbf{S}_c\rangle$ defined by $\tau_c = \{\langle I,J_c\rangle | (I,J) \in \tau\}$. Clearly, τ_c is a transformation over $\langle \mathbf{R},\mathbf{S}_c\rangle$, i.e., over an i-o schema with disjoint input an output schemas. By Theorem 2.1, there is a DL (so a DL*) program P_c over $\langle \mathbf{R},\mathbf{S}_c\rangle$, with effect τ_c. The computation of τ by a DL* program proceeds in two phases: (1) given some input I, run P_c and obtain an instance J_c over S_c, and (2) copy J_c to the relations of the output schema **S**. The only difficulty of the proof is to guarantee that the first phase is completed before the second starts. Like in the proof of Theorem 2.1, we can let the second phase start too early and use error handling rules based on non-terminating computations. (A more complicated proof not based on non-terminating computations is also possible. See Remark 4.2 below.) To simplify the presentation, we assume that **S** consists of a single relation S. We use two temporary relations S_i and S_d of the same arity as S. Intuitively, a tuple in S_i (respectively S_d) indicates that during the second phase, a tuple has been *effectively* inserted in (deleted from) S. These relations serve to recover the value of S at the

end of phase one, and allow the detection of errors arising from premature switches from phase one to phase two. The following rules are used:

P_c // $\neg$ phase2

phase2 $\leftarrow$

$S(\vec{x}), S_i(\vec{x}) \leftarrow S_c(\vec{x}), \neg S(\vec{x})$, phase2

$\neg S(\vec{x}), S_d(\vec{x}) \leftarrow \neg S_c(\vec{x}), S(\vec{x})$, phase2.

An error is detected (and leads to an infinite loop) if the instance at the end of phase one is not saturated w.r.t. P_c. Observe that phase two only changes S_d , S_i (originally empty) and S. Note that, at any point after the switch from phase one to phase two, $S - S_i \cup S_d$ equals the value of S at the end of phase one. It is clear thus that error handling can be achieved in DL (and so in DL*). $\square$

As mentioned in the beginning of the section, deletions provide additional control capability in our languages. This is discussed next.

Remark 4.2: In the simulation of TL by DL (for disjoint input and output), and in the previous proof, non-terminating computations were introduced to simulate explicit control. This was done by allowing non-deterministic transfer of control from one step to the next, then using error handling rules to detect premature transfers and prevent termination of the program. The deletions available in DL* provide the added capability to *correct* errors once detected, using a roll-back mechanism, rather than entering an infinite loop. Intuitively, separate journals of the updates are maintained for each point where an error can potentially occur. Each journal consists of an *ordered* record of all tuple insertions and deletions *effectively* performed. (The representation of the order is similar to that used in relation $NEXT_k$ in the proof of Lemma 2.6.) When an error is detected (i.e., it is determined that a transfer of control was performed too early), the corresponding journal is used to recover the state of the database at the time of the error. This is done by traversing the journal of updates in reverse order, undoing each update and removing it from the journal. Note that the completion of the roll-back can be enforced and detected due to the fact that the journal keeps records of updates *in order*. When the roll-back has been completed, the computation is re-started, with the transfer causing the error disabled for one step. (This guarantees termination.) This roll-back technique can be used to show:

Fact: Each TL program t can be simulated by a DL* program P without introducing additional

non-terminating computations. In particular, if t always terminates, then P always terminates. □

We next consider SDL*, i.e., a safe and non-deterministic language with deletions. In the unsafe case, the issue of the expressive power of DL* is almost vacuous, since DL is complete to start with (for disjoint i-o schemas). The safe case is more interesting, since the expressive power of SDL stops short of DB-NPTIME, and so SDL is much weaker than its procedural counterpart STL, which expresses DB-NPSPACE. It can therefore be expected that SDL* provides more power than SDL. Indeed, we next show that SDL* can compute exactly DB-NPSPACE, so it has precisely the same expressive power as STL. Before turning to the proof, we note a key difference between SDL* and the safe languages previously encountered, which is due to its capability to delete tuples. An SDL* computation does not always terminate, as shown by the following simple program:

$A \leftarrow \neg A,$

$\neg A \leftarrow A.$

Theorem 4.3: The set of transformations expressible in SDL* is DB-NPSPACE.

Proof: (Sketch) Clearly, each transformation corresponding to an SDL* program is in DB-NPSPACE because of the safety. By Theorem 1.1, it therefore suffices to show that each STL program can be simulated by an SDL* one. Recall that SDL can simulate STL^+ *on ordered instances* (Lemma 2.6). A straightforward variation of the same proof shows that SDL* can simulate STL on ordered instances. Thus an STL program P can be simulated by an SDL* program as follows. First, the SDL* program computes an ordering of the active domain. Next, P is simulated on an ordered instance. The necessary control to guarantee that the first phase is completed before the second starts is provided using infinite loops. (To avoid the use of non-terminating computations, see Remark 4.4 below.)

The construction of the ordering is similar to that used for $SDL_{\underline{1}}$ (Theorem 2.8):

(1) $MIN(x_i), ordered(x_i), start \leftarrow R(...x_i...), \neg start$

(2) $NEXT(z,x_i), ordered(x_i), old(z) \leftarrow R(...x_i...), \neg ordered(x_i), ordered(z), \neg old(z)$

(3-a) $MAX(z), old(z) \leftarrow ordered(z), \neg old(z)$

The application of rule (3-a) makes MAX non-empty and ends the first phase. Clearly, this constructs an ordering; however, if rule (3-a) is applied prematurely, only a subset of the active domain is ordered. We will see below how this can be detected. Now, suppose that P is

simulated by some program Q on ordered instances. Once the ordering was computed in the first phase, we use the program Q // MAX(z) where z is a variable not occurring in Q, to simulate P.

We finally add rules to detect premature termination of the first phase. There is a subtlety due to the fact that Q may delete tuples. The program Q // MAX(z) is modified in such a way that, for each relation R, the tuples that are deleted from R are recorded in a relation R_d (like in the proof of Theorem 4.3). If (3-a) was started too early, the following rules invalidate the computation:

(3-b') loop $\leftarrow$ MAX(z), R(...x_i...), $\neg$ordered(x_i),

loop $\leftarrow$ MAX(z), R_d(...x_i...), $\neg$ordered(x_i),

$\neg$ loop $\leftarrow$ loop

(where *loop* is a 0-ary predicate). □

Remark 4.4: The simulation of STL by SDL* in the proof of Theorem 4.3 uses infinite loops. However, as in the case of DL*, the use of infinite loops can be avoided by the roll-back technique described in Remark 4.2. Note that the technique described there does not involve invented values, so it applies to SDL* as well.

In view of the above, we have:

Fact: Each STL program t can be simulated by an SDL* program P without introducing additional non-terminating computations. In particular, if t always terminates, then P always terminates. □

4.2 Deterministic languages

In this section, we introduce the extensions detDL* and SdetDL* of our deterministic languages, and consider their expressive power.

The language detDL* is obtained from detDL by allowing the heads of rules to be negative literals. Again, negative literal are interpreted as deletions. If, on a given input, the parallel semantics requires a tuple to be both inserted and deleted at some stage, the computation blocks, and the effect is undefined for that input. More formally, the definition of the effect is like that for detDL except that (b) is replaced by (b*):

(b*) for each $i \leq p$, there is a valuation-set Γ_i for P and I_i such that

$I_{i+1} = I_i \cup (\Delta+_i) - (\Delta-_i)$, where

$\Delta+_i = \{\Gamma_i(r,v)(A) \mid$ for some $(r,v) \in dom(\Gamma_i)$, A is the positive head of r$\}$, and

$\Delta-_i = \{\Gamma_i(r,v)(A) \mid$ for some $(r,v) \in dom(\Gamma_i)$, $\neg$A is the head of r$\}$, and

$(\Delta+_i) \cap (\Delta-_i) = \emptyset$.

Remark: For each detDL* program P over an i-o schema <**R**,**S**>, if <**I**,**J**> $\in$ eff_{detDL*}(**R**,**S**,P), then J is the restriction of a model of Σ(P) to **S**.

The completeness proof for detDL* is straightforward.

Theorem 4.5: detDL* is deterministic complete.

Proof: (Sketch) Clearly, the effect of every detDL* program is a deterministic transformation. Conversely, let <**R**,**S**> be an i-o schema (**R** and **S** not necessarily disjoint) and τ a deterministic transformation over <**R**,**S**>. The proof that there is a detDL* program P such that τ = eff_{detDL*}(**R**,**S**,P), is similar to the proof of Theorem 4.1, with the difference that the control of the switch from phase one to phase two can be accomplished in a straightforward manner without infinite loops, using the increased control provided by the deterministic semantics. Details are omitted. $\square$

We now consider the extension SdetDL* of SdetDL (Datalog$^{\neg}$), which we also denote Datalog$^{\neg}$*. Again, in the safe case, deletions provide additional expressive power. Indeed, while SdetDL could express just the transformations in *FP*, it turns out that SdetDL* has the same power as the procedural language SdetTL (or, equivalently, the *while* language of Chandra). Before turning to the proof, we illustrate the language SdetDL* by the following program, which simulates a version of the *game of life*.

Example 4.6: Let G be a binary relation representing a graph G, and CELL a relation representing the vertices of the graph hosting a live cell. The rules are that a cell is created at a vertex if the vertex has two neighbours (living cells in an adjacent vertex); and a cell dies if it has 3 or more neighbours. We next present a Datalog$^{\neg}$* program corresponding to this. Two unary relations are used (compute and update). Intuitively, the computation alternates compute and update phases until a fixpoint for CELL is reached (if there is one). The control is given by the rules:

compute $\leftarrow$ $\neg$ compute, $\neg$ update

$\neg$ compute, update $\leftarrow$ compute.

The compute rules are:

3neighbours(x) $\leftarrow$

G(x,y), G(x,z), G(x,w), y$\neq$z, z$\neq$w, w$\neq$y, CELL(y), CELL(z), CELL(w), compute,

2neighbours(x) $\leftarrow$ G(x,y), G(x,z), y$\neq$z, CELL(y), CELL(z), compute.

The update rules are:

¬CELL(x), compute, ¬ update ← CELL(x), 3neighbours(x), update
CELL(x), compute, ¬ update ← ¬ CELL(x), 2neighbours(x), ¬3neighbours(x), update
¬3neighbours(x) ← 3neighbours(x), update
¬2neighbours(x) ← 2neighbours(x), update.

It is easy to see that some inputs lead to non-terminating computations. □

Indeed, we have:

Theorem 4.7: A transformation is computable by an SdetDL* program iff it is computable by an SdetTL program.

Proof (sketch): Clearly, each SdetDL* program can be simulated by an SdetTL program. Conversely, let t be an SdetTL program. The simulation can be reduced to the case when t is over disjoint i-o schemas, using a two-phase simulation similar to that in the proof of Theorem 4.1. Thus, assume t is over disjoint i-o schemas. In particular, t is a TL program so, by Theorem 3.1, it can be simulated by a detDL program. However, the detDL program uses invented values in the simulation to distinguish current from outdated content of some relations. To complete the proof, it is easy to verify that deletions can replace the use of invented values in the simulation. □

The above result shows that three of the languages we considered so far - SdetTL, SdetDL*, and the *while* language of [Ch] - have equivalent expressive power. In Section 5, we provide one more characterization of these languages based on a fixpoint extension of first-order logic.

5. THE CONNECTION WITH FIXPOINT EXTENSIONS OF FIRST-ORDER LOGIC

In this section, we investigate the connection between the extensions of Datalog presented so far and fixpoint extensions of first-order logic. In a first subsection, we consider Datalog¬, and in a second Datalog¬*. Lastly, we define fixpoint extensions of first-order logic corresponding to the non-deterministic languages.

5.1 Inflationary fixpoint logic and Datalog¬

In this section, we focus on our strongly safe, deterministic Declarative Language (Datalog¬, i.e., SdetDL). This language is of special interest, since it is a tractable language that provides a new, simple semantics for Datalog with negation. Furthermore, the language is strictly more expressive than Datalog with stratified negation [ABW, CH2, N, VG]. Indeed, we show that this language

expresses exactly the fixpoint queries. Since the predominant notation for Datalog with negation is Datalog$^\neg$ (e.g., see [KP, Ka]), we will also use this notation in the following.

As stated in Section 3, the syntax of Datalog$^\neg$ is that of Datalog extended with negation in bodies of rules, i.e., the syntax of detDL with the restriction that all variables occurring in the head of a rule also occur in the body of the rule. The semantics of Datalog$^\neg$ programs is a special case of the "parallel saturation" semantics of detDL programs, described in the previous section. However, it turns out that the same semantics can be expressed in a much simpler way for Datalog$^\neg$ due to the absence of invented values. Indeed, the semantics can be expressed straightforwardly using the notion of inductive fixpoint [GS]. We recall here the definitions of inflationary operator and inductive fixpoint for convenience.

Definition: Let **R** be a database schema. An *inflationary operator* over inst(**R**) is a total mapping ψ on inst(**R**) such that $I \subseteq \psi(I)$ for each I in inst(**R**) (the inclusion is pointwise). The *inductive fixpoint operator* IFP(ψ)(I) defined by ψ is the limit of the sequence $\psi^i(I)$, where $\psi^0(I) = I$, and $\psi^{i+1}(I) = \psi(\psi^i(I))$. (Note that, if the domain is finite, the sequence always converges.)

Each Datalog$^\neg$ program P defines an inductive fixpoint operator in a natural manner, as follows.

Definition: The *inflationary operator defined by* a Datalog$^\neg$ program P is the operator ψ_P on inst(sch(P)) such that, for every instance I over sch(P), $\psi_P(I)$ is the instance over sch(P) defined as follows. A ground literal A is true in $\psi_P(I)$ if A is true in I or for a ground instance

$$A \leftarrow B_1, \ldots, B_n$$

of some rule in P, each B_i is true in I. The *inductive fixpoint operator defined by P* is IFP(ψ_P)(I). (Note that, due to the absence of invented values, IFP(ψ_P)(I) is finite for every instance I.)

The semantics of Datalog$^\neg$ can now be easily expressed using the inductive fixpoint operator defined by programs, as stated next.

Fact: For each Datalog$^\neg$ program P, $eff_{Datalog^\neg}(P) = \{\langle I, IFP(\psi_P)(I)\rangle \mid I \text{ over } sch(P)\}$. $\square$

We next examine the expressive power of Datalog$^\neg$. We will show that Datalog$^\neg$ expresses exactly the well-known class of fixpoint queries (*FP*). In particular, this in conjunction with recent

results by Kolaitis [Ko] and Dahlhaus [D] showing that Datalog with stratified negation is strictly included in *FP*, implies that Datalog$^{\neg}$ is *strictly* more expressive than Datalog with stratified negation. Before presenting the proof, we briefly review some definitions and results related to the fixpoint queries.

Fixpoint queries are the queries definable in fixpoint logic (FO+FP), which is first-order logic augmented with a least fixpoint construct on positive formulas [GS, CH1]. Our proof will use a result of [GS] showing that the fixpoint queries can also be defined in *inflationary* fixpoint logic (FO+IFP), which is first-order logic extended with an inductive fixpoint operator.

Inflationary fixpoint (FO+IFP) formulas are defined next.

Definition: Inflationary fixpoint formulas are obtained by repeated applications of first-order operators and the inductive fixpoint operator starting from atoms. We omit the definitions of atoms and first-order operators ($\neg, \wedge, \vee, \exists, \forall$), which are standard. The inductive fixpoint operator is defined as follows. Let $\phi(S)$ be an FO+IFP formula with n free variables, where S is an n-ary predicate occurring in ϕ. Then IFP($\phi(S)$, S) denotes the n-ary predicate which is the limit of the sequence defined by: $J_0 = S$ and for each $i > 0$, $J_i = \phi(J_{i-1}) \cup J_{i-1}$. If $\vec{t}$ is a sequence of n variables or constants, IFP($\phi(S)$, S)$(\vec{t})$ is a formula.

Example 5.1: Let G be a binary relation schema. Consider the query "find all good nodes in the graph represented by an instance I of G, i.e. the nodes such that all their incoming edges originate from other good nodes". Note that a node is good iff it does not belong to a cycle in the graph represented by G. The query can be expressed in FO+IFP by the formula IFP(ϕ, good)(x), where

$$\phi = \{ x \mid \forall y \, (G(y,x) \rightarrow good(y)) \}$$

(good is initially empty). $\square$

Intuitively, IFP($\phi(S)$, S) can be viewed as an inductive definition of the predicate S, starting from some initial value (usually $\varnothing$). Note that this mechanism can be extended straightforwardly to define inductively *several* predicates simultaneously. Indeed, k relations $R_1, \ldots, R_k$ can be defined inductively by a system of k recursive equations of the form

$$R_i = \phi_i(R_1,\ldots,R_k) \;, 1 \leq i \leq k,$$

where the ϕ_i are formulas such that the arity of R_i equals the number of free variables of ϕ_i. The

Simultaneous Induction Lemma for FO+IFP, proven in [GS], shows that no power is gained by simultaneously defining by induction several relations rather than just one at a time. More precisely, it is shown that each predicate R_i defined by a system of equations as above can also be defined as $IFP(\psi_i, T)(\vec{t})$, for some ψ_i. We refer the reader to [GS] for the precise definitions and the proof.

We will prove that Datalog$^\neg$ computes the queries definable by FO+IFP formulas. As a side effect, we also obtain some interesting results concerning inflationary fixpoint logic itself. First, an alternate proof of the collapse of the FO+IFP hierarchy[3] is obtained (the original proof appears in [GS]). The collapse of the FO+IFP hierarchy means that each query in FO+IFP can be defined using a first-order formula and a *single* application of the IFP operator. The second result on FO+IFP provides an existential normal form for FO+IFP formulas.

The more complicated aspect of the proof of equivalence of Datalog$^\neg$ and FO+IFP is the simulation of FO+IFP formulas by Datalog$^\neg$. Intuitively, the simulation involves two main difficulties. The first involves delaying the firing of a rule until the completion of a fixpoint by another set of rules; intuitively, this is hard because checking that the fixpoint has been reached involves checking the *non-existence* rather than the existence of some valuation. The second concerns keeping track of iterations in the computation of a fixpoint. Before stating formally the result and providing the complete simulation, we illustrate these two main difficulties using two examples. (The second was suggested by Kanellakis.)

Example 5.2: The following Datalog$^\neg$ program computes the complement of the transitive closure of a binary relation R. The example illustrates the technique used to delay the firing of a rule (computing the complement) until the fixpoint of a set of rules (computing the transitive closure) has been reached. The idea is that, as the transitive closure of R is computed in relation S, the result is duplicated in relations *previous* and *previous-unless-last*, but with a delay of one iteration. However, the result of the previous iteration is copied in *previous-unless-last*, only if the current iteration is not the last. Thus, *previous* and *previous-unless-last* differ only at the last iteration, which allows recognizing when the fixpoint has been reached, and firing the rule computing the complement. The program consists of the rules:

$S(x,y) \leftarrow R(x,y)$

$S(x,y) \leftarrow R(x,z), S(z,y)$

$previous(x,y) \leftarrow S(x,y)$

previous-unless-last(x,y) ← S(x,y), R(x′,z′),S(z′,y′),¬S(x′,y′)
$\overline{S}$(x,y) ← ¬S(x,y), previous(x′,y′),¬previous–unless–last(x′,y′).

(It is assumed that R is not empty.) □

Example 5.1 (continued): Let G be a binary relation schema. Consider the query of Example 5.1, expressed in FO+IFP by IFP(ϕ, good)(x), where

$$\phi = \{ x \mid \forall\ y\ (G(y,x) \rightarrow good(y)) \},$$

and *good* is initially empty. The following Datalog¬ program P computes IFP(ϕ, good)(x) in relation *good*. Intuitively, the program P simulates consecutive iterations of ϕ. In order to distinguish between different iterations, we use timestamps to mark each iteration after the first. The timestamps used to mark iteration i are the values newly introduced in relation *good* at iteration i-1. The process continues until no new values are introduced in an iteration. Relations *delay* and *delay-stamped* are used to delay the derivation of new tuples in *good* until *bad* and *bad-stamped* (respectively) have been computed in the current iteration.

{perform first iteration}
bad(x) ← G(y,x), ¬ good(y)
delay ←
good(x) ← delay, ¬ bad(x)

{iteration with timestamp t}
bad-stamped(x,t) ← G(y,x), ¬good(y), good(t)
delay-stamped(t) ← good(t)
good(x) ← delay-stamped(t), ¬ bad-stamped(x,t). □

We next show formally the equivalence of FO+IFP and Datalog¬. Due to the interest of the result independently of rest of the development in the paper, we provide a direct proof. However, we note that a simpler proof is possible, based on our procedural languages: first, show that the language SdetTL without deletions can simulate FO+IFP (which is straightforward due to the procedural control available); next, show that SdetTL without deletions can be simulated by Datalog¬, using a variation of the proof of Theorem 3.1.

In order to state the result precisely, we must view formulas in FO+IFP as defining mappings

[3]The FO+IFP hierarchy is based on the depth of nesting of the IFP operator in a formula.

from database instances to database instances. To formalize this point of view, we use the following notation and definition.

Notation: With each variable x occurring in some FO+IFP formula is associated a unique attribute denoted A_x. We also assume that each attribute is A_x for some x. For each FO+IFP formula ϕ, the database schema consisting of one relation of appropriate arity for each predicate symbol in ϕ is denoted by in-sch(ϕ), and the database schema consisting of the single relation $\{A_x \mid x$ is a free variable in $\phi\}$ is denoted by out-sch(ϕ).

Definition: Each FO+IFP formula ϕ *defines* a transformation over (in-sch(ϕ),out-sch(ϕ)), also denoted ϕ for simplicity, as follows. For each instance I over in-sch(ϕ), ϕ(I) is equal to the predicate defined by ϕ for the interpretation of the predicates occurring in ϕ given by I.

Note that in the previous definition, the out-schema has a single relation. For simplicity, the equivalence result is also stated for single-relation output schemas. However, both the definition and result can be extended easily to multiple target schemas.

Theorem 5.3: Let <**R**,**S**> be an input-output schema where **S** contains a single relation schema not in **R**, and τ be a transformation over <**R**,**S**>. Then τ is definable by an FO+IFP formula iff it is definable in Datalog$^\neg$.

Proof: **(if)** Let P be a Datalog$^\neg$ program over the i-o schema <**R**,**S**>. It is easy to see that P is equivalent to a system of FO+IFP equations with simultaneous induction - one for each carrier. By the Simultaneous Induction Lemma for FO+IFP from [GS], the effect of P can be defined using a single FO+IFP formula ϕ. To ensure that out-sch(ϕ) = S, one has to use variable names corresponding to the attributes in S.

(only-if) The proof is by induction on the depth of the FO+IFP formula. The core of the proof involves a control mechanism which delays firing certain rules until other rules have been evaluated. Therefore, the induction hypothesis involves the capability to simulate the FO+IFP formula using a Datalog$^\neg$ program, as well as to concomitantly produce a predicate which only becomes true when the simulation has been completed. More precisely, we will prove by induction the following: for each FO+IFP formula ϕ, there exists a Datalog$^\neg$ program prog(ϕ) over the predicates in in-sch(ϕ), a predicate result_ϕ over $\{A_x \mid x$ occurs free in $\phi\}$, a 0-ary predicate done_ϕ, and possibly other predicates, such that:

(i) for every instance I over in-sch(ϕ) (extended with $\varnothing$ to the other relations of sch(prog(ϕ)), $IFP(\psi_{prog(\phi)})(I)[result_\phi] = \phi(I)$,

(ii) for every instance I over in-sch(ϕ) (extended with $\varnothing$ to sch(prog(ϕ))), the 0-ary predicate $done_\phi$ becomes true at the last stage in the evaluation of prog(ϕ) on I, and

(iii) prog(ϕ) does not modify any relation in in-sch(ϕ).

Note that (i) establishes the **only-if** part of the theorem. We will assume, without loss of generality, that no variable of ϕ occurs free and bound, or bound to more than one quantifier, and that ϕ contains no universal quantifier or $\vee$ operand. Suppose first that ϕ is an atom $R(\vec{t})$. Let $\vec{x}$ be the vector of distinct variables occurring in the $\vec{t}$. Then prog(ϕ) consists of the rules:

$result_\phi(\vec{x}) \leftarrow R(\vec{t})$

$done_\phi \leftarrow$

Suppose now that ϕ has depth more than one, and that (i), (ii) and (iii) are verified for each FO+IFP formula of depth less than that of ϕ. There are four cases to consider:

(1) $\phi = \alpha \wedge \beta$. Without loss of generality, we assume that [sch(prog(α)) - in-sch(α)] $\cap$ [sch(prog(β)) - in-sch(β)] = $\varnothing$. Thus, there is no interference between prog(α) and prog(β). Let $\vec{x}$ and $\vec{y}$ be the vectors of distinct free variables of α and β, respectively, and $\vec{t}$ the vector of distinct free variables occurring in $\vec{x}$ and $\vec{y}$. Then prog(ϕ) consists of the following rules:

prog(α)

prog(β)

$result_\phi(\vec{t}) \leftarrow done_\alpha, done_\beta, result_\alpha(\vec{x}), result_\beta(\vec{y})$

$done_\phi \leftarrow done_\alpha, done_\beta$

(2) $\phi = \exists\, x(\psi)$. Let $\vec{u}$ be the vector of distinct free variables of ψ, and $\vec{t}$ be the vector obtained from $\vec{u}$ by removing the variable x. Then prog(ϕ) consists of the rules:

prog(ψ)

$result_\phi(\vec{t}) \leftarrow done_\psi, result_\psi(\vec{u})$

$done_\phi \leftarrow done_\psi$

(3) $\phi = \neg(\psi)$. Let $\vec{t}$ be the vector of distinct free variables occurring in ψ. Then prog(ϕ) consists of:

prog(ψ)

$result_\phi(\vec{t}) \leftarrow done_\psi, \neg result_\psi(\vec{t})$

$done_\phi \leftarrow done_\psi$

(4) ϕ = IFP(ψ, S)($\vec{z}$), where S is a relation schema in sch(ψ), of the same arity as $result_\psi$. This case is the most involved, since it requires keeping track of the iterations in the computation of the fixpoint, as well as bookkeeping in order to control the value of the special predicate $done_\phi$. Intuitively, each iteration is marked by timestamps. The current timestamps consist of the tuples newly inserted in the previous iteration. The program prog(ϕ) uses the following new auxiliary relations:

- a relation *old* contains the timestamps introduced in the previous stages of the iteration.
- a relation *run* contains the timestamps. An active timestamp is in *run - old.*
- the relation *not_last* is used to detect the last iteration. A timestamp marks the last iteration if it is in *old - not_last.*
- the relation *fixpoint* contains IFP(ψ, S) at the end of the computation, and $result_\phi$ contains IFP(ψ, S)($\vec{z}$).
- relations *delay* and *not-empty* are used for timing and to detect an empty result.

In the following, $\vec{y}$ and $\vec{t}$ are vectors of distinct variables with the same arity as S. We first have particular rules to handle the first iteration and the case of an empty result (S is empty and the first iteration returns an empty result):

prog(ψ)
delay $\leftarrow done_\psi$
not_empty $\leftarrow result_\psi(\vec{y})$
$done_\phi \leftarrow$ delay, $\neg$ not_empty

The remainder of the program contains the rules:

<u>stamping of the database and starting an iteration:</u> for each R in sch(prog(ψ)) - {S}, and a vector $\vec{x}$ of distinct variables with same arity as R,

$R(\vec{x}, \vec{t}) \leftarrow$ fixpoint($\vec{t}$), $R(\vec{x})$
run($\vec{t}$) $\leftarrow$ fixpoint($\vec{t}$)
$S(\vec{x},\vec{t}) \leftarrow$ fixpoint($\vec{x}$), fixpoint($\vec{t}$)

<u>timestamped iteration:</u>

prog(ψ)[$\vec{t}$] // run($\vec{t}$), $\neg$old($\vec{t}$)

maintain *fixpoint, not-last* and *old*:

$\text{fixpoint}(\vec{y}) \leftarrow S(\vec{y})$

$\text{fixpoint}(\vec{y}) \leftarrow \text{result}_\psi(\vec{y})$

$\text{not_last}(\vec{t}), \text{fixpoint}(\vec{y}) \leftarrow \text{result}_\psi(\vec{y}, \vec{t}), \neg\, \text{fixpoint}(\vec{y})$

$\text{old}(\vec{t}) \leftarrow \text{done}_\psi(\vec{t})$

produce the result and detect termination

$\text{result}_\phi(\vec{y}) \leftarrow \text{fixpoint}(\vec{z})$ where $\vec{y}$ is the vector of distinct variables in $\vec{z}$

$\text{done}_\phi \leftarrow \text{old}(\vec{t}), \neg\, \text{not_last}(\vec{t})$

By inspection, it is seen that prog(ϕ) satisfies (i)-(iii) under the induction hypothesis. This concludes the induction. □

As seen from the above construction and the examples, the programs used to simulate FO+IFP queries use some temporary relations in addition to the input and output relations. However, the simulation can also be achieved using a single carrier which encodes the temporary and output relations, the result being obtained through a selection and a projection of the final value of the carrier. The selection is *simple*, i.e. the selecting condition is a conjunction of conditions of the form A = v, where A is an attribute and v is a constant or another attribute. Thus, the following can be easily verified (the proof is similar to that of the Simultaneous Induction Lemma of [GS] and is omitted).

Corollary 5.4: Let ϕ be a FO+IFP formula with k free variables. Then there is a Datalog$^\neg$ program P with a single carrier relation T, such that for each instance I of **R** (extended with $\varnothing$ to T) the formula $\text{IFP}(\phi, S)(\vec{t})$ defines $\pi_{A_1..A_k}(\sigma(\text{IFP}(\psi_P)(I)[T]))$, for some attributes $A_1,...,A_k$, and a simple selection σ. □

Remark (noted by Kolaitis): The use of the selection as in the above corollary is indispensable in general, when an FO+IFP formula is simulated by a Datalog$^\neg$ program with a single carrier. To see this, consider the complement of the transitive closure CTG of a binary relation G. Suppose that there exists a Datalog$^\neg$ program P with a single carrier T, such that CTG = $\pi_{A_1..A_k}(\text{IFP}(\psi_P)(G)[T])$. Then, CTG is empty iff T is empty. Next, note that T is empty iff it is empty after the first iteration of P. It follows easily that there is a first-order sentence stating that T is empty. Therefore, there is a first-order sentence stating that CTG is empty. But emptiness of CTG is equivalent to the strong connectivity of the directed graph represented by G. However,

strong connectivity is not definable in first-order logic (e.g., see [F,AU]), contradiction. □

It is easy to see that Datalog$^{\neg}$ programs with a single carrier can actually be simulated using the existential fragment of FO+IFP alone, since Datalog$^{\neg}$ programs do not involve universal quantification. This, in conjunction with Theorem 5.3 and Corollary 5.4, shows that FO+IFP has an existential normal form (see also Remark Section 6 of [G]). Note that the normal form implies the collapse of the FO+IFP hierarchy. More precisely, we have:

Corollary 5.5: For every FO+IFP formula ϕ, there exists an FO+IFP formula IFP(ν, T)($\vec{u}$) such that ν is an existential first-order formula and ϕ is equivalent to IFP(ν, T)($\vec{u}$). □

5.2 Partially-defined fixpoints and Datalog$^{\neg *}$

In the previous section, we showed the equivalence of Datalog$^{\neg}$ and inflationary fixpoint logic (FO+IFP). In this section, we show that Datalog$^{\neg *}$ is equivalent to another fixpoint extension of first-order logic, called *partial* fixpoint logic (FO+PFP). The main difference with FO+IFP is that the semantics of the fixpoint operator is not inflationary. Intuitively, this is the analog of the ability to perform deletions in Datalog$^{\neg *}$. Consequently, the fixpoint operator is partially defined, so interpretations of sentences in the logic are partially defined. This corresponds naturally to non-terminating computations in Datalog$^{\neg *}$. The results are similar to those obtained in the previous section. Therefore, the presentation here will be briefer and will focus on the differences with the previous results.

We first introduce the notion of "partial fixpoint operator", which is used to define partial fixpoint logic.

Definition: Let **R** be a database schema and ψ partially defined mapping over inst(**R**). The *partial fixpoint operator* PFP(ψ)(I) defined by ψ is the limit, <u>if it exists,</u> of the sequence ψ^i(I), where ψ^0(I) = I, and ψ^{i+1}(I) = $\psi(\psi^i(I))$ (ψ^{i+1}(I) is undefined if ψ is undefined on ψ^i(I), in which case PFP(ψ)(I) is also undefined).

Note that the semantics of Datalog$^{\neg *}$ program can be defined naturally using the partial fixpoint operator. First we define the partial operator corresponding to a Datalog$^{\neg *}$ program.

Definition: The *partial operator defined by* a Datalog$^{\neg *}$ program P is the operator ψ_P on

inst(sch(P)) defined as follows. Let I be an instance over sch(P) and $\chi(P,I)$ be the set of all (positive or negative) ground literals $(\neg)$ A such that for some $B_1,\ldots,B_n$,

$$(\neg)\ A \leftarrow B_1, \ldots, B_n$$

is the ground instance of some rule of P and each B_i is true in I. Then

- $\psi_P(I)$ is defined if $\chi(P,I)$ is consistent (i.e., there is no A such that A and $\neg A$ are both in $\chi(P,I)$);
- and a positive ground literal A is true in $\psi_P(I)$ if A is in $\chi(P,I)$, or A is in I and $\neg A$ is not in $\chi(P,I)$.

The *partial fixpoint operator defined by* P is $PFP(\psi_P)(I)$.

Similarly to the case of Datalog$^{\neg}$, we have:

Fact: For each Datalog$^{\neg *}$ program P,

$eff_{Datalog^{\neg *}}(P) = \{<I, PFP(\psi_P)(I)> \mid I \text{ over } sch(P), PFP(\psi_P)(I) \text{ is defined}\}$. $\square$

We next discuss *partial* fixpoint logic, which is first-order logic extended with a partial fixpoint operator.

Definition: Partial fixpoint formulas are obtained by repeated applications of first-order operators $(\neg, \wedge, \vee, \exists, \forall)$ and the partial fixpoint operator starting from atoms. The partial fixpoint operator is defined as follows. Let $\phi(S)$ be an FO+PFP formula with n free variables, where S is an n-ary predicate occurring in ϕ. Then $PFP(\phi(S), S)(\vec{t})$ is a formula, where $\vec{t}$ is a sequence of n variables or constants. The interpretation of $PFP(\phi(S), S)$ is the following. $PFP(\phi(S), S)$ denotes the n-ary predicate which is the limit, <u>if it exists</u>, of the sequence defined by: $J_0 = S$ and for each $i > 0$, $J_i = \phi(J_{i-1})$ (if ϕ is undefined on J_{i-1}, then J_i and the interpretation of $PFP(\phi(S), S)$ are undefined).

It is important to note that, unlike traditional fixpoint extensions of first-order logic (FO+FP and FO+IFP), sentences in FO+PFP do not generally have interpretations for all structures (instances). Hence, the transformations defined by FO+PFP formulas are partial mappings.

As we shall see, most properties of FO+IFP which we discussed carry to FO+PFP. As was the case for the IFP operator, the PFP operator can be extended straightforwardly to define inductively *several* predicates simultaneously. The *Simultaneous Induction Lemma* carries over to FO+PFP.

We next show the equivalence between Datalog$^{\neg *}$ and FO+PFP. Since we have already shown that Datalog$^{\neg *}$ is equivalent to the languages SdetTL and *while* of [Ch], this will provide a four-way characterization for the same class of transformations.

The proof of the equivalence is very similar to the proof for FO+IFP and Datalog$^{\neg}$.

Theorem 5.6: Let <**R**,**S**> be an output schema where **S** contains a single relation schema not in **R**, and τ a transformation over <**R**,**S**>. Then τ is defined by an FO+PFP formula iff it is the effect of a Datalog$^{\neg *}$ program.

Proof: **(if)** Similar to the if-part of the proof of Theorem 5.3. The difference concerns the simulation of deletions by FO+PFP. Specifically, a Datalog$^{\neg *}$ rule of the form

$$\neg R(\vec{x}) \leftarrow body(\vec{x})$$

gives rise to an equation of the form

$$R(\vec{x}) = R(\vec{x}) \wedge \neg body(\vec{x})$$

This is illustrated in Example 5.7 below.

(only-if) The proof is similar to the only-if part of the proof of Theorem 5.3. The induction is the same and so are the proofs for cases (1),(2) and (3). Now consider case (4). The simulation works as follows:

(a) evaluate prog(ψ),

(b) if result$_\psi$ is equal to S, the fixpoint is reached and result$_\phi$ is produced; otherwise

(c) assign result$_\psi$ to S, empty the temporary relations of prog(ψ) and go to (a).

Clearly, (a-c) can be realized in Datalog$^{\neg *}$. □

The simulation of Datalog$^{\neg *}$ by FO+PFP is illustrated next.

Example 5.7: Consider again the *game of life* Datalog$^{\neg *}$ program presented in Example 4.6. Consider the rules defining relation CELL:

$$\neg CELL(x), compute, \neg update \leftarrow CELL(x), 3neighbours(x), update$$

$$CELL(x), compute, \neg update \leftarrow \neg CELL(x), 2neighbours(x), \neg 3neighbours(x), update.$$

The equation corresponding to relation CELL in the system of PFP equations defining the carriers of the program is:

$$CELL(x) = [\ CELL(x) \vee [\neg CELL(x) \wedge 2neighbours(x) \wedge \neg 3neighbours(x) \wedge update]\ ] \wedge \neg[CELL(x) \wedge 3neighbours(x) \wedge update].$$

□

The following summarizes the characterizations available for the class of transformations defined by Datalog$^{\neg *}$.

Corollary 5.8: Let <**R**,**S**> be an output schema where **S** contains a single relation schema not in **R**, and τ a transformation over <**R**,**S**>. The following are equivalent:
(i) τ is the effect of a Datalog$^{\neg *}$ program,
(ii) τ is the effect of an SdetTL program,
(iii) τ is the effect of a *while* program (allowing constants), and
(iv) τ is definable by an FO+PFP formula.

Proof: Let τ be a transformation over <**R**,**S**>, as in the statement of the corollary. By Theorem 5.6, τ is expressible in FO+PFP iff τ is expressible in Datalog$^{\neg *}$. By Theorem 4.7, τ is expressible in Datalog$^{\neg *}$ iff τ is expressible in SdetTL. Finally, τ is expressible in SdetTL iff τ is expressible in the *while* language (extended with constants), by Theorem 1.1(e). □

The simulation of FO+PFP formulas by Datalog$^{\neg *}$ programs provides as a side effect some interesting results on FO+PFP itself: the collapse of the FO+PFP hierarchy, and an existential normal form for FO+PFP formulas. This is analogous to Corollary 5.5:

Corollary 5.9 For every FO+PFP formula ϕ, there exists an equivalent FO+PFP formula PFP(ν,T)($\vec{u}$) such that ν is an existential first-order formula.

Proof: The proof is the same as for Corollary 5.5. □

Note that the above normal form for FO+PFP implies the collapse of the FO+PFP hierarchy.

5.3 Variations of fixpoint logic for non-deterministic languages

In this section, we consider fixpoint extensions of first-order logic which correspond to non-deterministic languages. Such extensions must allow formulas that define *several* predicates for each given structure. This is achieved by a non-deterministic operator on formulas, called the *witness* operator. Informally, given a formula $\phi(x)$, the witness operator Wx applied to $\phi(x)$ chooses an arbitrary witness x which makes ϕ true. The extension based on the witness operator is orthogonal to the fixpoint extensions of first-order logic corresponding to the deterministic languages. Thus, we will consider inflationary and non-inflationary versions of fixpoint logic with

the W operator, corresponding to non-deterministic languages without or with deletions, respectively. The inflationary W-extension is denoted FO+W+IFP; the non-inflationary, W-extension is denoted FO+W+PFP.

We note that each "deterministic" logic has a natural W-extension. Thus, one can consider W-extensions of first-order logic, Horn clause logic (Datalog), etc. This yields a family of "non-deterministic" logics parallel to the traditional logics used for query languages. This raises several interesting questions of semantics, expressive power, and complexity, which will be explored in a separate paper. Here we focus on the W-extensions of fixpoint logic of interest in the context of the non-deterministic languages discussed earlier.

We now define the syntax of FO+W+IFP and FO+W+PFP formulas.

Definition: FO+W+IFP (FO+W+PFP) formulas are obtained by repeated applications of first-order operators, the inductive fixpoint operator IFP (the partial fixpoint operator PFP), and the witness operator starting from atoms. The syntax of atoms, first-order operators, and the IFP (PFP) operator are as before. The syntax of the W operator is defined next: if $\phi(\vec{x})$ is a formula, where $\vec{x}$ is a vector of distinct free variable in ϕ, then $W\vec{x}(\phi(\vec{x}))$ is a formula (all free variables of ϕ, including $\vec{x}$, remain free in $W\vec{x}(\phi(\vec{x}))$.

We next describe informally the semantics of FO+W+IFP (FO+W+PFP) formulas. The semantics of a formula ϕ is given by the set of predicates defined by ϕ for each given structure. We start with the W operator. In this context, a formula defines a *set* of predicates, i.e., the set of possible interpretations of the formula. Let $W\vec{x}(\phi(\vec{x},\vec{y}))$ be a formula, where $\vec{y}$ is the vector of variables other than $\vec{x}$ which are free in ϕ. The set of predicates defined by $W\vec{x}(\phi(\vec{x},\vec{y}))$ is the set of I such that for some J defined by ϕ,

- $I \subseteq J$,
- for each $\vec{y}$ for which $<\vec{x},\vec{y}>$ is in J for some $\vec{x}$, there exists a *unique* $\vec{x}_y$ such that $<\vec{x}_y,\vec{y}>$ is in I.

Intuitively, one "witness" $\vec{x}_y$ is chosen for each $\vec{y}$ satisfying $\exists\, \vec{x}\phi(\vec{x},\vec{y})$. It is also possible to describe the semantics of the W operator using functional dependencies: for each instance J defined by $\phi(\vec{x},\vec{y})$, $W\vec{x}(\phi(\vec{x},\vec{y}))$ defines all maximal sub-instances I of J such that the attributes corresponding to the variables in $\vec{y}$ form a key in I.

Note that $Wx(Wy\phi(x,y))$ is *not* equivalent[4] to $Wxy\phi(x,y)$; also, $Wx(Wy\phi(x,y))$ is not

[4]Two formulas are *equivalent* iff they define the same set of predicates for each given structure.

equivalent to Wy(Wxϕ(x,y)). To see the latter, let ϕ = R(x,y), where R is interpreted as {<0,1>,<2,1>,<2,3>}. Note first that {<0,1>,<2,1>} and {<0,1>,<2,3>} are the only possible interpretations of WyR(x,y), and {<0,1>}, {<2,1>} and {<0,1>,<2,3>} the only possible interpretations of Wx(WyR(x,y)). It is easily seen that {<2,3>} belongs to the set of predicates defined by Wy(WxR(x,y)), so Wx(WyR(x,y)) and Wy(WxR(x,y)) are not equivalent.

The semantics of the IFP and PFP operators are similar to the ones for the deterministic case, with the complication that each stage of the iteration has several possible outcomes. We outline the semantics for IFP (the one for PFP is analogous). Let ϕ(S) be an FO+W+IFP formula with n free variables, where S is a predicate of arity n occurring in ϕ. Then IFP(ϕ(S),S) defines all n-ary predicates J for which there exists a sequence $J_0,\ldots, J_k$ where J_0 = S, J_k = J, for each i, $0 < i \leq k$, J_i is the union of J_{i-1} with one predicate defined by $\phi(J_{i-1})$, and *each* predicate defined by $\phi(J_k)$ is included in J_k.

The definitions of the in-schema, out-schema, and transformation defined by an FO+W+IFP (FO+W+PFP) formula are analogous to those for FO+IFP (FO+PFP) and are omitted. Of course, the transformations defined by FO+W+IFP (FO+W+PFP) formulas are non-deterministic. Note that two formulas are equivalent iff they define the same transformation.

The following illustrates the use of the W operator.

Example 5.10: (i) Consider two relations

bonus(passenger-name) and

records(passenger-name, flight#, day, month, year)

of an airline database. Relation *bonus* holds the names of all passengers who have been given a bonus for which it is necessary to have flown in March 1988. The following (FO+W) formula defines a relation *verification* which exhibits a qualifying flight (flight# and day) for each passenger given the bonus:

verification(n,f,d) = bonus(n) $\wedge$ Wfd(records(n,f,d,"March","1988")).

(ii) Let G be a symmetric, binary relation. The FO+W+PFP formula PFP(ϕ(G),G)(x,y), defines a "triangular" representation G' of G, where one edge <x,y> is retained for each <x,y> and <y,x> in G:

$$\phi(x,y) = [G(x,y) \wedge \neg Wxy(G(x,y) \wedge G(y,x))].$$

This has the effect of removing from G one "redundant" edge at each stage. Also note that such G' cannot generally be defined without the *witness* operator (or by any deterministic and generic means). $\square$

As earlier, the definitions of the IFP and PFP operators can be extended to allow the definition of several predicates by simultaneous induction. Again, the Simultaneous Induction Lemma carries over.

It turns out that FO+W+IFP (FO+W+PFP) correspond naturally to some of the safe non-deterministic extensions of Datalog considered in the previous sections. Specifically, we show that FO+W+IFP is equivalent to SDL$\forall$, and to SDL on ordered databases. Thus, FO+W+IFP defines the DB-NPTIME transformations. Also, FO+W+PFP is equivalent to SDL*. Thus, FO+W+PFP defines the DB-NPSPACE transformations.

Theorem 5.11: For each i-o schema **<R,S>**, where S consists of a single relation, the following hold:
(i) FO+W+IFP expresses the same transformations over **<R,S>** as SDL on ordered databases,
(ii) FO+W+IFP expresses the same transformations over **<R,S>** as SDL$\forall$ (i.e., DB-NPTIME),
(iii) FO+W+PFP expresses the same transformations over **<R,S>** as SDL* (i.e., DB-NPSPACE).

Proof: Clearly, every transformation defined by a FO+W+IFP (FO+W+PFP) formula is in DB-NPTIME (DB-NPSPACE). Since SDL on ordered databases and SDL$\forall$ on arbitrary databases express DB-NPTIME (Proposition 2.7 and Theorem 2.8), and SDL* expresses the DB-NPSPACE transformations (Theorem 4.3), it is clear that the formulas can be simulated by the corresponding Datalog extensions. The converse is straightforward, and similar to the simulation of Datalog$^\neg$ and Datalog$^\neg$* by FO+IFP and FO+PFP, respectively. $\square$

The above result is illustrated by the following.

Example 5.12: (a) Consider the FO+W+PFP formula of Example 5.10 (ii). A corresponding SDL* program is:

$G'(x,y) \leftarrow G(x,y), \neg erased(x,y)$
$\neg G'(x,y), erased(x,y) \leftarrow G'(x,y), G'(y,x).$

Note that a simpler program can be obtained if G' is computed in-place, by modifying G:

$$\neg G(x,y) \leftarrow G(x,y), G(y,x).$$

(b) Consider the SDL* program above. Although Example 5.10 (ii) provides a simple FO+W+PFP formula equivalent to it, we will construct a second equivalent FO+W+PFP formula to illustrate the simulation of SDL* by FO+W+PFP in the general case. First, we construct an SDL* program with a single carrier T encoding G' and *erased*: T(x,y,1) means that <x,y> is in G', and T(x,y,0) means that <x,y> is in *erased*. This yields the following program:

$T(x,y,1) \leftarrow G(x,y), \neg T(x,y,0)$

$\neg T(x,y,1), T(x,y,0) \leftarrow T(x,y,1), T(y,x,1).$

The result is obtained by decoding G' from T by a simple selection and a projection: $\pi_{1..2}(\sigma_{T.3=1}(T))$. The SDL* program over one carrier is now transformed into an equivalent FO+W+PFP formula:

$PFP(\psi(T), T)(x,y,1)$, with

$\psi(x,y,\alpha) = \exists z\{ (Wz((z=1) \vee (z=2))) \wedge [((z=1) \wedge \psi_1) \vee ((z=2) \wedge \psi_2) \vee ((z=2) \wedge \psi_3)] \},$

where:

$\psi_1 = T(x,y,\alpha) \vee [(\alpha=1) \wedge Wxy(G(x,y) \wedge \neg T(x,y,0))],$

$\psi_2 = \neg \exists x',y'[T(x',y',1) \wedge T(y',x',1))] \wedge T(x,y,\alpha),$

$\psi_3 = \exists x',y'[T(x',y',1) \wedge T(y',x',1))] \wedge$

$\{ [(\alpha=0) \wedge T(x,y,0)] \vee [(\alpha=0) \wedge (x=x') \wedge (y=y')] \vee [(\alpha=1) \wedge T(x,y,1) \wedge \neg((x=x')\wedge(y=y'))] \}.$

In ψ, the choice of a value of z (1 or 2) simulates the non-deterministic choice of firing the first or second rule. The formula ψ_1 corresponds to the first rule of the program. (It is "active" if z=1 was chosen.) The formulas ψ_2 and ψ_3 correspond to the second. (They are "active" if z=2 was chosen.) The presence of an insertion and a deletion in the second rule forces us to distinguish two cases:

- the second rule is not applicable (ψ_2) and the database is just copied; and
- the second rule is applicable (ψ_3), tuples [x,y,0] are kept, the tuple [x',y',0] is derived, and tuples [x,y,1] are kept if $(x,y) \neq (x',y')$.

(c) We finally illustrate the straightforward simulation of SDL programs by FO+W+IFP formulas. Consider the following SDL program computing the transitive closure of G in T:

$T(x,y) \leftarrow G(x,y)$

$T(x,y) \leftarrow T(x,z), T(z,y).$

The equivalent FO+W+IFP formula is $IFP(\nu(T),T)(x,y)$, where

$\nu(x,y) = Wxy\exists z[G(x,y) \vee (T(x,z) \wedge T(z,y))].$

Note the form of ν, where all free variables are preceded by the W operator, and the others are existentially quantified. $\square$

The simulations of FO+W+IFP and FO+W+PFP by the Datalog extensions, and the converse

simulations, provide some interesting results on FO+W+IFP and FO+W+PFP themselves (as in the case of FO+IFP and FO+PFP). The results concern normal forms (implying the collapse of the respective hierarchies). In particular, it is shown that FO+W+IFP has a "W" normal form and a "W-$\exists$" normal form on ordered databases.

Corollary 5.13:

(i) For each FO+W+IFP formula ϕ, there exists a first-order formula ψ whose free variables are $\vec{x}$, such that ϕ is equivalent to

$$\mathrm{IFP}(W\vec{x}\psi(\vec{x}), T)(\vec{t})$$

for some $\vec{t}$ and predicate T of ψ.

(ii) For each FO+W+IFP formula ϕ, there exists an *existential* first-order formula ψ with free variables $\vec{x}$, such that ϕ is equivalent *on ordered databases* to

$$\mathrm{IFP}(W\vec{x}\psi(\vec{x}), T)(\vec{t})$$

for some $\vec{t}$ and predicate T of ψ.

(iii) For each FO+W+PFP formula ϕ, there exists a FO+W formula ψ such that ϕ is equivalent to

$$\mathrm{PFP}(\psi, T)(\vec{t})$$

for some $\vec{t}$ and predicate T of ψ.

Proof (sketch): To see (i), note that each FO+W+IFP program can be simulated by a SDL$\forall$ program (Theorem 5.11); and, each SDL$\forall$ program can be simulated by a FO+W+IFP formula of the stated form. Note that ψ may contain universal quantification inherited from the SDL$\forall$ program. Next, (ii) follows from the fact that SDL alone is sufficient to simulate FO+W+IFP *on ordered databases* (Theorem 5.11). Since SDL does not contain $\forall$, it is easily seen that ψ of the corresponding FO+W+IFP formula is existential. Consider (iii). By Theorem 5.11, each FO+W+PFP formula has an equivalent SDL* program. Conversely, each SDL* program has an equivalent FO+W+PFP formula $\mathrm{PFP}(\xi,T)(\vec{t})$, where ξ is in FO+W. $\square$

Remark: It turns out that FO+W+PFP has an *existential* normal form. The proof is non-trivial. To see the origin of the problem, consider the formulas ψ_i above. Universal quantification is used in ψ_3 to check the non-existence of an applicable valuation of the corresponding SDL* rule. This is due to the difference in the semantics of SDL* and the PFP operator: if an SDL* rule is not fired, the database is left unchanged; on the other hand, if an iteration in the computation of the PFP operator yields the empty set, the database becomes empty. However, the use of the universal quantifier can be avoided by "decomposing" the evaluation of ψ_3 into two stages: one to check the

existence of a valuation (and mark this in the carrier), the second to actually evaluate ψ_3 if such a valuation exists. □

6. CONCLUSION

The Datalog extensions discussed in this paper are summarized in Figure 1. They are classified according to three orthogonal characteristics: determinism, safety, and inflationary character. The expressive power of the languages is summarized in Figure 2. Note that we obtained deterministic and non-deterministic complete languages, as well as languages capturing important classes of database transformations. Thus, in the non-deterministic case, we obtained DB-NPTIME (captured by SDL$\forall$, SDL$\perp$, SDL on ordered instances, the procedural STL$^+$ and the fixpoint logic FO+W+IFP) and DB-NPSPACE (captured by SDL*, the procedural STL, and the fixpoint logic FO+W+PFP). In the deterministic case, we obtained the fixpoint queries (SdetDL or Datalog$^\neg$, the procedural SdetTL$^+$, and the fixpoint logic FO+IFP) and languages equivalent to the *while* language of [Ch] (SdetDL* or Datalog$^\neg$*, the procedural SdetTL and the fixpoint logic FO+PFP). (Note that the *while* language yields DB-PSPACE on ordered databases.) Thus, the class of transformations computed by the *while* language has a four-way characterization and emerges as an important class of deterministic transformations together with the fixpoint queries. The problem of capturing precisely the DB-PTIME transformations remains open.

Note that some results on expressive power are subject to the disjointness of the input and output schemas. This is not a significant restriction. Indeed, the intuitive meaning of a relation R present in both the input and output schema of a program is that R is being updated. Then one can force disjointness of the input and output schemas by referring to the original relation R as *old*R, and to the updated R as *new*R.

The connection between the Datalog extensions and the corresponding procedural languages is summarized in Figure 3. In particular, this provides some intuition on the impact of various features of a language on its ability to simulate explicit control. The features relevant to the simulation of control are:

- invented values.

As long as invented values are available, the Datalog extensions are equivalent to their procedural counterparts. Intuitively, invented values can be used to timestamp tuples controlling the firing of rules. This allows to simulate iterative control.

- determinism.

The deterministic semantics provides additional control capability over the non-deterministic one, since all rules of a program are forced to fire simultaneously for all applicable instantiations. While with non-deterministic semantics explicit control is simulated at the cost of introducing additional non-terminating computations, this disadvantage can be avoided with the deterministic semantics. Note that all deterministic Datalog extensions considered are equivalent to their corresponding procedural languages. (See Figure 3.)

- negations in heads (deletions).

Intuitively, the use of deletions to simulate control is similar to the use of invented values: deletions allow the repeated use of tuples controlling the firing of rules. Thus, all languages with deletion can simulate their procedural counterparts. Also, non-deterministic languages with deletions do not require additional non-terminating computations for the simulation. (As noted above, this is not an issue for the deterministic languages.)

Figure 3 also exhibits the connection between the safe Datalog extensions and various fixpoint extensions of first-order logic. (We did not provide fixpoint extensions corresponding to the unsafe languages, although this could be done by providing in the logics a mechanism for introducing new constants in the universe, in a manner similar to [HS].) The fixpoint logic FO+IFP is well known (see [GS]), while the other three variations we consider are new. The new fixpoint extensions, particularly the "non-deterministic" ones, are interesting in their own right from a logic point of view. However, we focus here on the connection with the Datalog extensions, and leave the more detailed investigation of the logics for a separate paper. Intuitively, the fixpoint logics are similar to the procedural languages with respect to control capability: composition is equivalent to nesting in the logics, and iteration is equivalent to an application of a fixpoint operator. Not surprisingly, the safe languages which can simulate corresponding fixpoint logics are precisely those which can simulate their procedural counterparts.

The simulation of the fixpoint logics by the Datalog extensions, and conversely, provided as a side-effect several normal forms for the fixpoint logics. In all cases, the normal forms imply the collapse of the respective hierarchies (based on the depth of nesting of the fixpoint operator). Some of these results are new; the collapse of the FO+IFP hierarchy was known ([GS]), but our simulations provide a simple alternate proof. Analogous normal forms are obtained for the procedural languages. (The analog of the collapse of the fixpoint hierarchies is a normal form without nested *while* loops.)

The ability of some of the Datalog extensions to simulate explicit control has practical

significance: if users are given the option of using a hybrid language mixing explicit control and "declarative" pieces, the resulting programs can still be interpreted within the "declarative" language. This may be preferable to encoding complicated control using the limited control capability of the Datalog extensions. For example, consider Datalog$^{\neg}$. The use of explicit composition in conjunction with Datalog$^{\neg}$ provides a straightforward way to specify stratified semantics for a program (as the composition of the Datalog$^{\neg}$ programs for each stratum); without explicit control, the simulation of the stratified semantics with the inflationary semantics of Datalog$^{\neg}$ is much more complicated (e.g., see Example 5.2 on the computation of the complement of transitive closure). We note that mixing procedural and "declarative" constructs to obtain programs with clean semantics is also suggested in [IN], where a "rule algebra" similar to our procedural language SdetTL^{+} is proposed.

Finally, we review some of the recurring techniques for simulating control in the Datalog extensions:

- the use of control predicates to trigger or inhibit rules;
- the use of timestamps;
- the use of non-deterministic switches and error handling for them;
- the use of copies of relations, offset by one stage, to detect the end of an iteration;
- the use of *done* predicates to indicate the end of the computation of a fixpoint for a subprogram;
- maintaining a journal and the use of roll-backs when deletions are available; and
- the use of a (provided or constructed) ordering of the active domain to replace non-deterministic transfer of control by an exhaustive search of the active domain.

The results of this paper concern primarily the expressive power. Other issues of interest remain to be investigated. We mention briefly a few:

- optimization of such programs, parallelization,
- conditions guaranteeing deterministic effects for non-deterministic programs
- verifying termination, and conditions for termination.

An investigation relevant to the second topic was conducted in [MS3] for a language similar to SDL*.

Acknowledgements

The authors wish to thank Ashok Chandra, Richard Hull, Paris Kanellakis, Phokion Kolaitis, Eric Simon, and Moshe Vardi for discussions related to the material contained in this paper. Catriel Beeri provided useful comments on a draft of the paper.

REFERENCES

[Ab] Abiteboul, S., Updates, a new frontier, Proc. of International Conf. on Database Theory, Bruges (1988).

[Ap] Apt, K., Introduction to Logic Programming, Technical Report TR-87-35, Univ. of Texas, Austin. To appear as a chapter in Handbook of Theoretical Computer Science, North-Holland.

[ABW] Apt, K., H. Blair, A. Walker, Toward a Theory of Declarative Knowledge, Proc. of Workshop on Foundations of Deductive Database and Logic Programming (J. Minker ed.) (1988).

[AU] Aho, A.V., J.D.Ullman, Universality of Data Retrieval Languages, Proc. 6th ACM Symp. on Principles of Prog. Languages, San Antonio, Texas (1979), 110-117.

[AV1] Abiteboul, S., V. Vianu, A transaction language complete for database update and specification. Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1987).

[AV2] Abiteboul, S., V. Vianu, Transaction languages for database update and specification. I.N.R.I.A. Technical Report No.715 (1987). Invited to special issue of J. of Computer and Systems Science, to appear.

[AV3] Abiteboul, S., V. Vianu, Procedural and declarative database update languages. Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1988), 240-250.

[BF] Bidoit, N., C. Froidevaux, Minimalism subsumes default logic and Circumscription in Stratified Logic Programming, Proc. of Symposium on Logic In Computer Science, IEEE, Ithaca NewYork, (1987), 89-97.

[C] Codd, E.F., a relational model of data for large shared data banks, CACM (1970).

[CH1] Chandra, A.K., D. Harel, Computable Queries for Relational Databases, Journal of Computer and System Sciences 21:2 (1980), 156-178.

[CH2] Chandra, A.K., D. Harel, Structure and Complexity of Relational Queries, Journal of Computer and System Sciences 25:1 (1982), 99-128.

[Ch] Chandra, A.K., Programming Primitives for Database Languages, Proc. ACM Symposium on Principles of Programming Languages, Williamsburg (1981), 50-62.

[D] Dahlhaus, E., Skolem Normal Forms Concerning the Least Fixpoint, Computation and Logic, in Lecture Notes in Computer Science, Springer Verlag 1987.

[DE] Delcambre,L., J.Etheredge, The Relational Production Language: a Production Language for Relational Databases, Int'l. Conf. on Expert Database Systems, Washington (1988).

[F] Fagin, R., Monadic Generalized Spectra, Z. Math. Logik, 21, (1975), pp. 89-96.

[G] Gurevich, Y., Logic and the Challenge of Computer Science, Trends in Theoretical Computer Science, (E.Borger), Computer Science Press (1988). pp.1-57.

[GS] Gurevich Y., S. Shelah, Fixed-Point Extensions of First-Order Logic, FOCS (1985)

[H] Hull, R. Relative information Capacity of Simple Relational Database Schemata, SIAM J. of Computing vol. 15:3 (1986), 856-886.

[HS] Hull, R., J.Su, On the Expressive Power of Database Queries with Intermediate Types, Proc. ACM SIGACT-SIGART-SIGMOD Symp. on Principles of Database Systems, (1988), 39-51

[IN] Imielinski, T., S. Naqvi, Explicit control of logic programs through rule algebra, Proc. ACM SIGACT-SIGART-SIGMOD Symp. on Principles of Database Systems, (1988), 103-116.

[Ka] Kanellakis, P.C., Elements of Relational Database Theory, to appear as a chapter in Handbook of Theoretical Computer Science, North-Holland.

[Ko] Kolaitis, P., The Expressive Power of Stratified Logic Programs, manuscript (1987).

[KP] Kolaitis, P., C. Papadimitriou, Why not negation by fixpoint? Proc. ACM SIGACT-SIGART-SIGMOD Symp. on Principles of Database Systems, (1988), 231-239.

[MS1] de Maindreville, C., E. Simon, A production rule based approach to deductive databases, Proc. Conf. on Data Engineering (1988).

[MS2] de Maindreville, C., E. Simon, Modelling a Production Rule Language for Deductive Databases, Proc. Conf. on Very Large Data Bases (1988).

[MS3] de Maindreville, C., E. Simon, Deciding whether a production rule is relational computable, Proc. Int'l. Conf. on Database Theory (1988).

[MW] Manchanda S., D.S. Warren, A logic-based language for database updates, Foundations of Logic Programming and Deductive Databases, ed. J. Minker (1987).

[N] Naqvi, S., A logic for negation in database systems, Proc. Workshop on Logic Databases (1986).

[NK] Naqvi, S., R. Krishnamurthy, Database Updates in Logic Programming, Proc. ACM SIGACT-SIGART-SIGMOD Symp. on Principles of Database Systems, (1988), 251-262.

[Pr] Przymunsinski, T.C., On the semantics of stratified deductive databases, Workshop on Foundations of Deductive Databases and Logic Programming, (1986), pp 378-387.

[SCG] Stevens, A., Collins, A., S.E.Goldin, *Intelligent Tutoring Systems*, Eds. D.Sleeman and J.S.Brown, Academic Press, 1982.

[U] Ullman, J.D., Principles of Database and Knowledge Base Systems, Computer Science Press (1988).

[VG] Van Gelder, A., Negation as failure using tight derivations for general logic programs, Proc. IEEE symposium on Logic Programming (1986).

	INFLATIONARY		NON-INFLATIONARY	
	non-deterministic semantics	deterministic semantics	non-deterministic semantics	deterministic semantics
unsafe	DL		DL*	
weakly safe		detDL		detDL*
safe	SDL	SdetDL Datalog¬	SDL*	SdetDL* Datalog¬*

Figure 1: DL Languages

LANGUAGE	POWER	RESTRICTIONS
DL	Non-deterministic Complete	Disjoint i-o
DL*	Non-deterministic Complete	
detDL	Deterministic complete	Disjoint i-o
detDL*	Deterministic Complete	
SDL	DB-NPTIME	Ordered instances Disjoint i-o
SDL$\not\vee$& SDL$\perp$	DB-NPTIME	Disjoint i-o
SDL*	DB-NPSPACE	
Datalog$^{\neg}$	Fixpoint queries	
Datalog$^{\neg}$*	While language	

Figure 2: Expressive power

LANGUAGE	PROCEDURAL	FIXPOINT LOGIC
DL	TL (disjoint i-o)	
DL*	TL	
detDL	detTL (disjoint i-o)	
detDL*	detTL	
SDL∀& SDL⊥	STL^+	FO+W+IFP
SDL*	STL	FO+W+PFP
$Datalog^\neg$	$SdetTL^+$	FO+IFP
$Datalog^\neg$*	SdetTL	FO+PFP

Figure 3: Connections with procedural languages and fixpoint logics

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique