Datalog Educational System V2.0 User's Manual

Technical Report SIP 139-04

Fernando Sáenz Pérez
Departamento de Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid
Original Report (V1.1) dated 4/3/2004
New Version dated 08/31/2010

[1] **http://www.fsf.org/**

Contents

1. Introduction **7**
1.1 Deductive Databases 7
1.2 Referring to DES 8
2. Installation **8**
2.1 Downloading DES 8
2.1.1 Source Distribution 9
2.1.2 Executable Distribution 9
2.1.2.1 Windows 9
2.1.2.2 DES+ACIDE Windows Bundle 10
2.1.2.3 Linux 11
2.2 Installing and Executing DES 11
2.2.1 MS Windows 12
2.2.1.1 Executable Distribution 12
2.2.1.2 Source Distribution 12
2.2.2 Linux 12
2.2.2.1 Executable Distribution 12
2.2.2.2 Source Distribution 12
2.2.3 Starting DES from a Prolog interpreter 13
3. Getting Started **13**
3.1 Datalog Mode 14
3.2 SQL Mode 16
3.3 Prolog Mode 19
3.4 Getting Help 20
4. Query Languages **20**
4.1 Datalog 21
4.1.1 Syntax 21
4.1.2 Rules 24
4.1.3 Programs 24
4.1.4 Queries 24
4.1.5 Duplicates 24
4.1.6 Temporary Views 26
4.1.7 Automatic Temporary Views 26
4.1.8 Negation 27
4.1.9 Null Values 28
4.1.10 Outer Joins 30
4.1.11 Aggregates 32
4.1.11.1 Aggregate Functions 32
4.1.11.2 Group_by Predicate 32
4.1.11.3 Aggregate Predicates 35
4.1.12 Disjunctive Bodies 36
4.2 SQL 37
4.2.1 Main Limitations 37
4.2.2 Main Features 38
4.2.3 Datalog vs. SQL 38
4.2.4 Data Definition Language 38
4.2.4.1 Creating Tables 38
4.2.4.2 Creating Views 41

4.2.4.3 Dropping Tables ... 42
4.2.4.4 Dropping Views ... 42
4.2.4.5 Dropping Databases ... 42
4.2.5 Data Manipulation Language ... 43
4.2.5.1 Inserting Tuples ... 43
4.2.5.2 Deleting Tuples ... 43
4.2.6 Data Query Language ... 44
4.2.6.1 Basic SQL Queries ... 44
4.2.6.2 Set SQL Queries ... 46
4.2.6.3 `WITH` SQL Queries ... 46
4.2.7 SQL Grammar ... 47
4.3 Prolog ... 53
4.4 Built-ins ... 53
4.4.1 Comparison Operators ... 53
4.4.2 Datalog and Prolog Arithmetic ... 54
4.4.3 SQL Arithmetic ... 55
4.4.4 Arithmetic Built-ins ... 55
4.4.4.1 Arithmetic Operators ... 56
4.4.4.2 Arithmetic Constants ... 56
4.4.4.3 Arithmetic Functions ... 56
4.4.5 Negation ... 58
4.4.6 Datalog Outer Joins ... 58
4.4.7 Datalog Aggregates ... 58
4.4.7.1 Aggregate Functions ... 58
4.4.7.2 Group_by Predicate ... 58
4.4.7.3 Aggregate Predicates ... 59
4.4.8 Datalog Null-related Predicates ... 59
5. System Description ... 59
5.1 RDBMS connections via ODBC ... 59
5.1.1 Important Caveats ... 63
5.1.2 Platform-specific Issues ... 65
5.2 Safety and Computability ... 65
5.3 Source-to-Source Transformations ... 67
5.4 Datalog and SQL Tracers ... 67
5.4.1 Tracing Datalog Queries ... 68
5.4.2 Tracing SQL Views ... 69
5.5 Datalog Declarative Debugger ... 70
5.7 Test Case Generator ... 71
5.8 Batch Processing ... 73
5.9 Messages ... 74
5.10 Commands ... 74
5.10.1 Rule Database Commands ... 75
5.10.2 Relational Database Commands ... 76
5.10.3 Debugging and Test Case Generation ... 76
5.10.4 Extension Table Commands ... 77
5.10.5 Operating System Commands ... 77
5.10.6 Log Commands ... 78
5.10.7 Informative Commands ... 78
5.10.8 Languages ... 79
5.10.9 Miscellanea ... 79

5.11 Notes about the Implementation of DES ... 79
5.11.1 Tabling ... 80
5.11.2 Finding Stable Models ... 81
5.11.3 Dependency Graphs and Stratification: Negation, Outer Joins, and Aggregates ... 81
5.11.4 Porting to Unsupported Systems ... 82
5.11.5 Differences among Platforms ... 82
6. Examples ... 82
6.1 Relational Operations (files `relop.{dl,sql}`) ... 83
6.2 Paths in a Graph (files `paths.{dl,sql}`) ... 86
6.3 Shortest Paths (file `spaths.{dl,sql}`) ... 87
6.4 Family Tree (files `family.{dl,sql}`) ... 88
6.5 Basic Recursion Problem (file `recursion.dl`) ... 90
6.6 Transitive Closure (files `tranclosure.{dl,sql}`) ... 90
6.7 Mutual Recursion (files `mutrecursion.{dl,sql}`) ... 91
6.8 Farmer-Wolf-Goat-Cabbage Puzzle (file `puzzle.dl`) ... 92
6.9 Paradoxes (file `russell.dl}`) ... 94
6.10 Parity (file `parity.dl`) ... 96
6.11 Grammar (file `grammar.dl`) ... 97
6.12 Fibonacci (file `fib.dl`) ... 98
6.13 Hanoi Towers (file `hanoi.dl`) ... 98
6.14 Other Examples ... 99
7. Contributions ... 99
8. Related Work ... 100
8.1 Deductive Database Systems ... 101
8.2 Technological Transfers ... 102
9. Future Enhancements ... 102
10. Caveats and Limitations ... 102
11. Release Notes History ... 103
11.1 Version 2.0 of DES (released on August, 31st, 2010) ... 104
11.2 Version 1.8.1 of DES (released on March, 17th, 2010) ... 105
11.3 Version 1.8.0 of DES (released on December, 18th, 2009) ... 105
11.4 Version 1.7.0 of DES (released on October, 30th, 2009) ... 107
11.5 Version 1.6.2 (released on March, 10th, 2009) ... 109
11.6 Version 1.6.1 (released on November, 10th, 2008) ... 110
11.7 Version 1.6.0 (released on July, 28th, 2008) ... 112
11.8 Version 1.5.0 (released on December, 30th, 2007) ... 113
11.9 Version 1.4.0 (released on September, 2nd, 2007) ... 114
11.10 Version 1.3.0 (released on May, 2nd, 2007) ... 116
11.11 Version 1.2.0 (released on February, 9th, 2007) ... 117
11.12 Version 1.1.2 (released on December, 20th, 2006) ... 118
11.13 Version 1.1.1 (released on February, 21st, 2005) ... 118
11.14 Version 1.1 (released on March, 4th, 2004) ... 118
11.15 Version 1.0 (released on December, 2003) ... 119
12. Acknowledgements ... 120
Appendix A. GNU General Public License ... 121
Bibliography ... 127

1. Introduction

The Datalog Educational System (DES) is a free, open-source, multiplatform, portable, Prolog-based implementation of a basic deductive database system. DES 2.0 is the current implementation, which enjoys Datalog and SQL query languages, full recursive evaluation with memoization techniques, arithmetic, stratified negation and novel approaches to declarative debugging, test case generation for SQL views, null values support, (tabled) outer join and aggregate predicates. The system is implemented on top of Prolog and it can be used from a Prolog interpreter running on any OS supported by the Prolog interpreter. Moreover, Windows and Linux executables are also provided.

The main novelty of the current release is the support of connections to relational database management systems (RDBMSs) to provide data sources for relations. This means that a relation defined in a RDBMS as a view or table is allowed as any other relation defined via a predicate in the deductive database. Then, computing a query can involve computations both in the deductive inference engine and in the external RDBMS SQL engine. In addition, duplicates are now supported and can be enabled upon request via a command. Also, naïve tracers for Datalog and SQL has been included, which are intended to catch buggy predicates and views, respectively. A complete list of addtions, changes and fixed bugs is included in the Release Notes (Section 11.1).

We have developed DES aiming to have a simple, interactive, multiplatform, and affordable system (not necessarily efficient) for students, so that they can get the fundamental concepts behind a deductive database with Datalog and SQL as query languages. SQL is supported with a reasonable coverage of the standard for teaching purposes. Other deductive systems are not fully suited to our needs due to the absence of some characteristics DES does offer for our educational purposes. This system is not targeted as a complete deductive database, so that it does not provide persistency, transactions, security, and other features present in current database systems.

A novel contribution implemented in this system is a declarative debugger of Datalog queries [CGS07,CGS08], which relies on program semantics rather than on the computation mechanism. The debugging process is usually started when the user detects an unexpected answer to a query. By asking questions about the intended semantics, the debugger looks for incorrect program relations. See Section 5.5 for further details.

Following the need for catching program errors when handling large amounts of data, we also include a test case generator for SQL correlated views [CGS10a]. Our tool can be used to generate positive, negative and both positive-negative test cases (cf. Section 5.7).

1.1 Deductive Databases

The intersection of databases, logic, and artificial intelligence delivered deductive databases. Deductive database systems are database management systems built around a logical model of data, and their query languages allow expressing logical queries. Relational database languages (where SQL is the *de-facto* standard) implement a limited form of logic whereas deductive database languages implement advanced forms of logic.

A deductive database is a system which includes procedures for defining deductive rules which can infer information (in the so-called intensional database) in addition to the facts loaded in the (so-called extensional) database. The logic model for deductive databases is closely related to the relational model and, in particular, with the domain relational calculus. Their query languages are related with the Prolog language and, mainly, with Datalog, a Prolog subset without constructed terms (in order to avoid infinite terms).

The relational algebra has been shown to be inefficient for expressing practical database queries. A main defect is the lack of recursion, which does not allow expressing recursive definitions as the transitive closure of a graph. Although the SQL standard (with origins in the relational algebra) has added recursion, it only include linear recursion.

Origins of deductive databases can be found in automatic theorem proving and, later, in logic programming. Minker [Mink87] suggested that Green and Raphael [GR68] were the pioneers in discovering the relation between theorem proving and deduction in databases. They developed several question–answer systems using a version of the Robinson resolution principle [Robi65], showing that deduction can be systematically performed in a database environment. Other pioneer systems were MRPPS [MN82], DEDUCE–2 [Chan78] and DADM [KT81]. See Section 7 for references to other current deductive database systems.

1.2 Referring to DES

Please use the following BiBTeX entry for referring to this system:

```
@techreport{des-user-manual-tr,
  author = {F. S\'aenz-P\'erez},
  title = {Datalog Educational System. User's Manual},
  institution = {Faculty of Computer Science, UCM},
  year = 2004,
  number = {139-04},
  note = {Available from http://des.sourceforge.net/}
 }
```

2. Installation

2.1 Downloading DES

You can download the system from the DES web page via the URL:

http://des.sourceforge.net/

There, you can find source distributions for several Prolog interpreters and operating systems, and executable distributions for Windows and Linux.

2.1.1 Source Distribution

Under the source distribution, there are several versions depending on the Prolog interpreter you select to run DES: Ciao Prolog [BCC97], GNU Prolog [Diaz], SICStus Prolog [SICStus], and SWI Prolog [Wiele]. However, adapting the code in the file **`des_glue.pl`**, it could be ported to any other Prolog system. (See Section 5.11.3 for porting to unsupported systems.) We have tested DES under several Prolog systems (Ciao Prolog 1.10p5, GNU Prolog 1.3.1, SICStus Prolog 4.1.2, and SWI–Prolog 5.10.0), and several operating systems (MS Windows XP/Vista/7 and Ubuntu 10.04.1).

The source distribution comes in a single archive file containing the following:

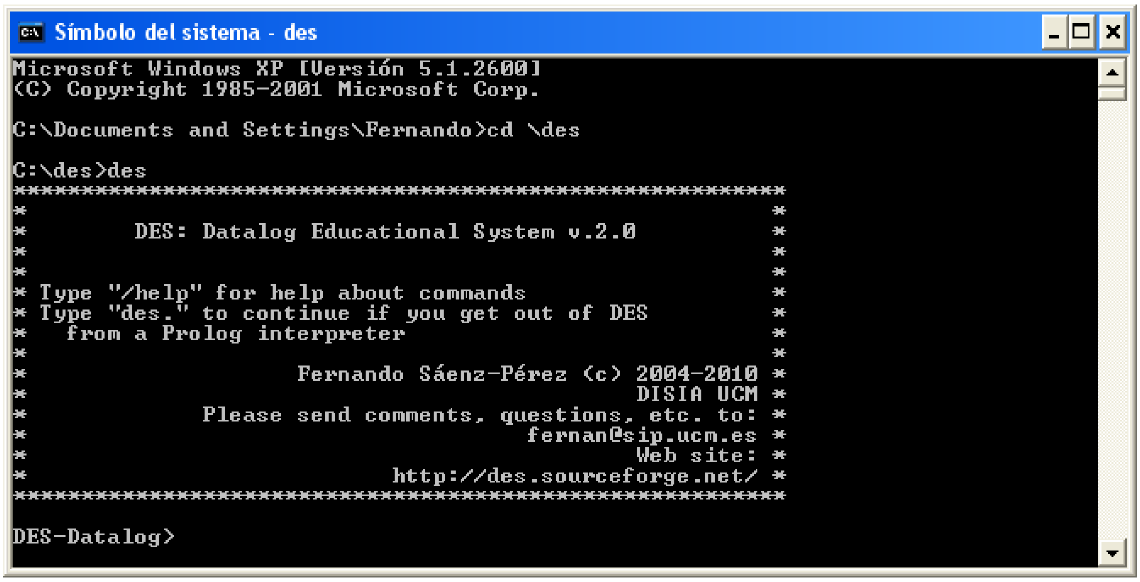
- **readmeDES<version>.txt**. A quick installation guide and file release contents
- **des.pl**. Contains the core of DES
- **des_debug.pl**. Contains the declarative debugger
- **des_sql.pl**. Contains the SQL parser and compiler
- **des_glue.pl**. Contains particular code for the selected Prolog system
- **systems/{ciao,gnu,sicstus,swi}**. Contains the same four previous files for all of the supported Prolog systems (these directories can be erased if desired, they are included only for reference)
- **doc/manualDES<version>.pdf**. This manual
- **examples/*.dl** Example files which will be discussed in Section 6
- **license/license** A verbatim copy of the GNU Public License for this distribution

2.1.2 Executable Distribution

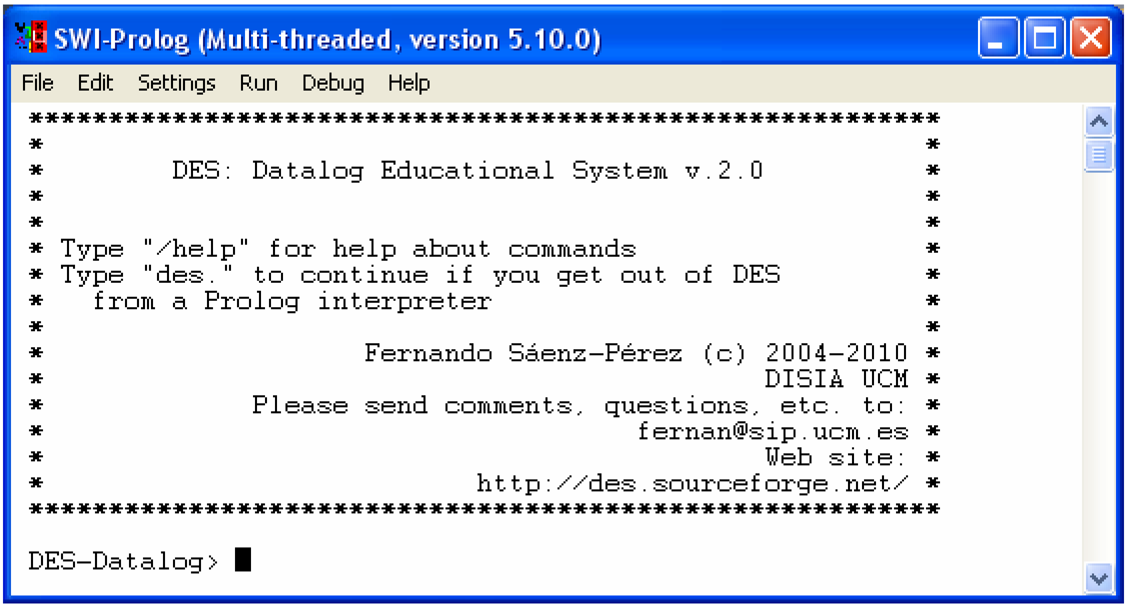
2.1.2.1 Windows

From the same URL above, you can download a Windows executable distribution in a single archive file containing the following:

- **readmeDES<version>.txt**. A quick installation guide and file release contents
- **des.exe**. Console executable file, intended to be started from a OS command shell, as depicted in the next figure:



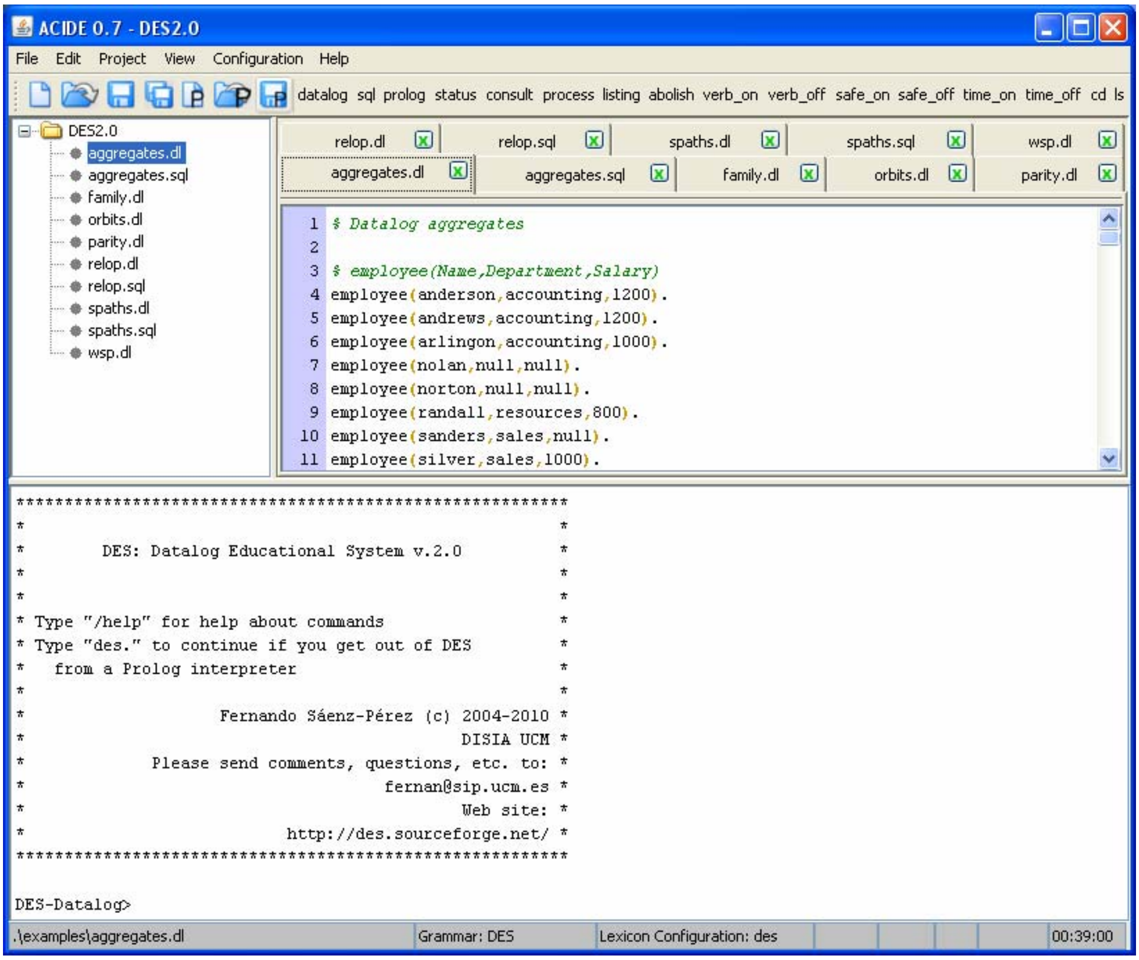
- **deswin.exe**. Windows-application executable file, as depicted below:



- ***.dll**. DLL libraries for the runtime system
- **doc/manualDES<version>.pdf**. This manual
- **examples/*.dl** Example files which will be discussed in Section 6
- **license/license** A verbatim copy of the GNU Public License for this distribution

2.1.2.2 DES+ACIDE Windows Bundle

From the same URL above, you can download a bundle including both DES and the integrated development environment ACIDE, preconfigured to work with DES. The following figure is a snapshot of the system:



2.1.2.3 Linux

From the same URL above, you can download a Linux executable distribution in a single archive file containing the following:

- **readmeDES<version>**. A quick installation guide and file release contents
- **des**. Console executable file
- **doc/manualDES<version>.pdf**. This manual
- **examples/*.dl** Example files which will be discussed in Section 6
- **license/license** A verbatim copy of the GNU Public License for this distribution

2.2 Installing and Executing DES

Unpack the distribution archive file into the directory you want to install DES, which will be referred to as the distribution directory from now on. This allows you to run the system, whether you have a Prolog interpreter or not (in this latter case, you have to run the system either on MS Windows or SunOS).

Although there is no need for further setup and you can go directly to Section 2.2.3, you can also configure a more user-friendly way for system start. In this way, you can follow two routes depending on the operating system.

2.2.1 MS Windows

2.2.1.1 Executable Distribution

Simply create a shortcut in the desktop for executing the executable of your choice: `des.exe` or `deswin.exe`. The former is a console-based executable, whereas the latter is a windows-based executable. Both have been generated under SICStus Prolog, so that all SICStus notes in the rest of this document also apply to these executables. In addition, since it is a portable application, it needs to be started from its distribution directory.

2.2.1.2 Source Distribution

Perform the following steps:

1. Create a shortcut in the desktop for running the Prolog interpreter of your choice.
2. Modify the start directory in the "Properties" dialog box of the shortcut to the installation directory for DES. This allows the system to consult the needed files at startup.
3. Append the following options to the Prolog executable path, depending on the Prolog interpreter you use:
   (a) Ciao Prolog: **`-l ciaorc`**
   (b) GNU Prolog: **`--entry-goal ['des.pl']`**
   (c) SICStus Prolog: **`-l des.pl`**
   (d) SWI Prolog: **`-g "[des]"`** (remove **`--win_app`** if present)

Another alternative is to write a batch file similar to the script file described in the next section.

2.2.2 Linux

2.2.2.1 Executable Distribution

You can create a script or an alias for executing the file **`des`** at the distribution root. This executable has been generated under SICStus Prolog, so that all SICStus notes in the rest of this document also apply to these executables. In addition, since it is a portable application, it needs to be started from its distribution directory.

2.2.2.2 Source Distribution

You can write a script for starting DES according to the selected Prolog interpreter, as follows:

(a) Ciao Prolog:

```
$CIAO –l ciaorc
```

Provided that **`$CIAO`** is the variable which holds the absolute filename of the Ciao Prolog executable.

(b) GNU Prolog:

```
$GNU --entry-goal ['des.pl']
```

Provided that **$GNU** is the variable which holds the absolute filename of the GNU Prolog executable.

(c) SICStus Prolog:

```
$SICSTUS -l des.pl
```

Provided that **$SICSTUS** is the variable which holds the absolute filename of the SICStus Prolog executable.

(d) SWI Prolog:

```
$SWI -g "[des]"
```

Provided that **$SWI** is the variable which holds the absolute filename of the SWI Prolog executable.

2.2.3 Starting DES from a Prolog interpreter

Besides the methods just described, you can start DES from a Prolog interpreter, disregarding the OS and platform, first changing to the distribution directory, and then submitting:

```
?- [des].
```

If the system does not start by itself, then type:

```
?- start.
```

3. Getting Started

Whichever method you use to start DES (a script, batch file, or shortcut, as described in Section 2.2), you get the following:

```
*********************************************************
*                                                       *
*        DES: Datalog Educational System v.2.0          *
*                                                       *
*                                                       *
* Type "/help" for help about commands                  *
* Type "des." to continue if you get out of DES         *
*   from a Prolog interpreter                           *
*                                                       *
*                    Fernando Sáenz-Pérez (c) 2004-2010 *
*                                             DISIA UCM *
*               Please send comments, questions, etc. to: *
*                                     fernan@sip.ucm.es *
*                                             Web site: *
*                           http://des.sourceforge.net/ *
*********************************************************

DES-Datalog>
```

This last line (**DES-Datalog>**) is the DES system prompt, which allows you to write commands, Datalog queries, temporary views and conjunctive queries (see next sections). If an error leads to an exit from DES and you have started from a Prolog interpreter, then you can write "**des.**" (*without* the double quotes and *with* the dot) at the Prolog prompt to continue.

There are currently three modes available for using different query interpreters: Datalog, SQL and Prolog. The first one is the default and both modes can be interchanged via the commands **/datalog**, **/sql** and **/prolog**, respectively. Anyway, if you are in a given mode, you can submit queries or goals to other interpreters simply writing the query or goal after any of the previous commands.

Data is stored in a Datalog database, including facts and rules. All queries and goals, irrespective of the language, refer to this database.

3.1 Datalog Mode

In this mode, queries are sent to the Datalog interpreter, whereas commands (cf. Section 5.10) are sent to the command processor. Both type of inputs (queries and commands) can end with an optional dot. This mode is the default and can be anyway enabled via the command **/datalog**.

The typical way of using the system is to write Datalog program files (with default extension **.dl**) and consulting them before submitting queries. Another alternative is to assert program rules from the system prompt.

Following the first alternative, you write the program in a text file, and then change to the path where the file is located by using the command **/cd *Path***, where ***Path*** is the new directory (relative or absolute). Next, the command **/consult *FileName*** is used to consult the file ***FileName***.

Provided there are a number or example files in the directory `examples` at the distribution directory, and assuming that the current path is the distribution directory (as by default), one can use the following commands to consult the example file `relop.dl`:[2]

```
DES-Datalog> /cd examples

DES-Datalog> /consult relop.dl
```

(where the default extension **.dl** can be omitted) or simply. Then, one can examine the contents of the database (see Section 5.10 for an explanation of the consulted program) via the command:

```
DES-Datalog> /listing

a(a1).
a(a2).
a(a3).
b(a1).
b(b1).
b(b2).
c(a1,a1).
c(a1,b2).
c(a2,b2).
```

[2] See section 5 for more details about commands.

```
cartesian(X,Y) :-
  a(X),
  b(Y).
difference(X) :-
  a(X),
  not(b(X)).
full_join(X,Y) :-
  fj(a(X),b(Y),X = Y).
inner_join(X) :-
  a(X),
  b(X).
left_join(X,Y) :-
  lj(a(X),b(Y),X = Y).
projection(X) :-
  c(X,Y).
right_join(X,Y) :-
  rj(a(X),b(Y),X = Y).
selection(X) :-
  a(X),
  X = a2.
union(X) :-
  a(X)
  ;
  b(X).

Info: 18 rules listed.
```

Submitting a query is pretty easy:

```
DES-Datalog> a(X)
{
  a(a1),
  a(a2),
  a(a3)
}
Info: 3 tuples computed.
```

You can interactively add new rules with the command **`/assert`**, as in:

```
DES-Datalog> /assert a(a4)
DES-Datalog> a(X)
{
  a(a1),
  a(a2),
  a(a3),
  a(a4)
}
Info: 4 tuples computed.
```

Saving the current databases, which may include such interactively added (or deleted) tuples, is allowed with the command **`/save_ddb`** ***`Filename`***, which saves in a plain file the Datalog rules in memory. Later, they can be restored with **`/restore_ddb`** ***`Filename`***. This command is only an alias for **`/consult`**. In the following session, the current database is stored, abolished (cleared), and finally restored. All the data, including the ones interactively added have been recovered:

```
DES-Datalog> /save_ddb db.dl
DES-Datalog> /abolish
DES-Datalog> /restore_ddb db.dl
DES-Datalog> a(X)
{
  a(a1),
  a(a2),
  a(a3),
  a(a4)
}
Info: 4 tuples computed.
```

Another useful command is **/list_et**, which lists, in particular, the answers already computed. Following the last series of queries and commands above, we submit:

```
Answers:
{
  a(a1),
  a(a2),
  a(a3),
  a(a4)
}
Info: 4 tuples in the answer table.
Calls:
{
  a(A)
}
Info: 1 tuple in the call table.
```

Here, we can see that the computed meaning of the queried relation is stored in the extension table, as well as the last call (see also sections 5.11.1 and 5.11.2). Unless a temporary view (see Section 4.1.7) or the command **/clear_et** is submitted, the extension table keeps those results, otherwise it is cleared.

3.2 SQL Mode

In this mode, queries are sent to the SQL processor, whereas commands (cf. Section 5.10) are sent to the command processor. SQL queries can end with an optional semicolon, and, in addition, both can end with an optional dot. This mode is enabled via the command **/sql**.

If we want to develop an analogous SQL example session to the Datalog example in the last section, we can submit the first inputs (also available in the file **examples/relop.sql**) listed below (the example is augmented to provide a first glance of SQL).

Now, answer relations to SQL queries are denoted by the relation name **answer**. Also note that lines starting by % are simply remarks[3]. If you wish to automatically reproduce the following interactive session of inputs, you can type

[3] Great, we finally added comments to SQL code.

/process examples/relop.sql (notice that you must omit **examples/** if you are in this directory already):

```
Info: Processing file 'relop.sql' ...
DES-Datalog> % Switch to SQL interpreter
DES-Datalog> /sql
DES-SQL> % Creating tables
DES-SQL> create or replace table a(a string);
DES-SQL> create or replace table b(b string);
DES-SQL> create or replace table c(a string,b string);
DES-SQL> % Listing the database schema
DES-SQL> /dbschema
Info: Table(s):
 * a(a:string(varchar))
 * b(b:string(varchar))
 * c(a:string(varchar),b:string(varchar))
Info: No views.
DES-SQL> % Inserting values into tables
DES-SQL> insert into a values ('a1');
Info: 1 tuple inserted.
DES-SQL> insert into a values ('a2');
Info: 1 tuple inserted.
DES-SQL> insert into a values ('a3');
Info: 1 tuple inserted.
DES-SQL> insert into b values ('b1');
Info: 1 tuple inserted.
DES-SQL> insert into b values ('b2');
Info: 1 tuple inserted.
DES-SQL> insert into b values ('a1');
Info: 1 tuple inserted.
DES-SQL> insert into c values ('a1','b2');
Info: 1 tuple inserted.
DES-SQL> insert into c values ('a1','a1');
Info: 1 tuple inserted.
DES-SQL> insert into c values ('a2','b2');
Info: 1 tuple inserted.
DES-SQL> % Testing the just inserted values
DES-SQL> select * from a;
answer(a.a) ->
{
  answer(a1),
  answer(a2),
  answer(a3)
}
Info: 3 tuples computed.
DES-SQL> select * from b;
answer(b.b) ->
{
  answer(a1),
  answer(b1),
  answer(b2)
}
Info: 3 tuples computed.
DES-SQL> select * from c;
```

```
answer(c.a, c.b) ->
{
  answer(a1,a1),
  answer(a1,b2),
  answer(a2,b2)
}
Info: 3 tuples computed.
DES-SQL> % Projection
DES-SQL> select a from c;
answer(c.a) ->
{
  answer(a1),
  answer(a2)
}
Info: 2 tuples computed.
DES-SQL> % Selection
DES-SQL> select a from a where a='a2';
answer(a.a) ->
{
  answer(a2)
}
Info: 1 tuple computed.
DES-SQL> % Cartesian product
DES-SQL> select * from a,b;
answer(a.a, b.b) ->
{
  answer(a1,a1),
  answer(a1,b1),
  answer(a1,b2),
  answer(a2,a1),
  answer(a2,b1),
  answer(a2,b2),
  answer(a3,a1),
  answer(a3,b1),
  answer(a3,b2)
}
Info: 9 tuples computed.
DES-SQL> % Inner Join
DES-SQL> select a from a inner join b on a.a=b.b;
answer(a) ->
{
  answer(a1)
}
Info: 1 tuple computed.
DES-SQL> % Left Join
DES-SQL> select * from a left join b on a.a=b.b;
answer(a.a, b.b) ->
{
  answer(a1,a1),
  answer(a2,null),
  answer(a3,null)
}
Info: 3 tuples computed.
DES-SQL> % Right Join
```

```
DES-SQL> select * from a right join b on a.a=b.b;
answer(a.a, b.b) ->
{
  answer(a1,a1),
  answer(null,b1),
  answer(null,b2)
}
Info: 3 tuples computed.
DES-SQL> % Full Join
DES-SQL> select * from a full join b on a.a=b.b;
answer(a.a, b.b) ->
{
  answer(a1,a1),
  answer(a1,null),
  answer(a2,null),
  answer(a3,null),
  answer(null,a1),
  answer(null,b1),
  answer(null,b2)
}
Info: 7 tuples computed.
DES-SQL> % Union
DES-SQL> select * from a union select * from b;
answer(a.a) ->
{
  answer(a1),
  answer(a2),
  answer(a3),
  answer(b1),
  answer(b2)
}
Info: 5 tuples computed.
DES-SQL> % Difference
DES-SQL> select * from a except select * from b;
answer(a.a) ->
{
  answer(a2),
  answer(a3)
}
Info: 2 tuples computed.
Info: Batch file processed.
```

3.3 Prolog Mode

This mode is enabled via the command **/prolog** and goals are sent to the Prolog interpreter. Assuming that the file **relop.dl** has been already consulted, let's consider the following example:

```
DES-Prolog> projection(X)
projection(a1)
? (type ; for more solutions, <Intro> to continue) ;
projection(a1)
? (type ; for more solutions, <Intro> to continue) ;
projection(a2)
```

```
? (type ; for more solutions, <Intro> to continue) ;
no

DES-Prolog> /datalog projection(X)
{
  projection(a1),
  projection(a2)
}
Info: 2 tuples computed.
```

The execution of this goal allows to noting the basic differences between the Prolog and Datalog engines. First, the former searches solutions, one-by-one, that satisfy the goal **`projection(X)`**. The latter gives the whole meaning[4] of the user-defined relation **`projection`** with the query **`projection(X)`** at a time. And, second, note the default set-oriented behaviour of the Datalog engine, which discards duplicates in the answer. However, it is possible to allow duplicates as follows:

```
DES-Prolog> /duplicates on
Info: Duplicates are on.

DES-Prolog> /datalog projection(X)
{
  projection(a1),
  projection(a1),
  projection(a2)
}
Info: 3 tuples computed.
```

3.4 Getting Help

You can get useful information with the following commands:

- **`/help.`** Shows the list of available commands, which are explained in Section 5.10.
- **`/builtins.`** Shows the list of built-ins, which are explained in Section 4.4.

Also, visit the URL for last information:

`http://des.sourceforge.net/`

Finally, you can contact the author via the e-mail address:

`fernan@sip.ucm.es`

4. Query Languages

DES has evolved from a quite simple Datalog interpreter to its current state, which relies on a deductive database engine which can be queried with both Datalog and SQL languages. In addition, a Prolog interface is also provided in order to highlight the differences between Datalog and Prolog systems. Since DES is intended to students, it has no full-blown features of either state-of-the-art Prolog, Datalog or

[4] The meaning of a relation is the set of facts inferred both extensionally and intensionally from the program.

SQL-based systems. However, it has many features that make it interesting as an educational tool, along with the novel implementations of declarative debugging (Section 5.5) and the test case generator (Section 5.7). In this section, we describe its three query languages: Datalog, SQL and Prolog.

The database engine is shared by all the query languages, so that queries or goals can refer to any object defined using any language. However, there are some dependent issues that must be taken into account. For instance, once a Datalog fact is loaded into the database, the relation it defines can be queried in Datalog. But, the same relation has to be defined in SQL via a **`create table`** statement if you wish to query the relation from SQL. This particular issue comes from the fact that Datalog relations have unnamed attributes, and a positional reference is used for accessing those relations. In turn, SQL uses a notational syntax, giving names to relation argumts. To illustrate this, let's consider the following session:

```
DES-Datalog> /assert t(1)
DES-Datalog> t(X)
{
  t(1)
}
Info: 1 tuple computed.
DES-Datalog> /sql
DES-SQL> select * from t;
Error: Input processing error.
DES-SQL> create table t(a int);
DES-SQL> select * from t;
answer(t.a) ->
{
  answer(1)
}
Info: 1 tuple computed.
```

The error above reflects that **`t`** is not a known object in the database scheme (SQL parsing errors are not identified by the system, yet).

4.1 Datalog

Since Datalog stems from Prolog, we have adopted almost all the Prolog syntax conventions for writing Datalog programs (the reader is assumed to have basic knowledge about Prolog). We allow (recursive) Datalog programs with stratified negation [Ullm95], i.e., normal logic programs without function symbols. Stratification is imposed to ensure a clear semantics when negation is involved, and function symbols are not allowed in order to guarantee termination of queries, a natural requirement with respect to a (relational) database user who is not able to deal with compound data.

Commands are somewhat different for Prolog programmers as they are accustomed to (see Section 5.10). Also, exceptions are noted when necessary.

4.1.1 Syntax

Definitions for Datalog mainly come from the field of Logic Programming. Here, we follow mainly [Lloy87], referring the reader to this book for a more general presentation of Logic Programming. DES syntax is borrowed from Prolog. Next, some definitions for understanding the syntax of programs, queries and views are introduced.

- Numbers. Integers and float numbers are allowed. A number is a float whenever the number contains a dot (`.`) between two digits. The range depends on the Prolog platform being used. Negative numbers are identified by a preceding minus (`-`), as usual.

  Scientific notation is supported as: **`aEb`**, where **`a`** is a fractional number (always including a dot), and **`b`** is an integer, which may start with **`+`** or **`-`** (but it is not required).

  Examples of numbers are **`1`**, **`1.1`**, **`-1.0`**, **`1.2E34`**, **`1.2E+34`**, and **`1.2E-34`**.

  Note that **`-1.`**, **`+1`**, **`.1`**, **`1.E23`**, and **`1E23`** are not valid numbers. A plus sign is not part of a positive number; however, a minus sign can be used as a prefix unary operator in arithmetical expressions (cf. Section 4.4.4.1) and also following the symbol **`E`** in scientific notation, as already seen.

- Constants. A constant can be:
  - A number (integer or float).
  - Any sequence of alphanumeric characters (including the underscore `_`), starting with a lowercase letter
  - Any sequence of characters delimited by single quotes.

  Examples of alphanumeric constants are **`foo`**, **`foo_foo`**, **`'foo foo'`**, **`'2*3'`**, and **`'X'`**.

- Variables. Variables are written with alphanumeric characters, and alternatively start with either an uppercase or with an underscore (`_`). Singleton, anonymous variables are also allowed, which are denoted with a single underscore. Each occurrence of an anonymous variable is considered different from any other anonymous variable. For instance, in the rule **`a :- b(_),c(_)`**, **`b`** and **`c`** do not share variables.

  Examples of variables are: **`X`**, **`_X`**, **`_var`**, and **`_`**.

- Unknowns. Unknowns are represented as null values and are written alternatively as both **`null`** and **`'$NULL'(`*`ID`*`)`**, where ***`ID`*** is a unique identifier. The first form is used for normal users, whilst the second one is intended for development uses (cf. **`development`** command in Section 5.10.7).

- Terms. Terms can be:
  - Noncompound. Variables or constants.
  - Compound. As in Prolog, they have the form **`t(t1, ..., tn)`**, where **`t`** is a function symbol (functor), and **`ti`** (**`1`** ≤ **`i`** ≤ **`n`**) are terms.

  Up to the current version, compound terms can only occur in arithmetic expressions. Their function symbols can be any of the built-in arithmetic operators and functions (cf. Section 4.4.2). These operators can be:
  - Infix, as the addition (e.g., **`1+2`**)

  - Prefix, as bitwise negation (e.g., **`\1`**)

  Examples of terms are: **`r(p)`**, and **`p(X,Y)`**, and **`X > Y`**.

- Atoms. An atom has the form **`a(t1, ..., tn)`**, where **`a`** is a predicate (relation) symbol, and **`ti`** (**0** ≤ **i** ≤ **n**) are terms. If **`i`** is **0**, then the atom is simply written as **`a`**.

  Positive, ground atoms are used to build the Herbrand universe.

  There are several built-in predicates: **`is`** (for evaluating arithmetical expressions), arithmetic functions, (infix and prefix) operators and constants, and comparison operators. Comparison operators are infix, as "less-than". For example, **`1 < 2`** is a positive atom built from an infix built-in comparison operator (see Section 4.4.1).

  Examples of atoms are: **`p`**, **`r(a,X)`**, **`1 < 2`**, and **`X is 1+2`**.

  Note that **`p(1+2)`** and **`p(t(a))`** are not valid atoms.

- Conditions. A condition is a Boolean expression containing conjunctions (**`,/2`**), disjunctions (**`;/2`**), built-in comparison operators, constants and variables.

  Four examples of conditions are: **`X>1`**, **`X=Y`**, **`(X>Y,Y>Z)`**, **`(X=<Y;Z<0)`**.

  Note that **`X>Y+Z`** is now supported; it can be solved whenever the rule where it occurs is safe (cf. Section 5.2).

- Relation functions. A function has the form **`f(a1, …, an)`**, where **`f`** is a function name, **`ai`** are its arguments, and maps to a relation. Only built-in functions are allowed. The current provision of built-in functins is as follows:
  - **`not(a)`**. Intended for computing the negation of its single argument **`a`**.
  - **`lj(a1,a2,a3)`**. Intended for computing the *left* outer join of the relations **`a1`** (left relation) and **`a2`** (right relation), committing the condition (Boolean expression) **`a3`** (join condition).
  - **`rj(a1,a2,a3)`**. Intended for computing the *right* outer join of the relations **`a1`** (left relation) and **`a2`** (right relation), committing the condition (Boolean expression) **`a3`** (join condition).
  - **`fj(a1,a2,a3)`**. Intended for computing the *full* outer join of the relations **`a1`** (left relation) and **`a2`** (right relation), committing the condition (Boolean expression) **`a3`** (join condition).

  Note that outer join functions can be nested.

- Literals. Literals can be:
  - Positive. An atom.
  - Negative. A negated atom of the form **`not(a(t1, ..., tn))`**, where **`a(t1, ..., tn)`** is an atom. Negative literals are used to express the negation of a relation (either as a query or as a part of a rule body).
  - Disjunctive. A disjunctive literal is of the form **`l;r`** , where **`l`** and **`r`** are literals.

  Examples of literals are: **`p`**, **`r(a,X)`**, **`not(q(X,b))`**, **`r(a,X);not(q(X,b))`**, **`1 < 2`**, and **`X is 1+2`**.

  Note that **`not(p,q)`** is not a valid literal.

A literal can occur in rule bodies, queries, and view bodies.

4.1.2 Rules

Datalog rules have the form **`head :- body`**, or simply **`head`**. Both end with a dot. A Datalog head is a positive atom that uses no built-in predicate symbol. A Datalog body contains a comma-separated sequence of literals which may contain built-in symbols as listed in Section 4.4, as well as disjunctions (**`;/2`**).

4.1.3 Programs

DES programs consist of a multiset of rules. Programs may contain remarks. A remark starts with the symbol `%`, and ends at the end of line.

4.1.4 Queries

A (positive) query is the name of a relation with as many arguments as the arity of the relation (a positive literal). Each one of these arguments can be a variable or a constant; a compound term is not allowed but as an arithmetic expression. Built-in relations may require relations and conditions as arguments. A negative query is written as **`not(PositiveQuery)`**, i.e., a negated relation (a negative literal)

Queries are typed at the DES system prompt. The answer to a query is the (multi)set of atoms matching the query which are deduced in the context of the program, from both the extensional and intensional database. A query with variables for all the arguments of the queried relation gives the whole set of facts defining the relations, as the query **`a(X)`** in the example of Section 3. If a query contains a constant in an argument position, it means that the query processing will select the facts from the meaning of the relation such that the argument position matches with the constant (i.e., analogous to a select relational operation). This is the case of the query **`a(a3)`** in the same example.

You can also write conjunctive queries on the fly, such as **`a(X), b(X)`** (see Section 4.1.7). Built-in comparison operators (listed in Section 4.4.1) can be safely used in queries whenever their arguments are ground at evaluation time (excepting equality, which performs unification). Disjunctive queries are also allowed, too, such as **`a(X); b(X)`**. Concluding, a query follows the same syntax as rule bodies.

4.1.5 Duplicates

Duplicates in answers are removed by default. However, it is also possible to enable them with the command **`/duplicates on`**. This allows to generate answers as multisets instead of as the common set-oriented deductive systems behave. Computing the meaning of a relation containing duplicates in the extensional database (i.e., its facts) will include all of them in the answer, as in:

```
DES-Datalog> /duplicates on
DES-Datalog> /assert t(1)
DES-Datalog> /assert t(1)
DES-Datalog> t(X)
{
  t(1),
  t(1)
```

```
}
Info: 2 tuples computed.
```

Rules can also be source of duplicates, as in:

```
DES-Datalog> /assert s(X):-t(X)
DES-Datalog> s(X)
{
  s(1),
  s(1)
}
Info: 2 tuples computed.
```

In addition, recursive rules are duplicate sources, as in:

```
DES-Datalog> /assert t(X):-t(X)
DES-Datalog> t(X)
{
  t(1),
  t(1),
  t(1),
  t(1)
}
Info: 4 tuples computed.
```

where two tuples directly come from the two facts for **t/1**, and the other two from the single recursive rule. Again, adding the same recursive rule yields:

```
DES-Datalog> /assert t(X):-t(X)
DES-Datalog> t(X)

{
  t(1),
  t(1),
  t(1),
  t(1),
  t(1),
  t(1),
  t(1),
  t(1),
  t(1),
  t(1)
}
Info: 10 tuples computed.
```

where this answer contains the outcome due to: two tuples directly from the two facts, and four tuples for each recursive rule. The first recursive rule is source of four tuples because of the two facts and the two tuples from the second recursive rule. Analogously, the second recursive rule is source of another four tuples: two facts and the two tuples from the first recursive rule.

The rule of thumb to understand duplicates in recursive rules is to consider all possible computation paths in the dependency graph, stopping when a (recursive) node already used in the computation is reached.

4.1.6 Temporary Views

Temporary views allow you to write conjunctive queries on the fly. A temporary view is a rule which is added to the database; its head is considered as a query and executed. Afterwards, the rule is deleted. Temporary views are useful for quickly submitting conjunctive queries. For instance, the view:

```
DES-Datalog> d(X) :- a(X), not(b(X))
```

computes the set difference between the sets **a** and **b**, provided they have been already defined.

Note that the view is evaluated in the context of the program; so, if you have more rules already defined with the same name and arity of the rule's head, the evaluation of the view will return its meaning under the whole set of rules matching the query. For instance:

```
DES-Datalog> a(X) :- b(X)
```

computes the set union of the sets **a** and **b**., provided they have been already defined.

4.1.7 Automatic Temporary Views

Automatic temporary views, shortly autoviews, are temporary views which do not need a head and allows you to write conjunctive queries on the fly. When you write a conjunctive query, a new temporary relation, named **autoview**, is built with as many arguments as variables occur in the conjunctive query. **autoview** is a reserved word and cannot be used for defining other relation. As an example of an autoview, let's consider:

```
DES-Datalog> a(X),b(Y)

Info: Processing:
  answer(X,Y) :-
    a(X),
    b(Y).
{
  answer(a1,a1),
  answer(a1,b1),
  answer(a1,b2),
  answer(a2,a1),
  answer(a2,b1),
  answer(a2,b2),
  answer(a3,a1),
  answer(a3,b1),
  answer(a3,b2)
}
Info: 9 tuples computed.
```

which computes the Cartesian product of the relations **a** and **b**, provided they have been already defined as:

```
  a(a1).
  a(a2).
```

```
  a(a3).
  b(b1).
  b(b2).
  b(a1).
```

4.1.8 Negation

DES ensures that negative information can be gathered from a program with negated goals provided that a restricted form of negation is used: Stratified negation. This broadly means that negation is not involved in a recursive computation path, although it can use recursive rules. The following program[5] illustrates this point:

```
a :- not(b).
b :- c,d.
c :- b.
c.
```

The query **a** succeeds with the meaning **{a}**. Observe also that **not(a)** does not succeed, i.e., its meaning is the empty set.

DES provides two different algorithms for computing negation: **strata** (a default algorithm following a bottom-up top-down-guided stratum saturation) and **et_not** (taken from [SD91]), which are selected via the command **/negation** ***Algorithm***. (cf. Section 5.10.9).

If you are interested in how programs with negation are solved for the algorithm **strata**, you can find useful the following commands (cf. Section 5.10.7):

```
DES-Datalog> /pdg

Nodes: [d/0,a/0,b/0,c/0]
Arcs : [a/0-b/0,c/0+b/0,b/0+d/0,b/0+c/0]

DES-Datalog> /strata

[(d/0,1),(a/0,2),(b/0,1),(c/0,1)]
```

The first command shows the predicate dependency graph (see, e.g., [ZCF+97]) for the loaded program. First, nodes in the graph are shown in a list whose elements P are predicates with their arities with the form predicate/arity. Next, arcs in the graph are shown in a list whose elementes are either P+Q or P-Q, where P and Q are nodes in the graph. An arc P+Q means that there exists a rule such that P is the predicate for its head, and Q is the predicate for one of its literals. If the literal is negated, the arc is negative, which is expressed as P-Q. The graph for this program can be depicted as in Figure 3.

[5] In file **negation.dl**, located at the **examples** distribution directory. Adapted from [RSSWF97].

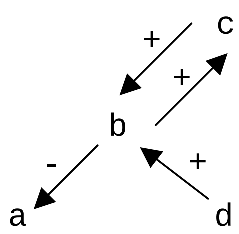


Figure 3. Predicate Dependency Graph for **`negation.dl`**

The second command shows the stratum assigned to each predicate. This assignment is computed by following an algorithm based on [Ullm95], but modified for taking advantage of the predicate dependency graph. Strata are shown as a list of pairs (P,S), where P is a predicate and S is its assigned stratum. In this example, all of the program predicates are in stratum 1 but **`a`**, which is assigned to stratum 2. This means that if the meaning of **`a`** is to be computed, then the meanings of predicates in lower strata (and only those predicates **`a`** depends on) have to be firstly computed.

Since the algorithm **`strata`** does not follow a naïve bottom-up solving, only the meanings of required predicates are computed. To illustrate this, consider the query **`b`** for the same program. DES computes the predicate dependency subgraph for **`b`**, i.e., all of the predicates which are reachable from **`b`**, and, then, a stratification is computed. Notice the different information given by the system for solving the queries **`a`** and **`b`** (here, verbose output is currently enabled, which is the default behaviour):

```
DES-Datalog> a
Info: Computing by stratum of [b].
{
  a
}

DES-Datalog> b
{
}
```

For the goal **`a`**, the system informs that **`b`** is previously computed (nevertheless taking advantage of the extension table mechanism), whereas for the goal **`b`** there is no need of resorting to the stratum-by-stratum solving.

4.1.9 Null Values

The null value is included in each program signature for denoting unknowns, in a similar way it is an inherent part of current relational database systems. Comparing null values in Datalog opens a new scenario: Two null values are not (known to be) equal, and are (not known to be) distinct. The following illustrates this expected behaviour:

```
DES-Datalog> null=null
{
}
Info: 0 tuples computed.

DES-Datalog> null\=null
```

```
{
}
Info: 0 tuples computed.
```

A null value is internally represented as **'$NULL'(*ID*)**, where ***ID*** is a unique identifier (an integer). Development listings (enabled via the command **/development on**) allow to inspect these identifiers, such as in:

```
DES-Datalog> /development on
DES-Datalog> p(X,Y):-X=null,Y=null,X=Y
Info: Processing:
  p(X,Y) :-
    X = '$NULL'(14),
    Y = '$NULL'(15),
    X = Y.
{
}
Info: 0 tuples computed.
DES-Datalog> p(X,Y):-X=null,Y=null,X\=Y
Info: Processing:
  p(X,Y) :-
    X = '$NULL'(16),
    Y = '$NULL'(17),
    X \= Y.
{
}
Info: 0 tuples computed.
```

The builtin predicate **is_null/1** tests whether its single argument is a null value:

```
DES-Datalog> is_null(null)
{
  is_null(null)
}
Info: 1 tuple computed.

DES-Datalog> X=null,is_null(X)

Info: Processing:
  answer(X) :-
    X = null,
    is_null(X).
{
  answer(null)
}
Info: 1 tuple computed.
```

You have also available its counterpart predicate: **is_not_null/1**.

Note that from a system implementor point of view, nulls can never unify because they are represented by different ground terms. On the other hand, disequality is explicitly handled in order to fail when comparing nulls.

4.1.10 Outer Joins

Three outer join operations are provided (cf. Section 4.4.6), following relational database query languages (SQL, extended relational algebra): left, right and full outer join. Having loaded the example program **relop.dl**, we can submit the following queries:

```
DES-Datalog> /c relop
DES-Datalog> /listing a
a(a1).
a(a2).
a(a3).
DES-Datalog> /listing b
b(a1).
b(b1).
b(b2).
DES-Datalog> lj(a(X),b(Y),X=Y)
Info: Processing:
  answer(X,Y) :-
    lj(a(X),b(Y),X = Y).
{
  answer(a1,a1),
  answer(a2,null),
  answer(a3,null)
}
Info: 3 tuples computed.
DES-Datalog> rj(a(X),b(Y),X=Y)
Info: Processing:
  answer(X,Y) :-
    rj(a(X),b(Y),X = Y).
{
  answer(a1,a1),
  answer(null,b1),
  answer(null,b2)
}
Info: 3 tuples computed.
DES-Datalog> fj(a(X),b(Y),X=Y)
Info: Processing:
  answer(X,Y) :-
    fj(a(X),b(Y),X = Y).
{
  answer(a1,a1),
  answer(a1,null),
  answer(a2,null),
  answer(a3,null),
  answer(null,a1),
  answer(null,b1),
  answer(null,b2)
}
Info: 7 tuples computed.
```

Note that the third parameter is the join condition. Be aware and do not miss a where condition with a join condition. Let´s consider the above query **lj(a(X),b(Y),X=Y)**. Do not expect the same result as above for the following query:

```
DES-Datalog> lj(a(X),b(X),true)
Info: Processing:
  answer(X) :-
    lj(a(X),b(X),true).
{
  answer(a1)
}
Info: 1 tuple computed.
```

Here, the same variable **X** for the relations **a** and **b** means that tuples from **a** and **b** with the same value are to be joined, as in the next equivalent query:

```
DES-Datalog> lj(a(X),b(Y),true),X=Y
Info: Processing:
  answer(X,Y) :-
    lj(a(X),b(Y),true),
    X = Y.
{
  answer(a1,a1)
}
Info: 1 tuple computed.
```

Outer join relations can be nested as well:

```
DES-Datalog> lj(a(X),rj(b(Y),c(U,V),Y=U),X=Y)
Info: Processing:
  answer(X,Y,U,V) :-
    lj(a(X),rj(b(Y),c(U,V),Y = U),X = Y).
{
  answer(a1,a1,a1,a1),
  answer(a1,a1,a1,b2),
  answer(a2,null,null,null),
  answer(a3,null,null,null)
}
Info: 4 tuples computed.
```

Note that compoud conditions must be enclosed between parentheses, as in:

```
DES-Datalog> lj(a(X),c(U,V),(X>U;X>V))
Info: Processing:
  answer(X,U,V)
in the program context of the exploded query:
  answer(X,U,V) :-
    lj(a(X),c(U,V),(X > U;X > V)).
{
  answer(a1,null,null),
  answer(a2,a1,a1),
  answer(a2,a1,b2),
  answer(a3,a1,a1),
  answer(a3,a1,b2),
  answer(a3,a2,b2)
}
Info: 6 tuples computed.
```

4.1.11 Aggregates

Aggregates refer to functions and predicates that compute values with respect to a collection of values instead of a single value. Aggregates are provided by means of five usual computations: **`sum`** (cumulative sum), **`count`** (element count), **`avg`** (average), **`min`** (minimum element), and **`max`** (maximum element). In addition, the less usual **`times`** (cumulative product) is also provided. They behave close to most SQL implementations, i.e., ignoring nulls.

4.1.11.1 Aggregate Functions

An aggregate function can occur in expressions and returns a value, as in **`R=1+sum(X)`**, where **`sum`** is expected to compute the cumulative sum of possible values for **`X`**, and **`X`** has to be bound in the context of a **`group_by`** predicate (cf. next section), wherein the expression also occur.

4.1.11.2 Group_by Predicate

A **`group_by`** predicate encloses a query for which a given list of variables builds answer sets (groups) for all possible values of these variables. Let's consider the following excerpt from the file **`aggregates.dl`**:

```
% employee(Name,Department,Salary)
employee(anderson,accounting,1200).
employee(andrews,accounting,1200).
employee(arlingon,accounting,1000).
employee(nolan,null,null).
employee(norton,null,null).
employee(randall,resources,800).
employee(sanders,sales,null).
employee(silver,sales,1000).
employee(smith,sales,1000).
employee(steel,sales,1020).
employee(sullivan,sales,null).
```

We can count the number of employees for each department with the following query:

```
DES-Datalog> group_by(employee(N,D,S),[D],R=count)
Info: Processing:
  answer(D,R) :-
    group_by(employee(N,D,S),[D],R = count).
{
  answer(accounting,3),
  answer(null,2),
  answer(resources,1),
  answer(sales,5)
}
Info: 4 tuples computed.
```

Note that two employees are not assigned to any department yet (**`nolan`** and **`norton`**). This query behaves as a SQL user would expect, though nulls do not have to represent the same data value (in spite of this, such tuples are collected in the same bag).

If we rather want to count *active* employees (those with assigned salaries), we pose the following query:

```
DES-Datalog> group_by(employee(N,D,S),[D],R=count(S))
Info: Processing:
  answer(D,R) :-
    group_by(employee(N,D,S),[D],R = count(S)).
{
  answer(accounting,3),
  answer(null,0),
  answer(resources,1),
  answer(sales,3)
}
Info: 4 tuples computed.
```

Note that null departments have no employee with assigned salary.

Conditions including aggregates on groups can be stated as well (cf. having conditions in SQL). For instance, the following query counts the active employees of departments with more than one employee.

```
DES-Datalog> group_by(employee(N,D,S),[D],count(S)>1)
Info: Processing:
  answer(D) :-
    group_by(employee(N,D,S),[D],(A = count(S),A > 1)).
{
  answer(accounting),
  answer(sales)
}
Info: 2 tuples computed.
```

Note that the number of employees can also be returned, as follows:

```
DES-Datalog> group_by(employee(N,D,S),[D],(R=count(S),R>1))
Info: Processing:
  answer(D,R) :-
    group_by(employee(N,D,S),[D],(R = count(S),R > 1)).
{
  answer(accounting,3),
  answer(sales,3)
}
Info: 2 tuples computed.
```

Conditions including no aggregates on tuples of the input relation (cf. SQL **FROM** clause) can also be used (cf. **WHERE** conditions in SQL). For instance, the following query computes the number of employees whose salary is greater than 1,000.

```
DES-Datalog> group_by((employee(N,D,S),S>1000),[D],R=count(S))
Info: Processing:
  answer(D,R)
in the program context of the exploded query:
  answer(D,R) :-
    group_by('$p2'(S,D,N),[D],R = count(S)).
  '$p2'(S,D,N) :-
    employee(N,D,S),
```

```
    S > 1000.
{
  answer(accounting,2),
  answer(sales,1)
}
Info: 2 tuples computed.
```

Note that the following query is not equivalent to the former, since variables in the input relation are not bound after a grouping computation. The following query illustrates this situation, which generates a run-time exception (this could also be avoided with a compile-time analysis, that is not implemented yet).

```
DES-Datalog> group_by(employee(N,D,S),[D],R=count(S)), S>1000
Info: Processing:
  answer(D,R) :-
    group_by(employee(N,D,S),[D],R = count(S)),
    S > 1000.
Exception: Non ground argument(s) found in goal S > 1000 in the
instanced rule:
...
```

The predicate **group_by** is more expressive (admits a more compact representation) than its SQL counterpart. Let's consider the following Datalog session:

```
DES-Datalog> /assert p(1,1)
DES-Datalog> /assert p(2,2)
DES-Datalog> /assert q(X,C):-
group_by(p(X,Y),[X],(C=count;C=sum(Y)))
DES-Datalog> q(X,C)
Info: Computing by stratum of [p(A,B)].
{
  q(1,1),
  q(2,1),
  q(2,2)
}
Info: 3 tuples computed.
```

An analogous SQL session follows:

```
DES-SQL> create table p(X int, Y int)

DES-SQL> create view q(X,C) as (select X,count(Y) as C from p
group by X) union (select X, sum(Y) as C from p group by X)

DES-SQL> select * from q
answer(q.X, q.C) ->
{
  answer(1,1),
  answer(2,1),
  answer(2,2)
}
Info: 3 tuples computed.
```

4.1.11.3 Aggregate Predicates

An aggregate predicate returns its result in its last argument position, as in **sum(p(X),X,R)**, which binds **R** to the cumulative sum of values for **X**, provided by the input relation **p**. These aggregate predicates simply allow another way of expressing aggregates, in addition to the way explained just above. Again, with the same file, the following queries are allowed:

```
DES-Datalog> count(employee(N,D,S),S,T)
Info: Processing:
  answer(T) :-
    count(employee(N,D,S),S,[],T).
{
  answer(7)
}
Info: 1 tuple computed.
```

A group_by operation is simply specified by including the grouping variable(s) in the head of a clause, as in the following view, which computes the number of active employees by department:

```
DES-Datalog> c(D,C):-count(employee(N,D,S),S,C)
Info: Processing:
  c(D,C) :-
    count(employee(N,D,S),S,[D],C).
{
  c(accounting,3),
  c(null,0),
  c(resources,1),
  c(sales,3)
}
Info: 4 tuples computed.
```

Note that the system adds to the aggregate predicate an argument with the list of grouping variables, which are the ones occurring in the first argument of the aggregate predicate that also occur in the head. This code translation is required for the aggregate predicate to be compute, although such form has not been made available to the user.

Having conditions are also allowed, including them as another goal of the first argument of the aggregate predicate as, for instance, in the following view, which computes the number of employees that earn more than the average:

```
DES-Datalog>
count((employee(N,D,S),avg(employee(N1,D1,S1),S1,A),S>A),C)
Info: Processing:
  answer(C)
in the program context of the exploded query:
  answer(C) :-
    count('$p2'(A,S,D,N),[],C).
  '$p2'(A,S,D,N) :-
    employee(N,D,S),
    avg(employee(N1,D1,S1),S1,[],A),
    S > A.
{
```

```
  answer(2)
}
Info: 1 tuple computed.
```

Note that this query uses different variables in the same argument positions for the two occurrences of the relation **employee**. Compare this to the following query, which computes the number of employees so that each one of them earns more than the average salary of his corresponding department. Here, the same variable name **D** has been used to refer to the department for which the counting and average are computed:

```
DES-Datalog>
count((employee(N,D,S),avg(employee(N1,D,S1),S1,A),S>A),C)
Info: Processing:
  answer(C)
in the program context of the exploded query:
  answer(C) :-
    count('$p2'(A,S,N),[],C).
  '$p2'(A,S,N) :-
    employee(N,D,S),
    avg(employee(N1,D,S1),S1,[],A),
    S > A.
{
  answer(3)
}
Info: 1 tuple computed.
```

Also, as a restriction of the current implementation, keep in mind that having conditions including aggregates (as the one including the average computations above) can only occur in the first argument of an aggregate. The following query, which should be equivalent to the last one, would generate a run-time exception:

```
DES-Datalog>
avg(employee(N1,D,S1),S1,A),count((employee(N,D,S),S>A),C)
Error: S > A will raise a computing exception at run-time.
```

Finally, recall that expressions including aggregate functions are not allowed in conjunction with aggregate predicates, but only in the context of a **group_by** predicate.

4.1.12 Disjunctive Bodies

As introduced in Section 4.1.1, rule bodies can contain disjunctions, such as the the one contained in the program **family.dl**:

```
parent(X,Y) :-
  father(X,Y)
  ;
  mother(X,Y).
```

This clause is equivalent to:

```
parent(X,Y) :-
  father(X,Y).
parent(X,Y) :-
  mother(X,Y).
```

If you list the database contents via the command **`/listing`** you will get the first form when development listings are off (via the command **`/development off`**). Otherwise, you get the second one (command **`/development on`**).

Datalog views and autoviews containing disjunctive bodies are allowed, and the system informs about the program transformation needed to compute them. For instance, you can directly submit the rule above as a view at the DES prompt:

```
DES-Datalog> parent(X,Y) :- father(X,Y) ; mother(X,Y)
Info: Processing:
  parent(X,Y)
in the program context of the exploded query:
  parent(X,Y) :-
    father(X,Y).
  parent(X,Y) :-
    mother(X,Y).
{
  parent(amy,fred),
  parent(carolI,carolII),
  parent(carolII,carolIII),
  parent(fred,carolIII),
  parent(grace,amy),
  parent(jack,fred),
  parent(tom,amy),
  parent(tony,carolII)
}
Info: 8 tuples computed.
```

4.2 SQL

The syntax recognized by the interpreter is borrowed from the SQL standard. This section describes the main limitations, features, and decisions taken in designing SQL, which coexists with Datalog. Also, we describe the three main parts of the supported subset of the SQL language: DDL (Data Definition Language, for defining the database schema), DQL (Data Query Language, for listing contents of the database), and DML (Data Manipulation Language, for inserting and deleting tuples). Section 4.2.7 resumes the SQL grammar.

4.2.1 Main Limitations

- A **`select all`** statement performs equal to a **`select distinct`** one, even when duplicates occur.
- The projection list consists of column references (***`column`***, ***`table.column`***, ***`alias.column`***), wildcards (**`*`**, ***`table.*`***, ***`alias.*`***), alias references, and arithmetic expressions. Other expressions may be supported in further releases.
- A limited coverage of database integrity constraints.
- No provision for ordering results (**`order by`** clause).
- No insertions/deletions/updates into views.
- No syntax error reports. The parser does not inform about the possible syntax error cause.

- One line statements, avoiding a clearer writing of complex statements which are best understood in several split lines.

4.2.2 Main Features

As main features, we highlight:

- Data query, data definition, and data manipulation language parts provided.
- Subqueries (nested queries without depth limits).
- Correlated queries (tables and relations in nested subqueries can be referenced by the host query). For example: **`select * from t,(select a from s) where t.a=s.a`**.
- Subqueries in comparisons, as **`select a from t where t.a > (select a from s)`**.
- Table, relation, and expression aliases with full scope.
- Non-linear recursive queries (using the standard syntax).
- Set operators build relations, which can be used wherever a data source is expected (**`from`** clause).
- Null values are supported, along with outer joins (full, left and right).
- Aggregate functions allowed in expressions at the projection list and 'having' conditions. Group_by clauses are also allowed.
- View support. Any relation built with a SQL query can be defined as a view (even recursive queries).
- Supported database integrity constraints include type constraints, primary keys, and referential integrity constraints.
- Parentheses can be used elsewhere they are needed and also for easing the reading of statements.

4.2.3 Datalog vs. SQL

With respect to Datalog, some decisions have been taken:

- As in Datalog, user identifiers are case-sensitive (table and attribute names, ...). This is not the normal behaviour of current relational database systems.
- In contrast to Datalog, built-in identifiers are not case-sensitive. This conforms to the normal behaviour of current relational database systems.

4.2.4 Data Definition Language

This part of the language deals with creating (or replacing), and dropping tables and views. There is no provision for updating the schema, which can be consulted with the command **`/dbschema`**.

4.2.4.1 Creating Tables

CREATE [OR REPLACE] TABLE *TableName*(*Column1 Type1* [*ColumnConstraint1*], ..., *ColumnN TypeN* [*ColumnConstraintN*] [, *TableConstraints*])

This statement defines the table schema with name ***TableName*** and column names ***Column1***, ..., ***ColumnN***., with types ***Type1***, ..., ***TypeN***, respectively. If the optional clause **OR REPLACE** is used, the table is dropped if existed already, deleting all of its tuples.

There is provision for a couple of column constraints:

- **PRIMARY KEY**. Primary key constraint for only one column
- **REFERENCES *TableName*[(*Column*)]**. Referential integrity constraint for only one column

Also, there is provision for a couple of table constraints:

- **PRIMARY KEY (*Column*,..., *Column*)** . Primary key constraint for one or more columns
- **FOREIGN KEY (*Column*,..., *Column*) REFERENCES *TableName*[(*Column*,..., *Column*)])]**. Referential integrity constraint for one or more columns

Allowed types include:

- **CHAR**. Fixed-length string of 1
- **CHAR(*n*)**. Fixed-length string of *n* characters
- **VARCHAR(*n*)**. Variable-length string of up to n characters
- **VARCHAR** (or **STRING**). Variable-length string of up to the maximum length of the underlying Prolog atom
- **INTEGER** (or **INT**) . Integer number
- **REAL**. Real number

Examples:

```
CREATE TABLE t(a INT PRIMARY KEY, b STRING)

CREATE OR REPLACE TABLE s(a INT, b INT REFERENCES t(a), PRIMARY
KEY (a,b))
```

Note that if, in this last example, the column name in the referential integrity constraint is missing, an error is thrown:

```
DES-SQL> CREATE OR REPLACE TABLE s(a INT, b INT REFERENCES t,
PRIMARY KEY (a,b))

Error: Type mismatch s.b:number(int) <> t.b:string(varchar).
Error: Imposing constraints.
```

A declared primary key or foreign key constraint is checked whenever a new tuple is added to a table, following relational database philosophy. Note that assertion of rules from the Datalog side are allowed but not checked. A Datalog rule should be viewed as a component of the intensional database. RDBs avoid to define a view with the same name as a table and, therefore, there is no way of unexpected behaviours such as the illustrated below:

```
DES-SQL> create or replace table t(a int,b int,c int,d int,
primary key (a,c))

DES-SQL> insert into t values(1,2,3,4)
Info: 1 tuple inserted.

DES-SQL> % The following is expected

DES-SQL> insert into t values(1,1,3,4)
Error: Primary key violation when trying to insert: t(1,1,3,4)
Info: 0 tuples inserted.

DES-SQL> % The following is allowed

DES-SQL> /assert t(X,Y,Z,U) :- X=1,Y=2,Z=3,U=4.

DES-SQL> /listing
t(1,2,3,4).
t(X,Y,Z,U) :-
  X = 1,
  Y = 2,
  Z = 3,
  U = 4.

DES-SQL> /simplification
Info: Program simplification is off.

DES-SQL> /simplification on
Info: Program simplification is on.

DES-SQL> % The very same rule is now rejected because it is
simplified, becoming a fact (i.e., a tuple which will be tried
to be asserted)
DES-SQL> /assert t(X,Y,Z,U) :- X=1,Y=2,Z=3,U=4.
Error: Primary key violation when trying to insert: t(1,2,3,4)
```

Production rules (those defining the intensional database) are not checked for primary key and foreign key constraints.

Next, a very simple example is reproduced to illustrate basic constraint handling:

```
DES-SQL> create or replace table u(b int primary key,c int)

DES-SQL> create or replace table s(a int,b int, primary key
(a,b))

DES-SQL> create or replace table t(a int,b int,c int,d int,
primary key (a,c), foreign key (b,d) references s(a,b), foreign
key(b) references u(b))

DES-SQL> insert into t values(1,2,3,4)
```

```
Error: Foreign key violation t.[b,d]->s.[a,b] when trying to
insert: t(1,2,3,4)
Info: 0 tuples inserted.

DES-SQL> insert into s values(2,4)
Info: 1 tuple inserted.

DES-SQL> insert into t values(1,2,3,4)
Error: Foreign key violation t.[b]->u.[b] when trying to insert:
t(1,2,3,4)
Info: 0 tuples inserted.

DES-SQL> insert into u values(2,2)
Info: 1 tuple inserted.

DES-SQL> insert into t values(1,2,3,4)
Info: 1 tuple inserted.

DES-SQL> /listing
s(2,4).
t(1,2,3,4).
u(2,2).
```

4.2.4.2 Creating Views

`CREATE [OR REPLACE] VIEW` ***`ViewName(Column1, ..., ColumnN)`***
`AS` ***`SQLStatement`***

This statement defines the view schema in a similar way as defining tables. If the optional clause **`OR REPLACE`** is used, the view is dropped if existed already. Other tuples or rules asserted (with the command **`/assert`**) are not deleted. The view is created with the SQL statement ***`SQLStatement`*** as its definition.

Note that column names are mandatory.

Examples:

```
DES-SQL> CREATE VIEW v(a,b,c,d) AS SELECT * FROM t WHERE a>1

DES-SQL> CREATE OR REPLACE VIEW w(a,b) AS SELECT t.a,s.b FROM
t,s WHERE t.a>s.a

DES-SQL> /dbschema
Info: Table(s):
 s(a:number(int),b:number(int))
   PK: [a,b]
 t(a:number(int),b:number(int),c:number(int),d:number(int))
   PK: [a,c]
Info: View(s):
 v(a:number(int),b:number(int),c:number(int),d:number(int))
  Defining SQL Statement:
    SELECT * FROM t WHERE a > 1;
  Datalog equivalent rules:
    v(A,B,C,D) :- t(A,B,C,D),A > 1.
 w(a:number(int),b:number(int))
```

```
  Defining SQL Statement:
    SELECT t.a, s.b FROM t, s WHERE t.a > s.a;
  Datalog equivalent rules:
    w(A,B) :- t(A,C,D,E),s(F,B),A > F.
```

Note that primary key constraints follow the table schema, and inferred types are in the view schema.

4.2.4.3 Dropping Tables

DROP TABLE *TableName*

This statement drops the table schema with name ***TableName***, deleting all of its tuples (whether they were inserted with INSERT or with the command **/assert**) and rules (which might have been added via **/assert**).

Example:

```
DROP TABLE t;
```

4.2.4.4 Dropping Views

DROP VIEW *ViewName*

This statement drops the view with name ***ViewName***, deleting all of its tuples (whether they were inserted with INSERT or with the command **/assert**) and rules (which might have been added via **/assert**). Other tuples or rules asserted (with the command **/assert**) are not deleted.

Example:

```
DROP VIEW v;
```

4.2.4.5 Dropping Databases

DROP DATABASE

This statement drops the current database, dropping all tables, views, and rules (this includes Datalog rules that may have been asserted or consulted).

Example:

```
DROP DATABASE;
```

Note that if Datalog rules included in the translation of views has been removed, then the statement **DROP DATABASE** will raise a warning indicating that the some rules cannot be retracted, as in:

```
DES-Datalog> /abolish
DES-Datalog> /sql drop database
Info: This will drop all views, tables, and Datalog rules.
      Do you want to proceed? (y/n) y
Warning: Cannot retract.
```

4.2.5 Data Manipulation Language

This part of the language deals with inserting and deleting tuples from tables. There is no provision for updating tuples.

4.2.5.1 Inserting Tuples

INSERT INTO *TableName* VALUES (*Cte1*,...,*CteN*)

This statement inserts into the table ***TableName*** a tuple built with the values ***Cte1***, ..., ***CteN***. A value for each column in the table has to be provided (here, ***N*** is the number of columns of ***TableName***).

Example:

```
INSERT INTO t VALUES (1,1)
```

Another form of the **INSERT** statement allows to inserting tuples which are the result set from a **SELECT** statement:

INSERT INTO *TableName* *SQLStatement*

This statement inserts into the table ***TableName*** as many tuples as returned by the SQL statement ***SQLStatement***. This statement has to return as many columns as the columns of ***TableName***.

Examples:

```
INSERT INTO t SELECT * FROM s
```

You can also insert tuples coming directly (or indirectly) from a table, as in:

```
INSERT INTO t SELECT * FROM t
```

For testing the new (duplicated) contents of **t**, you have to use **/listing t**, instead of a **SELECT**, since this statement always returns a set (no duplicates).

4.2.5.2 Deleting Tuples

DELETE FROM *TableName*

This statement deletes from the table ***TableName*** all of its tuples. It does not delete rules asserted via **/assert**.

Example:

```
DELETE FROM t
```

Another form of the **DELETE** statement allows to deleting tuples which fulfil a given condition:

DELETE FROM *TableName* WHERE *Condition*

This statement deletes from the table ***TableName*** all of its tuples matching the condition ***Condition***. It does not delete rules asserted via **/assert**.

Example:

```
DELETE FROM t WHERE a NOT IN (SELECT a FROM s)
```

4.2.6 Data Query Language

There are three main types of SQL query statements: **SELECT** statements, set statements (**UNION**, **INTERSECT**, and **EXCEPT**), and **WITH** statements (for building recursive queries).

4.2.6.1 Basic SQL Queries

The syntax of the basic SQL query statement is:

```
SELECT ProjectionList
FROM Relations
WHERE Condition
```

Where:

- ***ProjectionList*** is a list of comma-separated columns or arithmetic expressions that will be returned as a tuple result. Wildcards are allowed, as *****, and ***Relation*.***. ***Relation*** can be the name of a table or an alias (for a table or subquery).
- ***Condition*** is a logical condition built from comparison operators (**=**, **<>**, **<**, **>**, **>=**, and **<=**), Boolean operators (**AND**, **OR**, and **NOT**), Boolean constants (**TRUE**, **FALSE**), the existence operator (**EXISTS**) and the inclusion operator (**IN**). See the grammar description in Section 4.2.7 for details. The **WHERE** clause is not required. Subqueries are allowed without limitations.
- ***Relations*** is a list of comma-separated relation definitions. A relation can be either a table name, or a view name, or a subquery, or a join relation. They can be renamed via aliases.

Examples:

```
SELECT t.*, s.b
FROM t,s,v
WHERE t.a=s.a AND v.b=t.b

SELECT t.a, s.b, t.a+s.b
FROM t,s
WHERE t.a=s.a

SELECT *
FROM (SELECT * from t) as r1,
     (SELECT * from s) as r2
WHERE r1.a=r2.b;

SELECT *
FROM s
WHERE s.a NOT IN SELECT a FROM t;

SELECT *
FROM s
WHERE EXISTS
```

```
  SELECT a
  FROM t
  WHERE t.a=s.a;
```

```
SELECT *
FROM s
WHERE s.a > (SELECT a FROM t);
```

```
SELECT 1, a1+a2, a+1 AS a1, a+2 AS a2
FROM t;
```

Notes:

- SQL arithmetic expressions follow the same syntax as Datalog.
- A SQL arithmetic expression can be renamed and used in other expressions.
- The ordering of the expressions is irrelevant
- Circular uses will yield exceptions, as in **`a+a3 AS a3`**

A join relation is either of the form:

Relation **NATURAL** ***JoinOp Relation***

or:

Relation JoinOp Relation **[*JoinCondition*]**

Where ***Relation*** is as before (without any limitation), JoinOP is any join operator (including **INNER JOIN**, **LEFT OUTER JOIN**, **RIGHT OUTER JOIN**, and **FULL OUTER JOIN**), and ***JoinCondition*** can be either:

ON ***Condition***

or:

USING (*Column1*,...,*ColumnN*)

Where ***Condition*** is as described in a **WHERE** clause, and ***Column1***, ..., ***ColumnN*** are common column names of the joined relations.

Examples:

```
SELECT *
FROM t INNER JOIN s ON t.a=s.a AND t.b=s.b;
```

```
SELECT *
FROM t NATURAL INNER JOIN s;
```

```
SELECT *
FROM t INNER JOIN s USING (a,b);
```

```
SELECT *
FROM t INNER JOIN s USING (a);
```

```
SELECT *
FROM t INNER JOIN s USING (b);
```

```
SELECT *
FROM (t INNER JOIN s ON t.a=s.a) AS s, v
WHERE s.a=v.a;

SELECT *
FROM (t LEFT JOIN s ON t.a=s.a) RIGHT JOIN r ON t.a=r.a;

SELECT *
FROM t FULL JOIN s ON t.a=s.a;
```

Notes:

The keywords **ALL** and **DISTINCT** can be used following the **SELECT** keyword, but this does not take any effect in the current system version.

4.2.6.2 Set SQL Queries

The three set operators defined in the standard are available: **UNION**, **EXCEPT**, and **INTERSECT**. The syntax of a set SQL query is:

SQLStatement
SetOperator
SQLStatement

Where ***SQLStatement*** is any SQL statement described in the data query part (without any limitation). ***SetOperator*** is any of the abovementioned set operators.

Examples:

```
(SELECT * FROM s) UNION     (SELECT * FROM t);

(SELECT * FROM s) INTERSECT (SELECT * FROM t);

(SELECT * FROM s) EXCEPT    (SELECT * FROM t);
```

Note that parentheses are not mandatory in these cases and are only use for readability.

4.2.6.3 WITH SQL Queries

The WITH clause, as introduced in the SQL:1999 standard and available in several RDBMS as DB2, Oracle and SQL Server, is intended in particular to define recursive queries. Its syntax is:

```
WITH LocalViewDefinition1,
     ...,
     LocalViewDefinitionN
SQLStatement
```

Where ***SQLStatement*** is any SQL statement, and

LocalViewDefinition1, ..., ***LocalViewDefinition1*** are (local) view definitions that can only be used inside ***SQLStatement***. These local views are not stored in the database and are rather computed when executing ***SQLStatement***. Although they are local, they have to have different names from existing objects (tables or views). The syntax of a local view definition is as follows:

[RECURSIVE] ***ViewName*(*Column1*, ..., *ColumnN*)** **AS** ***SQLStatement***

Here, the keyword **RECURSIVE** for defining recursive views is not mandatory (the parser simply ignores it).

Examples[6]:

```
CREATE TABLE flights(airline,frm,to,departs,arrives);

WITH
  RECURSIVE reaches(frm,to) AS
    (SELECT frm,to FROM flights)
    UNION
    (SELECT r1.frm,r2.to
     FROM reaches AS r1, reaches AS r2
     WHERE r1.to=r2.frm)
SELECT * FROM reaches;

WITH
  Triples(airline,frm,to) AS
    SELECT airline,frm,to
    FROM flights,
  RECURSIVE Reaches(airline,frm,to) AS
    (SELECT * FROM Triples)
    UNION
    (SELECT Triples.airline,Triples.frm,Reaches.to
     FROM Triples,Reaches
     WHERE Triples.to = Reaches.frm AND
           Triples.airline=Reaches.airline)
(SELECT frm,to FROM Reaches WHERE airline = 'UA')
 EXCEPT
(SELECT frm,to FROM Reaches WHERE airline = 'AA');
```

Here, the keyword **RECURSIVE** for defining recursive views is not mandatory (the parser simply ignores it).

In addition, shorter definitions for recursive views are allowed in DES. The next view delivers the same result set as the first example above:

```
CREATE VIEW reaches(frm,to) AS
    (SELECT frm,to FROM flights)
    UNION
    (SELECT r1.frm,r2.to
     FROM reaches AS r1, reaches AS r2
     WHERE r1.to=r2.frm);
```

4.2.7 SQL Grammar

Here, terminal symbols are: Parentheses, commas, semicolons, single dots, asterisks, and apostrophes. Other terminal symbols are completely written in capitals,

---

[6] Adapted from [GUW02].

as **SELECT**. Percentage symbols (%) start comments. User identifiers must start with a letter and consist of letters and numbers; otherwise, a user identifier can be enclosed between quotation marks (both square brackets and double quotes are supported) and contain any characters. Next, **SQLstmt** stands for a valid SQL statement.

```
SQLstmt ::=
  DDLstmt[;]
  |
  DMLstmt[;]
  |
  DQLstmt[;]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DDL (Data Definition Language) statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DDLstmt ::=
  CREATE [OR REPLACE] TABLE CompleteConstrainedSchema
  |
  CREATE [OR REPLACE] VIEW CompleteSchema AS DQLstmt
  |
  DROP TABLE TableName
  |
  DROP VIEW ViewName
  |
  DROP DATABASE

Schema ::=
  RelationName
  |
  CompleteSchema

CompleteConstrainedSchema ::=
  RelationName(Att Type [ColumnConstraint ...
ColumnConstraint],...,Att Type [ColumnConstraint ...
ColumnConstraint] [, TableConstraints])

CompleteSchema ::=
  RelationName(Att Type,...,Att Type)

Type ::=
  CHAR(n)    % fixed-length string of n characters
  |
  CHAR    % fixed-length string of 1 character
  |
  VARCHAR(n)    % variable-length string of up to n characters
  |
  VARCHAR2(n)    % Oracle's variable-length string of up to n characters
  |
  VARCHAR    % variable-length string of up to the maximum length of the underlying
Prolog atom
  |
  STRING    % As VARCHAR
```

```
  |
  INT
  |
  INTEGER    % equivalent to the former
  |
  REAL

ColumnConstraint ::=
  PRIMARY KEY
  |
  REFERENCES TableName[(Att)]

TableConstraints ::=
  TableConstraint,...,TableConstraint

TableConstraint ::=
  PRIMARY KEY (Att,...,Att)
  |
  FOREIGN KEY (Att,...,Att) REFERENCES TableName[(Att,...,Att)]
```

RelationName is a user identifier for naming tables, views and aliases
TableName is a user identifier for naming tables
ViewName is a user identifier for naming views
Att is a user identifier for naming relation attributes

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DML (Data Manipulation Language) statements
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DMLstmt ::=
  INSERT INTO TableName VALUES (Cte,...,Cte)
  |
  INSERT INTO TableName DQLstmt
  |
  DELETE FROM TableName
  |
  DELETE FROM TableName WHERE Condition
  |
  UPDATE TableName SET Att1=Expr1,...,Attn=Exprn WHERE Condition
```

Cte is a constant

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DQL (Data Query Language) statements:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DQLstmt ::=
  (DQLstmt)
  |
  UBSQL

UBSQL ::=
  SELECTstmt
  |
```

```
  DQLstmt UNION DQLstmt
  |
  DQLstmt EXCEPT DQLstmt
  |
  DQLstmt MINUS DQLstmt
  |
  DQLstmt INTERSECT DQLstmt
  |
  WITH LocalViewDefinition,...,LocalViewDefinition DQLstmt

LocalViewDefinition ::=
  [RECURSIVE] CompleteSchema AS DQLstmt

SELECTstmt ::=
  SELECT [[ALL|DISTINCT]] SelectExpressionList
  FROM Rels
  [WHERE WhereCondition]
  [GROUP BY Atts]
  [HAVING HavingCondition]
  [ORDER BY OrderDescription]

Atts ::=
  Att,...,Att

OrderDescription ::=
  Att [[ASC|DESC]],...,Att [[ASC|DESC]]

SelectExpressionList ::=
  *
  |
  SelectExpression,...,SelectExpression

SelectExpression ::=
  UnrenamedSelectExpression
  |
  RenamedExpression

UnrenamedSelectExpression ::=
  Att
  |
  RelationName.Att
  |
  RelationName.*
  |
  ArithmeticExpression
  |
  DQLstmt

RenamedExpression ::=
  UnrenamedExpression [AS] Identifier

ArithmeticExpression ::=
  Op1 ArithmeticExpression
  |
```

```
  ArithmeticExpression Op2 ArithmeticExpression
  |
  ArithmeticFunction(ArithmeticExpression,...,
                     ArithmeticExpression)
  |
  Number
  |
  Att
  |
  RelationName.Att
  |
  ArithmeticConstant

Op1 ::=
  - | \

Op2 ::=
  ^ | ** | * | / | // | rem | \/ | # | + | - | /\ | << | >>

ArithmeticFunction ::=
    sqrt/1 | ln/1 | log/1 | log/2 | sin/1 | cos/1 | tan/1 |
cot/1
  | asin/1 | acos/1 | atan/1 | acot/1 | abs/1 | float/1
  | integer/1 | sign/1 | gcd/2 | min/2 | max/2 | truncate/1
  | float_integer_part/1 | float_fractional_part/1
  | round/1 | floor/1 | ceiling/1

Aggregate Functions:
  avg/1 | count/1 | count/0 | max/1 | min/1 | sum/1

ArithmeticConstant ::=
  pi | e

Rels ::=
  Rel,...,Rel

Rel ::=
  UnrenamedRel
  |
  RenamedRel

UnrenamedRel ::=
  TableName
  |
  ViewName
  |
  DQLstmt
  |
  JoinRel

RenamedRel ::=
  UnrenamedRel [AS] Identifier

JoinRel ::=
```

```
  Rel [NATURAL] JoinOp Rel [JoinCondition]

JoinOp ::=
  INNER JOIN
  |
  LEFT [OUTER] JOIN
  |
  RIGHT [OUTER] JOIN
  |
  FULL [OUTER] JOIN

JoinCondition ::=
  ON WhereCondition
  |
  USING (Atts)

WhereCondition ::=
  BWhereCondition
  |
  UBWhereCondition

HavingCondition
  As WhereCondition, but including aggregate functions

BWhereCondition ::=
  (WhereCondition)

UBWhereCondition ::=
  TRUE
  |
  FALSE
  |
  EXISTS DQLstmt
  |
  NOT (WhereCondition)
  |
  (AttOrCte,...,AttOrCte) [NOT] IN DQLstmt
  |
  WhereExpression IS [NOT] NULL
  |
  WhereExpression [NOT] IN DQLstmt
  |
  WhereExpression Operator [[ALL|ANY]] WhereExpression
  |
  WhereCondition [AND|OR] WhereCondition

WhereExpression ::=
  Att
  |
  Cte
  |
  ArithmeticExpression
  |
  DQLstmt
```

```
AggrArithmeticExpression ::=
  [AVG|MIN|MAX|SUM](Att)
  |
  COUNT([*|Att])

AttOrCte ::=
  Att
  |
  Cte

Operator ::=
  = | <> | < | > | >= | <=

Cte ::=
  Number
  |
  'String'
  |
  NULL
```

`Number` is an integer or floating-point number

4.3 Prolog

Syntax of Prolog programs and goals is the same of that for Datalog, including all built-in operators (cf. next Section) but aggregates.

4.4 Built-ins

Most built-ins are shared by the three languages. For instance, w.r.t. comparison operators, the only difference is the less or equal (**=<**) operator used in Datalog and Prolog. This operator is different from the used in SQL, which is written as **<=**. The former is written that way since in Prolog and Datalog, it is distinguished from the implication to the left operator (**<=**). SQL does not provide implications; so, the SQL syntax seems to be more appealing since the order of the two symbols matches the order of words.

Arithmetic expressions are constructed with the same built-ins in the three languages. However, in Datalog and Prolog, you need to use the infix **`is`** (cf. Section 4.4.2).

The built-in predicates **`is_null/1`** and **`is_not_null/1`** belong to the Datalog language.

4.4.1 Comparison Operators

All comparison operators are infix and applies to terms. For the inequality and disequality operators (greater than, less than, etc.), numbers are compared in terms of their arithmetical value; other terms are compared in Prolog standard order.

If a compound term is involved in a comparison operator, it is evaluated as an arithmetic expression and its result is then compared (for all operators by equality) or unified (for equality).

All comparison operators, but equality, demand ground arguments since they are not constraints, but test operators, and argument domains are infinite. If a ground argument is demanded and a variable is received, an exception is raised.

Next, we list the available comparison operators, where **`X`** and **`Y`** are terms (variables, constants or arithmetic expressions).

- **`X = Y`** (Syntactic equality)
  Tests syntactic equality between **`X`** and **`Y`**. It also performs unification when variables are involved. This is the only comparison operator that does not demand ground arguments.
- **`X \= Y`** (Syntactic disequality)
  Tests syntactic disequality between **`X`** and **`Y`**.
- **`X > Y`** (Greater than)
  Tests whether **`X`** is greater than **`Y`**.
- **`X >= Y`** (Greater than or equal to)
  Tests whether **`X`** is greater than or equal to than **`Y`**.
- **`X < Y`** (Less than)
  Tests whether **`X`** is less than **`Y`**.
- **`X =< Y`** (Less than or equal to)
  Tests whether **`X`** is less than or equal to **`Y`**.

4.4.2 Datalog and Prolog Arithmetic

Borrowed from most Prolog implementations, arithmetic is allowed by using the infix operator **`is`**, which is used to construct a query with two arguments, as follows:

`X` **`is`** ***`Expression`***

where ***`X`*** is a variable or a number, and ***`Expression`*** is an arithmetic expression built from numbers, variables, built-in arithmetic operators, constants and functions, mainly following ISO for Prolog (they are labelled, if so, in the listings below). Availability of arithmetic built-ins mainly depend on the underlying Prolog system (binary distributions cope with all the listed built-ins).

At evaluation time, the expression must be ground (i.e., its variables must be bound to numbers or constants); otherwise, problems as stated in the previous section may arise. Evaluating the above query amounts to evaluate the arithmetic expression according to the usual arithmetic rules, which yields a number (integer or float), and ***`X`*** is bound to this number if it is a variable or tested its equivalence if it is a number. Precision depends on the underlying Prolog system.

Arithmetic built-ins have meaning only in the second argument of **`is`**; they cannot be used elsewhere. For example:

```
DES-Datalog> X is sqrt(2)

{
  1.4142135623730951 is sqrt(2)
}
```

Here, **`sqrt(2)`** is an arithmetic expression that uses the built-in function **`sqrt`** (square root). But:

```
DES-Datalog> sqrt(2) is sqrt(2)
```

raises an input error because an arithmetic expression can only occur as the right argument of **is**. Another example is:

```
DES-Datalog> X is e

{
  2.718281828459045 is exp(1)
}

DES-Datalog> e is e

{
}
```

This means that the built-in arithmetic constant **e** cannot be used outside of an arithmetic expression, and it is otherwise understood as a user defined relation. Here, an input error is not raised since **e** could be a user defined relation. In fact, this should raise a type error, but they are not currently controlled.

In addition, note that arithmetic expressions are compound terms which are translated into an internal equivalent representation. The last example shows this since the constant **e** is translated to **exp(1)**.

Concluding, the infix (infinite) relation **is** is understood as the set of pairs **<V, E>** such that **V** is the equivalent value to the evaluation of the arithmetical expression **E**. Note that, since this relation is infinite, we may reach non-termination: Let's consider the following program (**loop.dl** in the distribution directory) with the query **loop(X)**:

```
loop(0).
loop(X) :-
  loop(Y),
  X is Y + 1.
```

Evaluating that query results in a non-terminating cycle because unlimited tuples **is(*N*,*N*+1)** become computed. To show it, try the query, press Ctrl-C, and type **listing(et)** at the Prolog prompt (only when DES has been started from a Prolog interpreter).

4.4.3 SQL Arithmetic

Arithmetic expressions are constructed with the arithmetic operators listed in the next section. They are used in projection lists and conditions.

4.4.4 Arithmetic Built-ins

This section contains the listings for the supported arithmetic operators, constants, and functions.

4.4.4.1 Arithmetic Operators

The following operators are the only ones allowed in arithmetic expressions, where **`X`** and **`Y`** stand also for arithmetic expressions.

- **`\X`** (Bitwise negation) *ISO*
Bitwise negation of the integer **`X`**.
- **`-X`** (Negative value) *ISO*
Negative value of its single argument **`X`**.
- **`X ** Y`** (Power) *ISO*
`X` raised to the power of **`Y`**.
- **`X ^ Y`** (Power)
Synonym for **`X ** Y`**.
- **`X * Y`** (Multiplication) *ISO*
`X` multiplied by **`Y`**.
- **`X / Y`** (Real division) *ISO*
Float quotient of **`X`** and **`Y`**.
- **`X + Y`** (Addition) *ISO*
Sum of **`X`** and **`Y`**.
- **`X - Y`** (Subtraction) *ISO*
Difference of **`X`** and **`Y`**.
- **`X // Y`** (Integer quotient) *ISO*
Integer quotient of **`X`** and **`Y`**. The result is always truncated towards zero.
- **`X rem Y`** (Integer remainder) *ISO*
The value is the integer remainder after dividing **`X`** by **`Y`**, i.e., **`integer(X)-integer(Y)*(X//Y)`**. The sign of a nonzero remainder will thus be the same as that of the dividend.
- **`X \/ Y`** (Bitwise disjunction) *ISO*
Bitwise disjunction of the integers **`X`** and **`Y`**.
- **`X /\ Y`** (Bitwise conjunction) *ISO*
Bitwise disjunction of the integers **`X`** and **`Y`**.
- **`X # Y`** (Bitwise exclusive or)
Bitwise exclusive or of the integers **`X`** and **`Y`**.
- **`X << Y`** (Shift left) *ISO*
`X` shifted left **`Y`** places.
- **`X >> Y`** (Shift right) *ISO*
`X` shifted right **`Y`** places.

4.4.4.2 Arithmetic Constants

- **`pi`** (π)
Archimedes' constant.
- **`e`** (Neperian number)
Neperian number.

4.4.4.3 Arithmetic Functions

- **`sqrt(X)`** (Square root) *ISO*
Square root of **`X`**.
- **`log(X)`** (Natural logarithm) *ISO*
Logarithm of **`X`** in the base of the Neperian number (**`e`**).
- **`ln(X)`** (Natural logarithm)
Synonym for **`log(X)`**.

- **`log(X,Y)`** (Logarithm)
Logarithm of **`Y`** in the base of **`X`**.
- **`sin(X)`** (Sine) *ISO*
Sine of **`X`**.
- **`cos(X)`** (Cosine) *ISO*
Cosine of **`X`**.
- **`tan(X)`** (Tangent) *ISO*
Tangent of **`X`**.
- **`cot(X)`** (Cotangent)
Cotangent of **`X`**.
- **`asin(X)`** (Arc sine)
Arc sine of **`X`**.
- **`acos(X)`** (Arc cosine)
Arc cosine of **`X`**.
- **`atan(X)`** (Arc tangent) *ISO*
Arc tangent of **`X`**.
- **`acot(X)`** (Arc cotangent)
Arc cotangent of **`X`**.
- **`abs(X)`** (Absolute value) *ISO*
Absolute value of **`X`**.
- **`float(X)`** (Float value) *ISO*
Float equivalent of **`X`**, if **`X`** is an integer; otherwise, **`X`** itself.
- **`integer(X)`** (Integer value)
Closest integer between **`X`** and 0, if **`X`** is a float; otherwise, **`X`** itself.
- **`sign(X)`** (Sign) *ISO*
Sign of **`X`**, i.e., -1, if **`X`** is negative, 0, if **`X`** is zero, and 1, if **`X`** is positive, coerced into the same type as **`X`** (i.e., the result is an integer, iff **`X`** is an integer).
- **`gcd(X,Y)`** (Greatest common divisor)
Greatest common divisor of the two integers **`X`** and **`Y`**.
- **`min(X,Y)`** (Minimum)
Least value of **`X`** and **`Y`**.
- **`max(X,Y)`** (Maximum)
Greatest value of **`X`** and **`Y`**.
- **`truncate(X)`** (Truncate) *ISO*
Closest integer between **`X`** and 0.
- **`float_integer_part(X)`** (Integer part as a float) *ISO*
The same as **`float(integer(X))`**.
- **`float_fractional_part(X)`** (Fractional part as a float) *ISO*
Fractional part of **`X`**, i.e., **`X - float_integer_part(X)`**.
- **`round(X)`** (Closest integer) *ISO*
Closest integer to **`X`**. **`X`** has to be a float. If **`X`** is exactly half-way between two integers, it is rounded up (i.e., the value is the least integer greater than **`X`**).
- **`floor(X)`** (Floor) *ISO*
Greatest integer less or equal to **`X`**. **`X`** has to be a float.
- **`ceiling(X)`** (Ceiling) *ISO*
Least integer greater or equal to **`X`**. **`X`** has to be a float.

4.4.5 Negation

- **`not(`*`Relation`*`)`** (Stratified negation)

It stands for the complement of the relation ***`Relation`*** w.r.t. the meaning of the program (i.e., closed world assumption). See Sections 4.1.8 and 5.11.3.

4.4.6 Datalog Outer Joins

- **`lj(`*`LeftRelation,RightRelation,JoinCondition`*`)`** (Left join)

It stands for the left outer join of the relations ***`LeftRelation`*** and relations ***`RightRelation`***, under the condition ***`JoinCondition`*** (expressed as literals, cf. Section 4.1.1), as understood in extended relational algebra (***`LeftRelation`*** ⟕ $_{JoinCondition}$ ***`RightRelation`***).

- **`rj(`*`LeftRelation,RightRelation,JoinCondition`*`)`** (Right join)

It stands for the right outer join of the relations ***`LeftRelation`*** and relations ***`RightRelation`***, under the condition ***`JoinCondition`*** (expressed as literals, cf. Section 4.1.1), as understood in extended relational algebra (***`LeftRelation`*** ⟖ $_{JoinCondition}$ ***`RightRelation`***).

- **`fj(`*`LeftRelation,RightRelation,JoinCondition`*`)`** (Full join)

It stands for the full outer join of the relations ***`LeftRelation`*** and relations ***`RightRelation`***, under the condition ***`JoinCondition`*** (expressed as literals, cf. Section 4.1.1), as understood in extended relational algebra (***`LeftRelation`*** ⟗ $_{JoinCondition}$ ***`RightRelation`***).

4.4.7 Datalog Aggregates

4.4.7.1 Aggregate Functions

Aggregate functions can only occur in the context of a **`group`**_by aggregate predicate (see next section) and apply to the result set for its input relation.

- **`count(`*`Variable`*`)`**

It returns the number of tuples so that the value for **`Variable`** is not null.

- **`count`**

It returns the number of tuples of the result set.

- **`sum(`*`Variable`*`)`**

It returns the sum of possible values for **`Variable`**, ignoring nulls.

- **`times(`*`Variable`*`)`**

It returns the product of possible values for **`Variable`**, ignoring nulls.

- **`avg(`*`Variable`*`)`**

It returns the average of possible values for **`Variable`**, ignoring nulls.

- **`min(`*`Variable`*`)`**

It returns the minimum value for **`Variable`**, ignoring nulls.

- **`max(`*`Variable`*`)`**

It returns the maximum value for **`Variable`**, ignoring nulls.

4.4.7.2 Group_by Predicate

- **`group_by(`*`Query,Variables,GroupConditions`*`)`**

It solves ***`GroupConditions`*** in the context of ***`Query`***, building groups w.r.t. the possible values the variables in the list ***`Variables`***. This list is specified as a Prolog list, i.e., a sequence of comma-separated values enclosed between brackets. If this list is empty, there is only one group: the answer set for ***`Query`***. The goal ***`GroupConditions`*** may contain expressions including aggregate functions.

4.4.7.3 Aggregate Predicates

- **`count(`*`Query,Variable,Result`*`)`**

  It counts in ***`Result`*** the number of tuples in the result set for the query ***`Query`*** so that ***`Variable`*** is a variable of ***`Query`*** (an attribute of the result relation set) and this attribute is not null. It returns 0 if no tuples are found in the result set.
- **`count(`*`Query,Result`*`)`**

  It counts in ***`Result`*** the total number of tuples in the result set for the query ***`Query`***, disregarding whether they contain nulls or not. It returns 0 if no tuples are found in the result set.
- **`sum(`*`Query,Variable,Result`*`)`**

  It sums in ***`Result`*** the numbers in the result set for the query ***`Query`*** and the attribute ***`Variable`***, which should occur in ***`Query`***. Nulls are simply ignored.
- **`times(`*`Query,Variable,Result`*`)`**

  It computes in **`Result`** the product of all the numbers in the result set for the query ***`Query`*** and the attribute ***`Variable`***, which should occur in ***`Query`***. Nulls are simply ignored.
- **`avg(`*`Query,Variable,Result`*`)`**

  It computes in **`Result`** the average of the numbers in the result set for the query ***`Query`*** and the attribute ***`Variable`***, which should occur in ***`Query`***. Nulls are simply ignored.
- **`min(`*`Query,Variable,Result`*`)`**

  It computes in ***`Result`*** the minimum of the numbers in the result set for the query ***`Query`*** and the attribute ***`Variable`***, which should occur in ***`Query`***. Nulls are simply ignored. If there are no such numbers, it returns **`null`**.
- **`max(`*`Query,Variable,Result`*`)`**

  It computes in ***`Result`*** the maximum of the numbers in the result set for the query ***`Query`*** and the attribute ***`Variable`***, which should occur in ***`Query`***. Nulls are simply ignored. If there are no such numbers, it returns **`null`**.

4.4.8 Datalog Null-related Predicates

- **`is_null(`*`Term`*`)`**

  It succeeds if ***`Term`*** is bound to a null value. It raises an exception if ***`Term`*** is a variable.
- **`is_not_null(`*`Term`*`)`**

  It succeeds if ***`Term`*** is not bound to a null value. It raises an exception if ***`Term`*** is a variable.

5. System Description

This section includes descriptions about the connection to relational database systems via ODBC connections, safety and computability issues, source-to-source transformations, the declarative debuggers and tracers, the batch processing, system messages, and lists all the available commands.

5.1 RDBMS connections via ODBC

DES provides support for connections to (relational) database management systems (RDBMSs) in order to provide data sources for relations. This means that a relation defined in a RDBMS as a view or table is allowed as any other relation defined via a predicate in the deductive database. Then, computing a query can involve computations both in the deductive inference engine and in the external RDBMS SQL engine. Such relations become first-class citizens in the deductive database and, therefore, can be queried in Datalog. If the relation is a view, it will be processed by the SQL engine. When an ODBC connection is opened, all SQL statements are redirected to such connection, so DES does not longer process such statements. This means that all the SQL features of the connected RDBMS are available.

Almost any relational database (RDB) can be accessed from DES using an ODBC connection. Relational database management system (RDBMS) manufacturers provide ODBC implementations which run on many operating systems (Microsoft Windows, Linux, Mac OS X, ...) RDBMSs include enterprise RDBMS (as Oracle, MySQL, DB2, ...) and desktop RDBMS (as MS Access and FileMaker).

ODBC drivers are usually bundled with OS platforms, as Windows OSs (ODBC implementation), Linux OS distributions as Ubuntu, Red Hat and Mandriva (UnixODBC implementation), and Mac OSs 10*x* (iODBC implementation).

Since each RDBMS provides an ODBC driver and each OS an ODBC implementation, details on how to configure such connections are out of the scope of this manual. However, to configure such a connection, typically, the ODBC driver is looked for and installed in the OS. Then, following the manufacturer recommendations, it is configured. You can find many web pages with advice on this. Here, we assume that there are ODBC connections already available.

To access a RDB in DES, first open the connection with the following command, where **`test`** is the name of a previously created ODBC connection to a database:

```
DES-SQL> /open_db test
```

You can also provide a username and password (if needed) as in:

```
DES-SQL> /open_db test user(smith) password(mypwd)
```

Note that if you have previously created some database objects (tables, views, ...) in DES without an ODBC connection, the system asks for dropping the current DES database before starting to use a new database[7].

Assuming that the connection links to an empty database, let's start creating some database objects:

(Note that the MySQL ODBC driver may display annoying messages.)

```
DES-SQL> create table t(a varchar(20) primary key)
DES-SQL> create table s(a varchar(20) primary key)
DES-SQL> create view v(a,b) as select * from t,s
DES-SQL> insert into s select * from t
Info: 1 tuple inserted.
DES-SQL> insert into s values(2)
Info: 1 tuple inserted.
```

[7] Further improvements of the system will include to handle multiple database connections, removing the requirement of dropping the DES database.

Next, one can ask for the database schema (metadata) with the command:

```
DES-SQL> /dbschema
Info: Table(s):
 * s(a:varchar)
 * t(a:varchar)
Info: View(s):
 * v(a:varchar,b:varchar)
```

All of these tables and views can be accessed from DES, as if they were local:

```
DES-SQL> select * from s;
answer(a:varchar) ->
{
  answer('1'),
  answer('2')
}
Info: 2 tuples computed.

DES-SQL> select * from t;
answer(a:varchar) ->
{
  answer('1')
}
Info: 1 tuple computed.

DES-SQL> select * from v;
answer(a:varchar,b:varchar) ->
{
  answer('1','1'),
  answer('1','2')
}
Info: 2 tuples computed.

DES-SQL> insert into t values('1')
Exception: error(odbc(23000,1062,[MySQL][ODBC 3.51
Driver][mysqld-5.0.41-community-nt]Duplicate entry '1' for key
1),_G3)
```

In this example, as table **t** has its single column defined as its primary key, trying to insert a duplicate entry results in an exception from the ODBC driver. Integrity constraints are handled by the RDBMS connected, instead of DES (notice that the exception message is different from the one generated by DES).

Moreover, you can submit SQL statements that are not supported by DES but otherwise by the connected RDBMS, as:

```
DES-SQL> alter table t drop primary key;
```

Then, you can insert again and see the result (including duplicates):

```
DES-SQL> insert into t values('1')
Info: 1 tuple inserted.

DES-SQL> select * from v;
answer(a:varchar,b:varchar) ->
```

```
{
  answer('1','1'),
  answer('1','1'),
  answer('1','2'),
  answer('1','2')
}
Info: 4 tuples computed.
```

Also, duplicate removing is also possible by the external RDBMS, while it is not from the DES engine (up to now):

```
DES-SQL> select distinct * from v;
answer(a:varchar,b:varchar) ->
{
  answer('1','1'),
  answer('1','2')
}
Info: 2 tuples computed.
```

Nonetheless, these external objects can be accessed from Datalog as well (please remember to enable duplicates to get the expected result):

```
DES-SQL> /datalog
DES-Datalog> /duplicates on
Info: Duplicates are on.
DES-Datalog> s(X),t(X)
Info: Processing:
  answer(X) :-
    s(X),
    t(X).
{
  answer('1'),
  answer('1')
}
Info: 2 tuples computed.
```

This is equivalent to the following SQL statement:

```
DES-Datalog> /sql select s.a from s,t where s.a=t.a
answer(a:varchar) ->
{
  answer('1'),
  answer('1')
}
Info: 2 tuples computed.
```

However, whilst the former has been processed by the Datalog engine, the latter has been processed by the external RDBMS. So, some complex SQL statements might be more efficiently processed by the external RDBMS.

Duplicates are relevant in a number of situations. For instance, consider the following, where duplicates are initially disabled:

```
DES-Datalog> group_by(v(X,Y),[X,Y],C=count)
Info: Processing:
  answer(X,Y,C) :-
```

```
    group_by(v(X,Y),[X,Y],C = count).
{
  answer('1','1',1),
  answer('1','2',1)
}
Info: 2 tuples computed.
```

Although there are a couple of tuples for each group (see the table contents above), only one is returned in the count because they are indistinguishable in a set. Now, if duplicates are allowed, we get the expected result:

```
DES-Datalog> /duplicates on
Info: Duplicates are on.

DES-Datalog> group_by(v(X,Y),[X,Y],C=count)
Info: Processing:
  answer(X,Y,C) :-
    group_by(v(X,Y),[X,Y],C = count).
{
  answer('1','1',2),
  answer('1','2',2)
}
Info: 2 tuples computed.
```

Note that, even when you can access SQL objects from Datalog, the contrary is not allowed because there is no metadata information regarding Datalog objects for the SQL engine, which needs it.

To find out the current opened ODBC database, use the command:

```
DES-SQL> /current_db
```

Finally, closing the connection is simply done with:

```
DES-SQL> /close_db
```

5.1.1 Important Caveats

Data in relational tables are cached in the memo table during Datalog computations, and it is not requested anymore until this cache is cleared (either explicitly with the command **/clear_et** or because a command or statement invalidating its contents, as a SQL update query). Therefore, it could be possible to access outdated data from a Datalog query. Let's consider:

```
DES-SQL> /datalog t(X)
{
  t('1')
}
Info: 1 tuple computed.
```

Then, from the MySQL client:

```
mysql> insert into t values('2');
Query OK, 1 row affected (0.06 sec)
```

And, after, in DES, the new tuple is not listed via a Datalog query:

```
DES-SQL> /datalog t(X)
{
  t('1')
}
Info: 1 tuple computed.
```

However, a SQL statement will return the correct answer:

```
DES-SQL> select * from t;
answer(a:varchar) ->
{
  answer('1'),
  answer('2')
}
Info: 2 tuples computed.
```

In addition, it is not recommended to mix Datalog and SQL data. It is possible to assert tuples with the same name and arity as existing RDBMS's tables and/or views. Let's consider the same table t as above with the same data (two tuples **`t('1')`** and **`t('2')`**) and assert a tuple **`t('3')`** as follows:

```
DES-SQL> /assert t('3')

DES-SQL> /datalog t(X)
{
  t('1'),
  t('2'),
  t('3')
}
Info: 3 tuples computed.

DES-SQL> select * from t
answer(a:varchar) ->
{
  answer('1'),
  answer('2')
}
Info: 2 tuples computed.
```

This reveals that, although on the DES side, Datalog data are known, it is not on the RDBMS side. This is in contrast to the DES management of data: if no ODBC connection is opened, the DES engine is aware of any changes to data, both from Datalog and SQL sides.

Concluding, those updates that are external to DES might not be noticed by the DES engine. And, also, an ODBC connection should be seen as a source of external data that should not be mixed with Datalog data. However, you can safely use the more powerful Datalog language to query external data (and to be sure the current data is retrieved, clear the cache).

5.1.2 Platform-specific Issues

ODBC connections are only supported by the provided binaries, and the source distributions for SWI-Prolog and SICStus Prolog. The former Prolog system enjoys a more robust implementation than the latter. In addition, the latter crashes on the tested Windows platform (Windows XP SP2) when some ODBC exceptions are raised (as, e.g., due to syntax errors in SQL statements).

ODBC connections in SICStus source distributions have been implemented and tested for MySQL, Oracle and MS Access. The last one does not provide access to metadata information. So, if you need to use it, you have to copy the table MSysObjects to a new table called db_schema (both data and structure) for each database you need access. Other RDBMSs are not supported.

The stable releases Ciao Prolog 1.10p5 and GNU-Prolog 1.3.1 do not support this implementation.

5.2 Safety and Computability

Built-in predicates are appealing, but they come at a cost, already noticed in Section 4.4. The domain of their arguments is infinite, in contrast to the finite domains of each argument of any user-defined predicate. Since it is neither reasonable nor possible to (extensionally) give an infinite answer, when a subgoal involving a built-in is going to be computed, its arguments need to be range restricted, i.e., the arguments have to take values provided by other subgoals. To illustrate this point, consider submitting the following view to the program file **`relop.dl`**:

```
less(X,Y) :- X < Y, c(X,Y).
```

Since the goal is **`less(X,Y)`**, and the computation is left to right, both **`X`** and **`Y`** are not range restricted when computing the goal **`X < Y`** and, therefore, this goal ranges over two infinite domains: the one for **`X`** and the one for **`Y`**. We do not allow the computation of such rules. However, if we reorder the two goals as follows:

```
less(X,Y) :- c(X,Y), X < Y.
```

we get the expected result:

```
{
  less(a1, b2),
  less(a2, b2)
}
```

Note, then, that built-in predicates affect declarative semantics, i.e., the intended meaning of the two former views should be the same, although actually it is not. Declarative semantics is therefore affected by the underlying operational mechanism. Notice, nonetheless, that Datalog is less sensitive to operational issues than Prolog and it could be said to be more declarative. First, because of terminating issues as already introduced, and second, because the problematic first view can be automatically transformed into the second, computation-safe, one, as we explain next.

We can check whether a rule is safe in the sense that all its variables are range restricted and, then, reorder the goals for allowing its computation. First, we need a notion of safety, which intuitively seems clear but that actually is undecidable [ZCF+97]. Some simple sufficient conditions for the safety of Datalog programs can be imposed, which means that rules obeying these conditions can be safely computed, although there are rules that, even violating some conditions, can be actually computed. We impose the following (weak) conditions [Ullm95, ZCF+97] for safe rules adapted to our context:

1. Any variable *X* in a rule *r* is safe if:
   a. *X* occurs in some positive goal referring to a user-defined predicate
   b. *r* contains some equality goal *X*=*Y*, where *Y* is safe (*Y* can be a constant, which, obviously, makes *X* safe)
   c. A variable *X* in the goal *X is Expression* is safe whenever all variables in *Expression* are safe
2. A rule is safe if all its variables are safe.

Notice that these conditions, currently supported by the system, are weak since they assume that user-defined predicates are safe, which is not always the case (but only require analysing locally each rule for deciding weak safety). To make these conditions stronger, 1.a. has to be changed to: "*X* occurs in some positive goal referring to a *safe* user-defined predicate", and add "3. A predicate is safe if all of its variables are safe". The changed conditions would require a global analysis of the program, which is not supported by DES up to now.

The built-in predicate **`is`** has the same problem as comparison operators as well, but it only demands ground its second argument (cf. condition 3.c above). Negation requires its argument to have no unsafe variables. In addition, to be correctly computed, the restrictions in the domains of the safe variables it may contain should be computed before. The reader is referred to Section 3.6 in [Ullm95] for finding the problems when interpreting rules with negation.

DES provides a program transformation that allows deciding if a rule is safe and, if so, transform it by reordering the goals in order to make it computable. This translation comes by default, and it can be changed with the command **`/safe`** ***`Switch`***, where ***`Switch`*** can take two values: **`on`**, for activating program transformation, and **`off`**, for disallowing this transformation. If ***`Switch`*** is not included, then the command informs whether program transformation is enabled or disabled.

The analysis performed by the system at compile-time warns about safety and computability as follows:

1. Raise an error if:
   a. A goal involving a comparison operator *will* be non-ground at run-time.
   b. The expression **`E`** in a goal **`X is E`** *will* be non-ground at run-time.
   c. The goal **`not(G)`** contains unsafe variables or its safe variables are not restricted so far.
2. Raise a warning if:
   a. A goal involving a comparison operator *may* be non-ground at run-time.
   b. The expression **`E`** in a goal **`X is E`** *may* be non-ground at run-time.

This analysis is performed in several cases:

- Whenever a rule is asserted (either manually with the command **`/assert`** or automatically when consulting programs). A rule is always asserted, even when it is detected as unsafe or it may raise an exception at run-time. Recall that safety is undecidable and there are rules detected as unsafe that can be actually and correctly computed.
- When a query, conjunctive query (autoview) or view is submitted. They are rejected and not computed if unsafety or uncomputability is detected and cannot be repaired (because program transformation is disabled or there is no way). Notice that there can be unsafe or uncomputable rules already consulted than can yield an incorrect result or raise a run-time exception.

Concluding, one can expect a correct answer whenever no unsafe, uncomputable rule has been asserted to an empty database. Recall that the local analysis relies on the weak condition that assumes that the consulted rules are safe.

5.3 Source-to-Source Transformations

Currently, two source-to-source transformations are possible under demand: First, as explained in the previous section, when safety transformations are enabled via the command **`/safe on`**, rule bodies are reordered to try to produce a safe rule. Second, when simplification is enabled via the command **`/simplification on`**, rule bodies containing equalities, **`true`**, and **`not(`*`BooleanValue`*`)`** are simplified.

In addition, there is also place for several automatic transformations:

- A clause containing a disjunctive body are translated into a sets of clauses with conjunctive bodies.
- A clause containing an outer join predicate is transformed into an executable form.
- A clause containing the goal **`not(is_null(+`*`Term`*`))`** is transformed into a clause with this goal replaced by **`is_not_null(+`*`Term`*`)`**.

5.4 Datalog and SQL Tracers

In contrast to imperative programming languages, deductive and relational database query languages feature solving procedures which are far from the query languages itself. Whilst one can trace an imperative program by following each statement as it is executed, along with the program state, this is not feasible in declarative (high abstraction) languages as Datalog and SQL. However, this does not apply to Prolog, also acknowledged as a declarative language, because one can follow the execution of a goal via the sld resolution tree and use the four-port debugging approach.

Datalog stems from logic programming and Prolog in particular, and it can also understood as a subset of Prolog. However, its operational behavior is quite different, since the outcome of a query represents all the possible resolutions, instead of a single one as in Prolog. In addition, tabling (cf. Section 5.3) and program transformations (due to outer joins, aggregates, simplifications, disjunctions, ...) make tracing cumbersome.

Similarly, SQL represents a true declarative language which is even fartest from its computation procedure than Prolog. Indeed, the execution plan for a query include transformations considering data statistics to enhance performance. These query plans are composed of primitive relational operations (such as Cartesian product) and specialized operations for which efficient algorithms have been developed, containing in general references to index usage.

Therefore, instead of following a more imperative approach to tracing, here we focus on a (naïve) declarative approach which only take into account the outcomes at some program points. This way, the user can inspect each point and decide whether its outcome is correct or not. This approach will allow to examine the syntactical graph of a query, which possibly depends on other views or predicates (SQL or Datalog, resp.) This graph may be cyclic when recursive views or predicates are involved. However, a given node in the graph will be traversed only once. In the case of Datalog queries, this graph contains the nodes and edges in the dependency graph restricted to the query, ignoring other nodes which do not take part in its computation. In the case of SQL, the graph shows the dependencies between a view and its data sources (in the **`FROM`** clause).

Next, tracing for both Datalog queries and SQL views are explained and illustrated with examples.

5.4.1 Tracing Datalog Queries

The command **`/trace_datalog`** ***`Goal`*** **`[`*`Order`*`]`** allows to trace a Datalog goal in the given order (**`postorder`** or the default **`preorder`**). Goals should be basic, i.e., no conjunctive or disjunctive goals are allowed. For instance, let's consider the program in the file **`negation.dl`** and its dependency graph, shown in Figure 3. A tracing session could be as follows:

```
DES-Datalog> /c negation
Warning: Undefined predicate(s): [d/0]
DES-Datalog> /trace_datalog a
Info: Tracing predicate 'a'.
{
  a
}
Info: 1 tuple in the answer table.
Info : Remaining predicates: [b/0,c/0,d/0]
Input: Continue? (y/n) [y]:
Info: Tracing predicate 'b'.
{
  not(b)
}
Info: 1 tuple in the answer table.
Info : Remaining predicates: [c/0,d/0]
Input: Continue? (y/n) [y]:
Info: Tracing predicate 'c'.
{
  c
}
Info: 1 tuple in the answer table.
Info : Remaining predicates: [d/0]
Input: Continue? (y/n) [y]:
Info: Tracing predicate 'd'.
```

```
{
}
Info: No more predicates to trace.
```

5.4.2 Tracing SQL Views

Tracing SQL views is similar to tracing Datalog queries, but, instead of posing a goal (involving in general variables and constants) to trace, only the name of a view should be given. For example, let's consider the file **family.sql**, which contains view definitions for **ancestor** and **parent**, where tables **father** and **mother** are involved in the latter view. Note that this view is recursive since it depends on itself:

```
create view parent(parent,child) as
    select * from father
  union
    select * from mother;

create or replace view ancestor(ancestor,descendant) as
    select parent,child from parent
  union
    select parent,descendant
      from parent,ancestor where parent.child=ancestor.ancestor;
```

Then, tracing the view **ancestor** is as follows:

```
DES-SQL> /trace_sql ancestor
Info: Tracing view 'ancestor'.
{
  ancestor(amy,carolIII),
  ...
  ancestor(tony,carolIII)
}
Info: 16 tuples in the answer table.
Info : Remaining views: [parent/2,father/2,mother/2]
Input: Continue? (y/n) [y]:
Info: Tracing view 'parent'.
{
  parent(amy,fred),
  ...
  parent(tony,carolII)
}
Info: 8 tuples in the answer table.
Info : Remaining views: [father/2,mother/2]
Input: Continue? (y/n) [y]:
Info: Tracing view 'father'.
{
  father(fred,carolIII),
  ...
  father(tony,carolII)
}
Info: 4 tuples in the answer table.
Info : Remaining views: [mother/2]
Input: Continue? (y/n) [y]:
Info: Tracing view 'mother'.
```

```
{
  mother(amy,fred),
  ...
  mother(grace,amy)
}
Info: 4 tuples in the answer table.
Info: No more views to trace.
DES-SQL> /trace_datalog father(X,Y)
Info: Tracing predicate 'father'.
{
  father(fred,carolIII),
  ...
  father(tony,carolII)
}
Info: 4 tuples in the answer table.
Info: No more predicates to trace.
```

5.5 Datalog Declarative Debugger

Our approach [CGS07] to debug Datalog programs is anchored to the semantic level instead of the computation level. We have implemented a novel way of applying declarative debugging, also called algorithmic debugging (a term first coined in the logic programming field by E.H. Shapiro [Shap83]) to Datalog programs. With this approach, it is possible to debug queries and diagnose missing answers (an expected tuple is not computed) as well as wrong answers (a given computed tuple should not be computed). Our system uses a question-answering procedure which starts when the user detects an unexpected answer for some query. Then, if possible, it points to the program fragment responsible of the incorrectness.

The debugging process consists of two phases. During the first phase the debugger builds a computation graph (CG) for the initial query Q w.r.t. the program P. This graph represents how the meanings of queries are constructed. See more details in [CGS07]. The second phase consists of traversing the CG to find either a buggy vertex or a set of related incorrect vertices. The vertex associated to the initial query Q is marked automatically as non-valid by the debugger. The rest of the vertices are marked initially as unknown. In order to minimize the number of questions asked by a declarative debugger, several traversing strategies have been studied [Caba05,Silv07]. However, these strategies are only adequate for declarative debuggers based on trees and not on graphs. The currently implemented strategy already contains some ideas of how to minimize the number of questions in a CG:

- First, the debugger asks about the validity of vertices that are not part of cycles in order to find a buggy vertex, if it exists. Only when this is no longer possible, the vertices that are part of cycles are visited.
- Each time the user indicates that a vertex (Query = FactSet) is valid, i.e., the validity of the answer for the subquery Query is ensured, the tool changes to valid all the vertices with queries subsumed by Query.
- Each time the user indicates that a vertex (Query = FactSet) is non-valid, the tool changes to non-valid all the vertices with queries subsumed by Query.

The last two items help to reduce the number of questions, deducing automatically the validity of some vertices from the validity of others.

As an example, we show a debugger session for the query **br_is_even** in the program **parity.dl**, which has been changed to contain an error in the following rule:

```
has_preceding(X) :- br(X), br(Y), Y>X. %error: Y>X should be Y<X
```

In this case, the user expects the answer for the query **br_is_even** to be **{br_is_even}**, because the relation **br** contains two elements: **a** and **b**. However, the answer returned by the system is **{}**, which means that the corresponding query was unsuccessful.

The available command for starting a debugging session is **/debug_datalog *Goal***, where ***Goal*** is a basic goal, i.e., no conjunctive or disjunctive goals are allowed. Therefore, the user can start a typical debugging session as follows:

```
DES-Datalog> /debug_datalog br_is_even
Debugger started ...
Is br(b) = {br(b)} valid(v)/non-valid(n) [v]? v
Is has_preceding(b) = {} valid(v)/non-valid(n) [v]? n
Is br(X) = {br(b),br(a)} valid(v)/non-valid(n) [v]? v
! Error in relation: has_preceding/1
! Witness query: has_preceding(b) = { }
```

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

5.7 Test Case Generator

Checking that a view produces the same result as its intended interpretation is a daunting task when large databases and both dependent and correlated queries are considered. Test case generation provides tuples that can be matched to the intended interpretation of a view and therefore be used to catch posible design errors in the view.

A test case for a view in the context of a database is a set of tuples for the different tables involved in the computation of the view. Executing a view for a *positive* test case (PTC)[8] should return, at least, one tuple. This tuple can be used by the user to catch errors in the view, if any. This way, if the user detects that this tuple should not be part of the answer, it is definitely a witness of the error in the design of the view. On the contrary, the execution of the view for a *negative* test case (NTC) should return at least one tuple which should not be in the result set of the query. Again, if no such a tuple can be found, this tuple is a witness of the error in the design.

A PTC in a basic query means that at least one tuple in the query domain satisfies the **where** condition. In the case of aggregate queries, a PTC will require finding a valid aggregate verifying the **having** condition, which in turn implies that all its rows verify the **where** condition.

In the case of basic query, a NTC will contain at least one tuple in the result set of the view not verifying the **where** condition. In queries containing aggregate

[8] That is, executing the view using as input data for the tables those in the PTC.

functions, this tuple either does not satisfy either the **where** condition or the **having** condition. Set operations are also allowed in both PTC and NTC generation.

It is possible to obtain a test case which is both positive and negative at the same time thus achieving *predicate coverage* with respect to the where and having clauses (in the sense of [AO08]). We will call these tests PNTCs. For instance, consider the following system session:

```
DES-SQL> create table t(a int primary key)
DES-SQL> create view v(a) as select a from t where a=5
DES-SQL> /test_case v
Info: Test case over integers:
[t(5),t(4)]
```

The test case {**t(5)**,**t(4)**} is a PNTC. However, a PNTC is not always possible to be generated. For instance, it is possible for the following view to generate both PTCs and NTCs but no PNTC:

```
create view v(a) as
select a
from t
where a=1 and not exists (select a from t where a<>1);
```

The only one PTC for this view is {**t(1)**} (modulo duplicates). There are many NTCs, as, e.g., {**t(2)**} and {**t(1) ,t(2)**}.

The command **/test_case *View* [*Options*]** allows two kind of options: first, to specify which *class* of test case is to be generated: **all** (PNTC, the default option), **positive** (PTC) or **negative** (NTC). The second option specifies an *action*: the results are to be displayed via the option **display** (default option), added to the corresponding tables (**add** option) or the contents of the tables replaced by the generated test case tuples (**replace** option).

For experimenting with the domain of attributes, we provide the command **/tc_domain *Min Max***, which defines de range of values the integer attributes may take. This range is determinant in the search of test cases in a constraint network that can easily become too complex as long as involved views grow. So, keeping this domain small allows to manage bigger problems.

String constants occurring in all the views on which the view for the test case generated depends are mapped to integers in the same domain, starting from 0. So, the size of the domain has to be larger enough to hold, at least, the string constants in those views.

Also, we provide the command **/tc_size *Min Max*** for specifying the size of the test case generated, in number of tuples. Again, keeping this value small helps in being able to cope with bigger problems.

Currently, we provide support for integer and string attributes. Binary distributions, and both SICStus and SWI Prolog source distributions allow the functionality descrbed. GNU Prolog source distribution only allows non-negative integers in the domain declaration. Ciao Prolog source distribution does not support finited domain constraints, so it neither support test case generation.

5.8 Batch Processing

There are two ways for processing batch files:

1. If the file **des.ini** is located at the distribution directory, its contents are interpreted as input prompts and executed before giving control to the user at start-up of the system.
2. The command **/process *filename*** (or **/p** as a shorthand) allows to process each line in the file as it was an input, the same way as before.

When processing batch files, prompt inputs starting with the symbol % are interpreted as comments. This way, the batch file des.ini may contain comments. The user can also interactively input such comments, but again produce no effects.

Batch processing can include logging to produce output. This is useful to feed the system with batch input and get its output in a file, maybe avoiding any interactive input. For example, consider the following **des.ini** excerpt:

```
% Dump output to output.txt
/log output.txt
/pretty_print off
% Process (Datalog, SQL, ... queries and commands)
/c examples/fib
fib(100,F)
% End log
/nolog
```

The result found in output.txt should be (modulo blank lines):

```
DES-Datalog> /pretty_print off
Info: Pretty print is off.
DES-Datalog> % Process (Datalog, SQL, ... queries and commands)
DES-Datalog> /c examples/fib
Warning: N > 1 may raise a computing exception if non-ground at
run-time.
Warning: N2 is N - 2 may raise a computing exception if non-
ground at run-time.
Warning: N1 is N - 1 may raise a computing exception if non-
ground at run-time.
Warning: Next rule is unsafe because of variable(s):
  [N]
fib(N,F) :- N > 1,N2 is N - 2,fib(N2,F2),N1 is N -
1,fib(N1,F1),F is F2 + F1.
DES-Datalog> fib(100,F)
{
  fib(100,573147844013817084101)
}
Info: 1 tuple computed.
DES-Datalog> % End log
DES-Datalog> /nolog
```

5.9 Messages

DES system messages are prefixed by:

- **`Info`**: An information message which requires no attention from the user. Several information messages are hidden with the command **`/verbose off`**.
- **`Warning`**: A warning message which does not necessarily imply an error, but the user is requested to focus on its origin. These messages are always shown.
- **`Error`**: An error message which requires attention from the user. These messages are always shown.
- **`Exception`**: An exception message which requires attention from the user. These messages are always shown. Examples of exception messages include instantiation errors and undefined predicates.

Prolog exceptions are caught by DES and shown to the user without any further processing. Depending on the Prolog platform, the system may continue by itself; otherwise the user must type **`des`**. (including the ending dot) to continue.

5.10 Commands

The input at the prompt (i.e., commands or queries) must be written in a line (i.e., without carriage returns, although it can be broken by the DES console due to space limitations) and can end with an optional dot.

Commands are issued by preceding the command with a slash (`/`) at the DES system prompt. An argument for a command is not enclosed between brackets, it simply occurs separated by one or more blanks. This cuts short typing.

Ending dots are considered as part of the argument wherever they are expected. For instance, **`/cd ..`** behaves as **`/cd ...`** (this command changes the working directory to the parent directory). In this last case, the final dot is not considered as part of the argument. The command **`/ls .`** shows the contents of the working directory, whereas **`/ls ..`** shows the contents of the parent directory (which behaves as **`/ls ...`**).

Filenames and directories can be specified with relative or absolute names. There is no need of enclosing such names between separators. For instance, file or directory names can contain blanks (for Windows users) and you neither need to use double quotes nor are allowed to use them.

Since commands are submitted with a preceding slash, they are only recognized as commands in this way. Therefore, you can use command names for your relation names without confusion.

When consulting Datalog files, filename resolution works as follows:

- If the given filename ends with **`.dl`**, DES tries to load the file with this (absolute or relative) filename.
- If the given filename does not end with **`.dl`**, DES firstly tries to load a file with **`.dl`** appended to the end of the filename. If such a file is not found, it tries to load the file with the given filename.

In command arguments, when applicable, you can use relative or absolute pathnames. In general, you can use a slash (**/**) as a directory delimiter, but depending on the platform, you can also use the backslash (****).

See Section 4.1.2 for information about DES queries.

5.10.1 Rule Database Commands

- **/[*FileNames*]**
  Load the Datalog programs found in the comma–separated list **[*Filenames*]**, discarding the rules already loaded. The extension table is cleared, and the predicate dependency graph and strata are recomputed. Arguments in the list are comma–separated.
  Examples:
  Assuming we are on the examples distribution directory, we can write:
  DES-Datalog> /[mutrecursion,family]
  See also **/consult *Filename***.
- **/[+*FileNames*]**
  Load the Datalog programs found in the comma–separated list ***Filenames***, keeping rules already loaded. The extension table is cleared, and the predicate dependency graph and strata are recomputed.
  See also **/[*Filenames*]**.
- **/abolish**
  Delete all the loaded rules, including those which are the result of SQL compilations. The extension table is cleared, and the predicate dependency graph and strata are recomputed.
- **/abolish *Name***
  Delete all the loaded rules for the predicates matching ***Name***, including those which are the result of SQL compilations. The extension table is cleared, and the predicate dependency graph and strata are recomputed.
- **/abolish *Name/Arity***
  Delete all the loaded rules for the predicate matching the pattern ***Name/Arity***, including those which are the result of SQL compilations. The extension table is cleared, and the predicate dependency graph and strata are recomputed.
- **/assert *Head*[*:-Body*]**
  Add a Datalog rule. If ***Body*** is not specified, it is simply a fact. Rule order is irrelevant for Datalog computation. The extension table is cleared, and the predicate dependency graph and strata are recomputed.
- **/consult *FileName***
  Load the Datalog program found in the file ***Filename***, discarding the rules already loaded. The extension table is cleared, and the predicate dependency graph and strata are recomputed. The default extension **.dl** for Datalog programs can be omitted.
  Examples:
  Assuming we are on the distribution directory, we can write:
  DES-Datalog> /consult examples/mutrecursion
  which behaves the same as the following:
  DES-Datalog> /consult examples/mutrecursion.dl
  DES-Datalog> /consult ./examples/mutrecursion
  DES-Datalog> /consult c:/des2.0/examples/mutrecursion.dl
  This last command assumes that the distribution directory is **c:/des2.0**.

Synonyms: **/c**, **/restore_ddb**.

- **/listing**

List the loaded rules.

- **/listing *Name***

List the loaded rules matching ***Name***.

- **/listing *Name/Arity***

List the loaded rules matching the pattern ***Name/Arity***.

- **/listing *Head***

List the loaded rules whose heads are subsumed by the head ***Head***.

- **/listing *Head:-Body***

List the loaded rules that are subsumed by ***Head:-Body***.

- **/reconsult *FileName***

Load a Datalog program found in the file ***Filename***, keeping the rules already loaded. The extension table is cleared, and the predicate dependency graph and strata are recomputed.
See also **/consult *Filename***.
Synonyms: **/r**.

- **/restore_ddb Filename**

Restore the Datalog database in the given file (same as **consult**)

- **/retract *Head*[*:-Body*]**

Delete the first Datalog rule that unifies with ***Head:-Body*** (or simply with ***Head***, if ***Body*** is not specified. In this case, only facts are deleted). The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/retractall *Head***

Delete all the Datalog rules whose heads unify with ***Head***. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/save_ddb *Filename***

Save the current Datalog database to the file ***Filename***

5.10.2 Relational Database Commands

- **/open_db *Name* [*Options*]**

Open and set the current ODBC connection to ***Name***, where ***Options***=[**user(*Username*)**] [**password(*Password*)**]. This connection must be already defined at the OS layer.

- **/close_db**

Close the current ODBC connection.

- **/current_db**

Display the current ODBC connection name and DSN provider.

5.10.3 Debugging and Test Case Generation

- **/debug_datalog *Goal* [*Level*]**

Start the debugger for the basic goal ***Goal*** at predicate or clause levels, which is indicated with the options **p** and **c** for ***Level***, respectively. Default is **p**.

- **/trace_datalog *Goal* [*Order*]**

Trace a Datalog goal in the given order (**postorder** or the default **preorder**).

- **/trace_sql *View* [*Order*]**

Trace a SQL view in the given order (**postorder** or the default **preorder**).

- **/test_case *View* [*Options*]**

Generate test case classes for the view ***`View`***. ***`Options`*** may include a class and/or an action parameters. The test case class is indicated by the values **`all`** (positive-negative, the default), **`positive`**, or **`negative`** in the class parameter. The action is indicated by the values **`display`** (only display tuples, the default), **`replace`** (replace contents of the involved tables by the computed test case), or **`add`** (add the computed test case to the contents of the involved tables) in the action parameter.

- **`/tc_size`** ***`Min Max`***

Set the minimum and maximum number of tuples generated for a test case.

5.10.4 Extension Table Commands

- **`/list_et`**

List the contents of the extension table in lexicographical order. First, answers are displayed, then calls.

- **`/list_et`** ***`Name`***

List the contents of the extension table matching ***`Name`***.

- **`/list_et`** ***`Name/Arity`***

List the contents of the extension table matching the pattern ***`Name/Arity`***.

- **`/clear_et`**

Delete the contents of the extension table.

5.10.5 Operating System Commands

- **`/cd`** ***`Path`***

Set the current directory to ***`Path`***.

- **`/cd`**

Set the current directory to the directory where DES was started from.

- **`/pwd`**

Display the absolute filename for the current directory.

- **`/ls`**

Display the contents of the current directory in alphabetical order. First, files are displayed, then directories.

Synonym: **`/dir`**.

- **`/ls`** ***`Path`***

Display the contents of the given directory in alphabetical order. It behaves as **`/ls`**.

Synonym: **`/dir`** ***`Path`***.

- **`/shell`** ***`Command`***

Submit ***`Command`*** to the operating system shell.

Notes for platform specific issues:

- Windows users:
  `command.exe` is the shell for Windows 98, whereas **`cmd.exe`** is the one for Windows NT/2000/2003/XP.
- Ciao users:
  The environment variable **`SHELL`** must be set to the required shell.
- SICStus users:
  Under Windows, if the environment variable **`SHELL`** is defined, it is expected to name a Unix like shell, which will be invoked with the option **`-c Command`**. If SHELL is not defined, the shell named by **`COMSPEC`** will be invoked with the option **`/C Command`**.
- Windows and Linux/Unix executable users:
  The same note for SICStus is applied.

Synonyms: **/s**.

5.10.6 Log Commands

- **/log**

Display the current log file, if any.
- **/log *Filename***

Set the current log to the given filename.
- **/nolog**

Disable logging.

5.10.7 Informative Commands

- **/builtins**

List predefined operators, functions, and predicates.
- **/dbschema**

Display the database schema.
- **/dbschema *Name***

Display the database schema for the given view or table name.
- **/development**

Display whether development listings are enabled.
- **/development *Switch***

Enable or disable development listings (**on** or **off**, resp.). These listings show the source-to-source translations needed to handle null values, Datalog outer join built-ins, and disjunctive literals.
- **/duplicates**

Display whether duplicates are enabled.
- **/help**

Display the help on commands.
Synonyms: **/h**.
- **/negation**

Display the selected algorithm for solving negation (**strata** or **et_not**).
- **/pdg**

Display the current predicate dependency graph.
- **/pretty_print**

Display whether pretty print listings is enabled.
- **/pretty_print *Switch***

Enable or disable pretty print for listings (**on** or **off**, resp.)
- **/safe**

Display whether program transformation is enabled.
- **/simplification**

Display whether program simplification is enabled.
- **/status**

Display the current system status, i.e., verbose mode, the selected negation algorithm, logging, elapsed time display, program transformation, and system version.
- **/strata**

Display the current stratification as a list of pairs (PredName/Arity, Stratum).
- **/timing**

Display whether elapsed time display is enabled.

- **`/timing Switch`**
Disable or enable either a basic or detailed elapsed time display (**`off`**, **`on`**, **`detailed`**, resp.)
- **`/verbose`**
Display whether verbose output is enabled.
- **`/verbose Switch`**
Enable or disable verbose output messages (**`on`** or **`off`**, resp.)
- **`/version`**
Display the current system version.

5.10.8 Languages

- **`/datalog`**
Switch to Datalog interpreter (all queries are parsed and executed in Datalog).
- **`/datalog Query`**
Trigger Datalog resolution for the query Query.
- **`/prolog`**
Switch to Prolog interpreter (all queries are parsed and executed in Prolog).
- **`/prolog Goal`**
Trigger Prolog's SLD resolution for the goal ***`Goal`***.
- **`/sql`**
Switch to SQL interpreter (all queries are parsed and executed in SQL).
- **`/sql SQL_statement`**
Trigger SQL resolution for ***`SQL_statement`***.

5.10.9 Miscellanea

- **`/duplicates Switch`**
Enable or disable duplicates (**`on`** or **`off`**, resp.)
- **`/negation Algorithm`**
Set the required ***`Algorithm`*** for solving negation (**`strata`** or **`et_not`**) .
- **`/halt`**
Quit the system.
Synonyms: **`/quit`**, **`/q`**, **`/exit`**, **`/e`**.
- **`/output Switch`** Enable or disable display output (**`on`** or **`off`**, resp.)
- **`/process Filename`**
Process the contents of ***`Filename`*** as if they were typed at the system prompt. Extensions by default are: **`.sql`** and **`.ini`**. When looking for a file f, the following filenames are checked in this order: **`f`**, **`f.sql`**, and **`f.ini`**.
Synonyms: **`/p`**.
- **`/safe Switch`**
Enable or disable program transformation (**`on`** or **`off`**, resp.)
- **`/simplification Switch`**
Enable or disable program simplification (**`on`** or **`off`**, resp.). Rules with equalities, **`true`**, and **`not(`*****`BooleanValue`*****`)`** are simplified.

5.11 Notes about the Implementation of DES

DES is implemented with the seminar ideas found in [Diet87, TS86], that deal with termination issues of Prolog programs. These ideas have been already used in the deductive database community. Our implementation uses extension tables for achieving a top–down driven bottom–up approach. In its current form, it can be seen as an extension of the work in [Diet87] in the sense that, in addition, we deal with negation, undefined (although incomplete) information, nulls and aggregates, also providing a more efficient tabled mechanism. Also, the implementation follows a different approach: Instead of translating rules, we interpret them.

DES does not pretend to be an efficient system but a system capable of showing the nice aspects of the more powerful form of logic we can find in Datalog systems wrt. relational database systems.

5.11.1 Tabling[9]

DES uses an extension table which stores answers to goals previously computed, as well as their calls. For the ease of the introduction, we assume an answer table and a call table to store answers and calls, respectively. Answers may be positive or negative, that is, if a call to a positive goal **`p`** succeeds, then the fact **`p`** is added as an answer to the answer table; if a negated goal **`not(p)`** succeeds, then the fact **`not(p)`** is added. Calls are also added to the call table whenever they are solved. This allows us to detect whether a call has been previously solved and we can use the results in the extension table (if any). The algorithm which implements this idea can be sketched as follows:

First, test whether there is a previous call that subsumes[10] the current call. There are two possibilities: 1) there is such a previous call: then, use the result in the answer table, if any. It is possible that there is no such a result (for instance, when computing the goal **`p`** in the program **`p :- p`**) and we cannot derive any information, 2) otherwise, process the new call knowing that there is no call or answer to this call in the extension table. So, firstly store the current call and then, solve the goal with the program rules (recursively applying this algorithm). Once the goal has been solved (if succeeded), store the computed answer if there is no any previous answer subsuming the current one (note that, through recursion, we can deliver new answers for the same call). This so–called memoization process is implemented with the predicate **`memo/1`** in the file **`des.pl`** of the distribution, and will also be referred to as a memo function in the rest of this manual.

Negative facts are produced when a negative goal is proved by means of negation as failure (closed world assumption). In this situation, a goal as **`not(p)`** which succeeds produces the fact **`not(p)`** which is added to the answer table, just the same as proving a positive goal.

The command **`/list_et`** shows the current state of the extension table, both for answers and calls already obtained by solving one or more queries (incidentally, recall that you can focus on the contents of the extension table for a given predicate, cf. Section 5.10.4). This command is useful for the user when asking for the meaning of relations, and for the developer for examining the last calls being performed. Before

---

[9] For a complementary understanding of this section, the reader is advised to read [Diet87].

[10] A term T1 subsumes a term T2 if T1 is "more general" than T2 and both terms are unifiable. Eg: **`p(X,Y)`** subsumes **`p(a,Z)`**, **`p(X,Y)`** subsumes **`p(U,V)`**, **`p(X,Y)`** subsumes **`p(U,U)`**, but **`p(U,U)`** neither subsumes **`p(a,b)`**, nor **`p(X,Y)`**.

executing any query, the extension table is empty; after executing a query, at least the call is not empty. Also, the extension table is empty after the execution of a temporary view.[11] The extension table contains the calls made during the last fixpoint iteration (see next section for details); the calls are cleared before each iteration whereas the answers are kept. The command **`/clear_et`** clears the extension table contents, both for calls and answers.

5.11.2 Finding Stable Models

The tabling mechanism is insufficient in itself for computing all of the possible answers to a query. The rationale behind this comes from the fact that the computed information is not complete when solving a given goal, because it can use incomplete information from the goals in its defining rules (these goals can be mutually recursive). Therefore, we have to ensure that we produce all the possible information by finding a fixpoint of the memo function. First, the call table is emptied in order to allow the system to try to obtain new answers for a given call, preserving the previous computed answers. Then, the memo function is applied, possibly providing new answers. If the answer table remains the same as before after this last memo function application, we are done. Otherwise, the memo function is reapplied as many times as needed until we find a stable answer table (with no changes in the answer table). The answer table contains the stable model of the query (plus perhaps other stable models for the relations used in the computation of the given query).

The fixpoint is found in finite time because the memo function is monotonic in the sense that we only add new entries each time it is called while keeping the old ones. Repeatedly applying the memo function to the answer table delivers a finite answer table since the number of new facts that can be derived from a Datalog program is finite (recall that there are no compound terms such as **s**^{k}**(z)**). On the one hand, the number of positive facts which can be inferred are finite because there is a finite number of ground facts which can be used in a given proof, and proofs have finite depth provided that tabling prevents recomputations of older nodes in the proof tree. On the other hand, the number of negative facts which can be inferred is also finite because they are proved using negation as failure. (Failures are always finite because they are proved trying to get a success.) Finally, there are facts that cannot be proved to be true or false because of recursion. These cases are detected by the tabling mechanism which prevent infinite recursion such as in **`p :- p`**.

It is also possible that both a positive and a negative fact have been inferred for a given call. Then, an undefined fact replaces the contradictory information. The implementation simply removes the contradictory facts and informs about the undefinedness. As already indicated (see Section 6.9), the algorithm for determining undefinedness is incomplete.

5.11.3 Dependency Graphs and Stratification: Negation, Outer Joins, and Aggregates

Each time a program is consulted or modified (i.e., via submitting a temporary view or changing the database), a predicate !, graph is built [ZCF+97]. This graph

[11] The contents of the extension table in this case should be restored instead of being cleared; left for further improvements.

shows the dependencies, through positive and negative atoms, among predicates in the program. Also, a negative dependency is added for each outer join goal and aggregate goal.

This dependency graph is useful for finding a stratification for the program [ZCF+97]. A stratification collects predicates into numbered strata (1..N). A basic bottom-up computation would solve all of the predicates in stratum 1, then 2, and so on, until the meaning of the whole program is found. With our approach, we only resort to compute by stratum when a negative dependency occurs in the predicate dependency graph restricted to the query; nevertheless, each predicate that is actually needed is solved by means of the extension table mechanism described in the previous section. As a consequence, many computations are avoided w.r.t. a naïve bottom-up implementation. Outer join and aggregate goals are also collected into strata as if they were negative atoms in order to have their answer set completely defined and therefore ensure termination of the computation algorithm in presence of null values.

5.11.4 Porting to Unsupported Systems

DES is implemented with five Prolog files: **`des.pl`**, **`des_sql.pl`**, **`des_debug.pl`**, **`des_tc.pl`**, and **`des_glue.pl`**. The first file contains the common predicates for all of the platforms (both Prolog interpreters and operating systems) using the ISO standard. The second file contains the sql compiler. The third one contains the debugger code, and the fourth one the test case generator code. The last file contains Prolog system specific code, which vary from a system to another. Adapting the predicates found there should not pose problems, provided that the Prolog interpreter and operating system feature some basic characteristics (mainly about the file system commands). If you plan to port DES to other systems not described here, you will have to modify the system specific Prolog file to suit your system. If so, and if you want to figure as one of the system contributors, please send an e–mail message with the code and reference information to: **`fernan@sip.ucm.es`**, accepting that your contribution will be under the GNU General Public License. (See the appendix for details.)

5.11.5 Differences among Platforms

Ciao, SWI, and SICStus Prolog implementations use a sort which eliminates duplicates whereas GNU Prolog implementation does not.

In its current version, the Ciao system forces to use some directives for using several basic Prolog primitives. This can only be done by writing them in the core file (**`des.pl`**) of the system, making it not compatible with other platforms. This is why the core file for Ciao has some preliminary directives not found in the core file shared by the rest of the platforms. Future Ciao versions may change this particular behaviour.

See also Section 10 for consult unsupported features of source distributions.

6. Examples

The DES distribution contains the directory **`examples`** which shows several features of the system. Unless explicitly noted, all queries have been solved after the commands **`/verbose off`** and **`/pretty_print off`** have been executed.

6.1 Relational Operations (files `relop.{dl,sql}`)

The program **`relop.dl`** is intended to show how to mimic with Datalog rules the basic relational operations that can be found in the file **`relop.sql`**. It contains three relations (**`a`**, **`b`**, and **`c`**), which are used as arguments of relational operations. In order to have loaded this program and be able to submit queries you can consult it with **`/c relop`**.

```
% Relations
a(a1).
a(a2).
a(a3).

b(b1).
b(b2).
b(a1).

c(a1,b2).
c(a1,a1).
c(a2,b2).

% (Extended) Relational Algebra Operations

% pi(X)(c(X,Y))
projection(X) :- c(X,Y).

%sigma(X=a2)(a)
selection(X) :- a(X), X=a2.

% a X b
cartesian(X,Y) :- a(X), b(Y).

% a |x| b
inner_join(X) :- a(X), b(X).

% a =|x| b
left_join(X,Y) :- lj(a(X), b(Y), X=Y).

% a |x|= b
right_join(X,Y) :- rj(a(X), b(Y), X=Y).

% a =|x|= b
full_join(X,Y) :- fj(a(X), b(Y), X=Y).

% a U b
union(X) :- a(X) ; b(X).

% a - b
difference(X) :- a(X), not(b(X)).
```

Once the program is consulted, you can query it by, for example:

```
DES-Datalog> projection(X)
```

```
{
  projection(a1),
  projection(a2)
}
Info: 2 tuples computed.
```

The result of a query is the meaning of the view, i.e., the fact set for the query derived from the program whether intensionally or extensionally. In the above example, **projection(X)** corresponds to the projection of the first argument of relation **c**.

The second view in Section 4.1.7 returns:

```
Info: Processing:
  a(X) :- b(X).
{
  a(a1),
  a(a2),
  a(a3),
  a(b1),
  a(b2)
}
Info: 5 tuples computed.
```

For abolishing this program and execute the SQL statements in **relop.sql**, you can type **/abolish** and **/process relop.sql**. Note that the extension is needed in the **process** command.

Here, we depart from the Datalog interpreter and, if you are to submit SQL queries, it is useful to switch to the SQL interpreter via the command **/sql**. Otherwise, you may forget to identify your SQL query by prepending it with **/sql**, and the Datalog parser will issue a syntax error. But you may think that this error comes from your SQL sentence, and the fact is that the SQL query is interpreted as a Datalog query. So, better switch to SQL for a batch of SQL queries.

Note that in the file **relop.sql** listed below, strings are enclosed between apostrophes. This is not needed in the Datalog language. In order to execute the contents of this file, type **/process relop.sql**.

```
% Switch to SQL interpreter
/sql
% Creating tables
create or replace table a(a);
create or replace table b(b);
create or replace table c(a,b);
% Listing the database schema
/dbschema
% Inserting values into tables
insert into a values ('a1');
insert into a values ('a2');
insert into a values ('a3');
insert into b values ('b1');
insert into b values ('b2');
insert into b values ('a1');
insert into c values ('a1','b2');
```

```
insert into c values ('a1','a1');
insert into c values ('a2','b2');
% Testing the just inserted values
select * from a;
select * from b;
select * from c;
% Projection
select a from c;
% Selection
select a from a where a='a2';
% Cartesian product
select * from a,b;
% Inner Join
select a from a inner join b on a.a=b.b;
% Left Join
select * from a left join b on a.a=b.b;
% Right Join
select * from a right join b on a.a=b.b;
% Full Join
select * from a full join b on a.a=b.b;
% Union
select * from a union select * from b;
% Difference
select * from a except select * from b;
```

If we have created the relations in Datalog, we cannot access them from SQL unless they had been defined as tables or views. For example, after consulting the file **relop.dl**, we can submit:

```
create table a(a);
```

And, then, accessing with a SQL statement the tuples that were asserted in Datalog:

```
DES-SQL> select * from a;
answer(a.a) ->
{
  answer(a1),
  answer(a2),
  answer(a3)
}
Info: 3 tuples computed.
```

Otherwise, an error is submitted:

```
Error: Input not recognized as a valid SQL statement or command.
  Queries  : Consult the manual for syntax
  Commands : /Command Argument(s)
Queries and commands can optionally end with a semicolon or a
dot, resp.
```

6.2 Paths in a Graph (files `paths.{dl,sql}`)

This program[12] introduces the use of recursion in DES by defining the graph in Figure 1 and the set of tuples <origin, destination> such that there is a path from origin to destination.

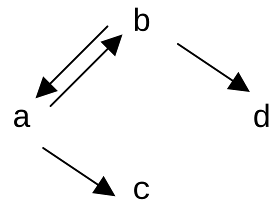


Figure 1. Paths in a Graph

The file **`paths.dl`** contains the following Datalog code, which can be consulted with **`/c paths`**:

```
% Paths in a Graph

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

path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).
```

The query **`path(X,Y)`** yields the following answer:

```
{
  path(a,a),
  path(a,b),
  path(a,c),
  path(a,d),
  path(b,a),
  path(b,b),
  path(b,c),
  path(b,d)
}
Info: 8 tuples computed.
```

The file **`paths.sql`** contains the SQL counterpart code, which can be executed with **`/process paths.sql`**:

```
create table edge(origin,destination);
insert into edge values('a','b');
insert into edge values('a','c');
insert into edge values('b','a');
insert into edge values('b','d');
create view paths(origin,destination) as
  with
    recursive path(origin,destination) as
```

[12] Adapted from [TS86].

```
    (select * from edge)
    union
    (select path.origin,edge.destination
     from path,edge
     where path.destination =edge.origin)
  select * from path;
```

So, you can get the same answer as before with the SQL statement:

```
DES-SQL> select * from paths;
answer(paths.origin, paths.destination) ->
{
  answer(a,a),
  answer(a,b),
  answer(a,c),
  answer(a,d),
  answer(b,a),
  answer(b,b),
  answer(b,c),
  answer(b,d)
}
Info: 8 tuples computed.
```

Another shorter formulation is allowed in DES with the following view definition:

```
create view path(origin,destination) as
  select * from
  (select * from edge)
  union
  (select path.origin,edge.destination
   from path,edge
   where path.destination=edge.origin)
```

6.3 Shortest Paths (file `spaths.{dl,sql}`)

Thanks to aggregate predicates, one can code the following version of the shortest paths problem (file **`spaths.dl`**), which uses the same definition of edge as the previous example:

```
path(X,Y,1) :-
  edge(X,Y).
path(X,Y,L) :-
  path(X,Z,L0),
  edge(Z,Y),
  count(edge(A,B),Max),
  L0<Max,
  L is L0+1.

sp(X,Y,L) :-
   min(path(X,Y,Z),Z,L).
```

Note that the infinite computation that may raise from using the builtin is/2 is avoided by limiting the total length of a path to the number of edges in the graph.

The following query returns all the possible paths and their corresponding minimal distances:

```
DES-Datalog> sp(X,Y,L)
{
  sp(a,a,2),
  sp(a,b,1),
  sp(a,c,1),
  sp(a,d,2),
  sp(b,a,1),
  sp(b,b,2),
  sp(b,c,2),
  sp(b,d,1)
}
Info: 8 tuples computed.
```

Below is the SQL formulation for the same problem (file **spaths.sql**) :

```
DES-SQL> create or replace view
spaths(origin,destination,length) as with recursive
path(origin,destination,length) as
(select edge.*,1 from edge)
 union
(select path.origin,edge.destination,path.length+1
 from path,edge
 where path.destination=edge.origin and
       path.length<(select count(*) from edge))
select origin,destination,min(length) from path group by
origin,destination;

DES-SQL> select * from spaths
answer(spaths.origin, spaths.destination, spaths.length) ->
{
  answer(a,a,2),
  answer(a,b,1),
  answer(a,c,1),
  answer(a,d,2),
  answer(b,a,1),
  answer(b,b,2),
  answer(b,c,2),
  answer(b,d,1)
}
Info: 8 tuples computed.
```

6.4 Family Tree (files `family.{dl,sql}`)

This (yet another classic) program defines the family tree shown in Figure 2, the set of tuples <**parent**,**child**> such that **parent** is a parent of **child** (the relation **parent**), the set of tuples <**ancestor**,**descendant**> such that **ancestor** is an ancestor of **descendant** (the relation **ancestor**), the set of tuples <**father**,**child**> such that **father** is the father of **child** (the relation **father**), and the set of tuples <**mother**,**child**> such that **mother** is the mother of **child** (the relation **mother**).

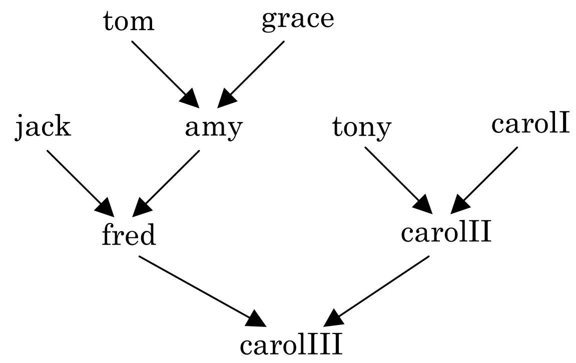


Figure 2. Family Tree

The file **family.dl** contains the following Datalog code, which can be consulted with **/c family**:

```
father(tom,amy).
father(jack,fred).
father(tony,carolII).
father(fred,carolIII).
mother(grace,amy).
mother(amy,fred).
mother(carolI,carolII).
mother(carolII,carolIII).

parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

The query **ancestor(tom,X)** yields the following answer (that is, it computes the set of descendants of **tom**):

```
{
  ancestor(tom,amy),
  ancestor(tom,carolIII),
  ancestor(tom,fred)
}
Info: 3 tuples computed.
```

Solving the view:

```
son(S,F,M) :- father(F,S),mother(M,S).
```

yields the following answer, computing the set of sons:

```
Info: Processing:
  son(S,F,M) :- father(F,S),mother(M,S).
{
  son(amy,tom,grace),
  son(carolII,tony,carolI),
  son(carolIII,fred,carolII),
  son(fred,jack,amy)
}
Info: 4 tuples computed.
```

The file **family.sql** contains the SQL counterpart code, which can be executed with **/process family.sql**:

```
/sql
create table father(father,child);
insert into father values('tom','amy');
insert into father values('jack','fred');
insert into father values('tony','carolII');
insert into father values('fred','carolIII');
create table mother(mother,child);
insert into mother values('grace','amy');
insert into mother values('amy','fred');
insert into mother values('carolI','carolII');
insert into mother values('carolII','carolIII');
create view parent(parent,child) as
  select * from father
  union
  select * from mother;
create or replace view ancestor(ancestor,descendant) as
  select parent,child from parent
  union
  select parent,descendant from parent,ancestor
   where parent.child=ancestor.ancestor;
```

The two example queries above can be formulated in SQL as:

```
select * from ancestor where ancestor='tom';

select child,father,mother
 from father,mother
 where father.child=mother.child;
```

6.5 Basic Recursion Problem (file `recursion.dl`)

This example is intended to show that queries involving recursive predicates do terminate thanks to DES fixpoint solving, by contrast with Prolog's usual SLD resolution.

```
p(0).
p(X) :- p(X).
p(1).
```

The query **p(X)** returns the inferred facts from the program irrespective of the apparent infinite recursion in the second rule. (Note that the Prolog goal **p(1)** does not terminate. You can easily check it out with **/prolog p(1)**.)

6.6 Transitive Closure (files `tranclosure.{dl,sql}`)

With this example, we show a possible use of mutual recursion by means of a Datalog program that defines the transitive closure of the relations p and q[13]. It can be consulted with **/c tranclosure**.

[13] Taken from [Diet87].

```
p(a,b).
p(c,d).
q(b,c).
q(d,e).
pqs(X,Y) :- p(X,Y).
pqs(X,Y) :- q(X,Y).
pqs(X,Y) :- pqs(X,Z),p(Z,Y).
pqs(X,Y) :- pqs(X,Z),q(Z,Y).
```

The query **`pqs(X,Y)`** returns the whole set of inferred facts that model the transitive closure.

The file **`tranclosure.sql`** contains the SQL counterpart code, which can be executed with **`/process tranclosure.sql`**:

```
create table p(x,y);
insert into p values ('a','b');
insert into p values ('c','d');
create table q(x,y);
insert into q values ('b','c');
insert into q values ('d','e');
create view pqs(x,y) as
  select * from p
  union
  select * from q
  union select pqs.x,p.y from pqs,p where pqs.y=p.x
  union select pqs.x,q.y from pqs,q where pqs.y=q.x;
```

The query **`select * from pqs`** returns the same answer as before.

6.7 Mutual Recursion (files `mutrecursion.{dl,sql}`)

The following program shows a basic example about mutual recursion:

```
p(a).
p(b).
q(c).
q(d).
p(X) :- q(X).
q(X) :- p(X).
```

Submitting the goal **`p(X)`**, we get:

```
{
  p(a),
  p(b),
  p(c),
  p(d)
}
Info: 4 tuples computed.
```

which is the same set of values for arguments for the query **`q(X)`**. The file **`mrtc.dl`** is a combination of this example and that of the previous section.

The file **`mutrecursion.sql`** contains the SQL counterpart code, which can be executed with **`/process mutrecursion.sql`**:

```
/sql
/assert p(a)
/assert p(b)
/assert q(c)
/assert q(d)
create view q(x) as select * from q;
create or replace view p(x) as select * from q;
create or replace view q(x) as select * from p;
```

Note that it is needed to build a void view for **q** in order to have it declared when defining the view **p**. The void view is then replaced by its actual definition. The contents of both views can be tested to be equal with:

```
select * from p;
select * from q;
```

6.8 Farmer-Wolf-Goat-Cabbage Puzzle (file `puzzle.dl`)

This example[14] shows the classic Farmer–Wolf–Goat–Cabbage puzzle (also Missionaries and Cannibals as another rewritten form). The farmer, wolf, goat, and cabbage are all on the north shore of a river and the problem is to transfer them to the south shore. The farmer has a boat which he can row taking at most one passenger at a time. The goat cannot be left with the wolf unless the farmer is present. The cabbage, which counts as a passenger, cannot be left with the goat unless the farmer is present. The following program models the solution to this puzzle. The relation **state/4** defines the valid states under the specification (i.e., those situations in which there is no danger for any of the characters in our story; a state in which the goat is left alone with the cabbage may result in an eaten cabbage) and imposes that there is a previous valid state from which we depart from. The arguments of this relation are intended to represent (from left to right) the position (north –**n**– or south –**s**– shore) of the farmer, wolf, goat, and cabbage. We use the relation **safe/4** to verify that a given configuration of positions is valid. The relation **opp/2** simply states that north is the opposite shore of south and viceversa.

```
% Initial state
state(n,n,n,n).
% Farmer takes Wolf
state(X,X,U,V) :-
  safe(X,X,U,V),
  opp(X,X1),
  state(X1,X1,U,V).
% Farmer takes Goat
state(X,Y,X,V) :-
  safe(X,Y,X,V),
  opp(X,X1),
  state(X1,Y,X1,V).
% Farmer takes Cabbage
state(X,Y,U,X) :-
  safe(X,Y,U,X),
  opp(X,X1),
```

[14] Adapted from [Diet87].

```
  state(X1,Y,U,X1).
% Farmer goes by himself
state(X,Y,U,V) :-
  safe(X,Y,U,V),
  opp(X,X1),
  state(X1,Y,U,V).

% Opposite shores (n/s)
opp(n,s).
opp(s,n).

% Farmer is with Goat
safe(X,Y,X,V).
% Farmer is not with Goat
safe(X,X,X1,X) :- opp(X,X1).
```

If we submit the query **state(s,s,s,s)**, we get the expected result:

```
{
  state(s,s,s,s)
}
Info: 1 tuple computed.
```

That is, the system has proved that there is a serial of transfers between shores which finally end with the asked configuration (this problem is not modeled to show this serial, although it could be). If we ask for the extension table contents regarding the relation **state/4** (with the command **/list_et state/4**), we get for the answers:

```
{
  state(n,n,n,n),
  state(n,n,n,s),
  state(n,n,s,n),
  state(n,s,n,n),
  state(n,s,n,s),
  state(s,n,s,n),
  state(s,n,s,s),
  state(s,s,n,s),
  state(s,s,s,n),
  state(s,s,s,s)
}
Info: 10 tuples in the answer set.
```

This is the complete set of valid states which includes all of the valid paths from **state(n,n,n,n)** to **state(s,s,s,s)**. However, the order of states to reach the latter is not given, but we can find it by observing this relation, i.e.:

```
state(n,n,n,n) → Farmer takes Goat to south shore →
state(s,n,s,n) → Farmer returns to north shore →
state(n,n,s,n) → Farmer takes Wolf to south shore →
state(s,s,s,n) → Farmer takes Goat to north shore →
state(n,s,n,n) → Farmer takes Cabbage to south shore →
state(s,s,n,s) → Farmer returns to north shore →
state(n,s,n,s) → Farmer takes Goat to south shore →
state(s,s,s,s)    Final safe state
```

Observe that there is two states in the relation **state/4** which do not form part of the previous path:

```
state(s,n,s,s)
state(n,n,n,s)
```

These states come from another possible path:[15]

```
state(n,n,n,n) → Farmer takes Goat to south shore →
state(s,n,s,n) → Farmer returns to north shore →
state(n,n,s,n) → Farmer takes Cabbage to south shore →
state(s,n,s,s) → Farmer takes Goat to north shore →
state(n,n,n,s) → Farmer takes Wolf to south shore →
state(s,s,s,n) → Farmer takes Goat to north shore →
state(s,s,n,s) → Farmer returns to north shore →
state(n,s,n,s) → Farmer takes Goat to south shore →
state(s,s,s,s)    Final safe state
```

6.9 Paradoxes (file `russell.dl}`)

When negation is used, we can find paradoxes, such as the Russell's paradox (the barber in a town shaves every person who does not shave himself) shown in the next example (please note that this example is not stratified and, in general, we cannot ensure correctness for non-stratifiable programs):

```
shaves(barber,M) :- man(M), not(shaves(M,M)).
man(barber).
man(mayor).
```

If we submit the query **shaves(X,Y)**, we get the positive facts as well as a set of undefined inferred information (in our example, whether the barber shaves himself), as follows (here, verbose output is enabled):

```
DES-Datalog> shaves(X,Y)
Warning: Unable to ensure correctness for this query.
{
  shaves(barber,mayor)
}
Info: 1 tuple computed.
Undefined:
{
  shaves(barber,barber)
}
Info: 1 tuple undefined.
```

If we look at the extension table contents by submitting the command **/list_et**, we get as answers:

```
Answers:
{
  man(barber),
```

[15] Remember that the system returns *all* of the possible solutions.

```
  man(mayor),
  not(shaves(mayor,mayor)),
  shaves(barber,mayor)
}
Info: 4 tuples in the answer set.
```

We can see that, in particular, we have proved additional negative information (the mayor does not shaves himself) and that no information is given for the undefined facts. The current implementation uses an incomplete algorithm for finding such undefined facts. We can see this incompleteness by adding the following rule:

```
shaved(M) :- shaves(barber,M).
```

The query **shaved(M)** returns:

```
Warning: Unable to ensure correctness for this query.
{
  shaved(mayor)
}
Info: 1 tuple computed.
```

That is, the system is unable to prove that **shaved(barber)** is undefined.

If you look at the predicate dependency graph and the stratification of the program:

```
DES-Datalog> /pdg

Nodes: [man/1,shaved/1,shaves/2]
Arcs : [shaves/2-shaves/2,shaves/2+man/1,shaved/1+shaves/2]

DES-Datalog> /strata

[non-stratifiable]
```

you get the predicate dependency graph shown in Figure 4, and you are informed that the program is non-stratifiable. This figure shows a negation in a cycle, so that the program is not stratifiable. (The system warned of this situation when the program was loaded.)

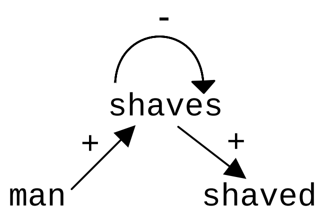


Figure 4. Predicate Dependency Graph for **russell.dl**

However, even when a program is non-stratifiable, there may exist a query with an associated predicate dependency subgraph so that negation does not occur in any cycle. For instance, this occurs with the query **man(X)** in this program:

```
DES-Datalog> man(X)
Info: Stratifiable subprogram found for the given query.
```

```
{
  man(barber),
  man(mayor)
}
Info: 2 tuples computed.
```

Here, the system recomputed the strata for the predicate dependency subgraph, and informed that it found a stratifiable subprogram for such a query. In this simple case, no more negations were involved in the subgraph, but more elaborated dependencies can be found in other examples (cf. Sections 6.10 and 6.11).

Stratification may be needed for programs without negation as long as a temporary view contains a negated goal. Consider the following view under the program **relop.dl** (rules in the program with negation are not present in the subgraph for the query **d(X)**):

```
DES-Datalog> d(X) :- a(X), not(b(X))
Info: Processing:
  d(X) :- a(X),not(b(X)).
{
  d(a2),
  d(a3)
}
Info: 2 tuples computed.
```

In this view, the query **d(X)** is solved with a solve-by-stratum algorithm, described in Section 5.11.3. In this case, this means that the goal **b(X)** is solved before obtaining the meaning of **d(X)** because **b** is in a lower stratum than **d** and it is needed for the computation of **d**.

The basic paradox **p:-not(p)** can be found in the file paradox.dl, whose model is undefined as you can test with the query **p**.

6.10 Parity (file `parity.dl`)

This example program[16] is intended to compute the parity of a given base relation **br(X)**, i.e., it can determine whether the number of elements in the relation (cardinality) is even or odd by means of the predicates **br_is_even**, and **br_is_odd**, respectively. The predicate **next** defines an ascending chain of elements in **br** based on their textual ordering, where the first link of the chain connects the distinguished node **nil** to the first element in **br**. The predicates **even** and **odd** define the even, resp. odd, elements in the chain. The predicate **has_preceding** defines the elements in **br** such that there are previous elements to a given one (the first element in the chain has no preceding elements). The rule defining this predicate includes an intended error (fourth rule in the example) which will be used in Section 6.13 to show how it is catched by the declarative debugger.

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

16 Adapted from [ZCF+97].

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

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

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

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

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

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

% Base relation
br(a).
br(b).
```

6.11 Grammar (file `grammar.dl`)

Parsers can also be coded as Datalog programs. In this example[17], a simple left-recursive grammar analyser is coded for the following grammar rules.

A –> a

A –> Ab

A –> Aa

It was tested with the input string "ababa", which is coded with the relation **`t(F,T,L)`**, **`F`** for the position of token **`T`** that ends at position **`L`**.

```
t(1,a,2).
t(2,b,3).
t(3,a,4).
t(4,b,5).
t(5,a,6).
a(F,L) :- t(F,a,L).
a(F,L) :- a(F,M), t(M,b,L).
a(F,L) :- a(F,M), t(M,a,L).
```

[17] Taken from [FD92].

```
DES-Datalog> a(1,6)
{
  a(1,6)
}
Info: 1 tuple computed.
```

6.12 Fibonacci (file `fib.dl`)

The all-time classics Fibonacci program[18] can be coded in DES thanks to arithmetic built-ins. It can be formulated as follows:

```
fib(0,1).
fib(1,1).
fib(N,F) :-
  N>1,
  N2 is N-2,
  fib(N2,F2),
  N1 is N-1,
  fib(N1,F1),
  F is F2+F1.
```

Since DES is implemented with extension tables, computing high Fibonacci numbers is possible with linear complexity:

```
DES-Datalog> fib(1000,F)
{
fib(1000,70330367711422815821835254877183549770181269836358732742604905087154537118196933579742249494562611733487750449241765991088186363265450223647106012053374121273867339111198139373125598767690091902245245323403501)
}
Info: 1 tuple computed.
```

6.13 Hanoi Towers (file `hanoi.dl`)

Another well-known toy puzzle is the towers of Hanoi, which can be coded as:

```
hanoi(1,A,B,C).
hanoi(N,A,B,C) :-
  N>1,
  N1 is N-1,
  hanoi(N1,A,C,B),
  hanoi(N1,C,B,A).
```

We can submit the following query for 10 discs:

```
DES-Datalog> hanoi(10,a,b,c)
{
  hanoi(10,a,b,c)
}
Info: 1 tuple computed.
```

[18] Taken from [FD92].

Note that the answer to this query does not reflect the movements of the discs, which can be otherwise shown as the intermediate results kept in the extension table:

```
DES-Datalog> /list_et hanoi
Answers:
{
  hanoi(1,a,c,b),
  hanoi(1,b,a,c),
  hanoi(1,c,b,a),
  hanoi(2,a,b,c),
  hanoi(2,b,c,a),
  hanoi(2,c,a,b),
  hanoi(3,a,c,b),
  hanoi(3,b,a,c),
  hanoi(3,c,b,a),
  hanoi(4,a,b,c),
  hanoi(4,b,c,a),
  hanoi(4,c,a,b),
  hanoi(5,a,c,b),
  hanoi(5,b,a,c),
  hanoi(5,c,b,a),
  hanoi(6,a,b,c),
  hanoi(6,b,c,a),
  hanoi(6,c,a,b),
  hanoi(7,a,c,b),
  hanoi(7,b,a,c),
  hanoi(7,c,b,a),
  hanoi(8,a,b,c),
  hanoi(8,b,c,a),
  hanoi(8,c,a,b),
  hanoi(9,a,c,b),
  hanoi(9,c,b,a),
  hanoi(10,a,b,c)
}
Info: 27 tuples in the answer set.
...
```

6.14 Other Examples

The files **bom.dl** (bill of materials) and **trains.dl** (train connections) show more example applications including negation. Other examples are **orbits.dl** (a cosmos tiny database), **sg.dl** (same generation for a family database) and **tc.dl** (transitive closure).

7. Contributions

This section collects the contributions from external developers up to now:

- Test Case Generator.
  Authors: Rafael Caballero-Roldán, Yolanda García-Ruiz, and Fernando Sáenz-Pérez
  Date: 10/2009 (upgraded version supported since DES 1.8.0)
  Description: Tool for generating test cases for SQL views

License: GPL
Contact: Yolanda García-Ruiz

- Datalog Declarative Debugger.
  Authors: Rafael Caballero-Roldán, Yolanda García-Ruiz, and Fernando Sáenz-Pérez
  Date: 5/2007
  Description: Tool for the declarative debugging of Datalog programs
  License: GPL
  Contact: Yolanda García-Ruiz

- ACIDE (A Configurable Development Environment).
  Authors: Diego Cardiel Freire, Juan José Ortiz Sánchez, Delfín Rupérez Cañas (SI 2006/2007), and Miguel Martín Lázaro (SI 2007/2008) leaded by Fernando Sáenz.
  Date: 3/2007 (ACIDE 0.1, first version), 11/2008 (ACIDE 0.7, current alpha version)
  Description: This project is aimed to provide a multiplatform configurable integrated development environment which can be configured in order to be used with any development system such as interpreters, compilers and database systems. Features of this system include: project management, multifile editing, syntax colouring, and parsing on-the-fly (which informs of syntax errors when editing programs prior to the compilation).
  License: GPL.
  Project Web Page: **`http://acide.sourceforge.net/`**

- Emacs development environment.
  Author: Markus Triska.
  Date: 2/22/2007
  Description: Provides an integration of DES into Emacs. Once a Datalog file has been opened, you can consult it by pressing F1 and submit queries and commands from Emacs. This works at least in combination with SWI Prolog (it depends on the –s switch); other systems may require slight modifications.
  License: GPL.
  Project Web Page: **`http://stud4.tuwien.ac.at/~e0225855/index.html`**
  Contact: **`markus.triska@gmx.at`**
  Installation: Copy **`des.el`** (in the contributors web page) to your home directory and add to your **`.emacs`**:

```
(load "~/des")
; adapt the following path as necessary:
(setq des-prolog-file "~/des/systems/swi/des.pl")
(add-to-list 'auto-mode-alist '("\\.dl$" . des-mode))
```

  Restart Emacs, open a **`*.dl`** file to load it into a DES process (this currently only works with SWI Prolog). If the region is active, F1 consults the text in the region. You can then interact with DES as on a terminal.

8. Related Work

There has been a high amount of work around deductive databases [RU95] (its interest delivered many workshops and conferences for this subject) which dealt to several systems. However, to the best of our knowledge, there is no a friendly system oriented to introducing deductive databases with several query languages to students. Nevertheless, on the one hand, we can comment some representative deductive database systems. On the other hand, also some technological transfers to face real-world problems.

8.1 Deductive Database Systems

ConceptBase [JJNS98] is a multi-user deductive object manager mainly intended for conceptual modeling and coordination in design environments. It is multiplatform, object-oriented, it enjoys integrity constraints, database updates and several other interesting features.

The LDL project at MCC lead to the LDL++ system [AOTWZ03], a deductive database system with features such as X-Y stratification, set and complex terms, database updates and aggregates. It can be currently used through Internet using a Java–enabled client.

DLV [FP96] is a multiplatform system for disjunctive Datalog with constraints, true negation (à la Gelfond & Lifschitz) and queries. It includes the K planning system, a frontend for abductive diagnosis and Reiter's diagnosis, support for inheritance, and an SQL frontend which prototypes some novel SQL3 features. DLV^{DB} is an extension of DLV which provides interfaces with relational databases, taking advantage of their efficient implementations to speed-up computations.

XSB [RSSWF97] (**`http://xsb.sourceforge.net/`**) is an extended Prolog system that can be used for deductive database applications. It enjoys a well–founded semantics for rules with negative literals in rule bodies and implements tabling mechanisms. It runs both on Unix/Linux and Windows operating systems. Datalog++ [Tang99] is a front-end for the XSB deductive database system.

bddbddb [WL04] stands for BDD-Based Deductive DataBase. It is an implementation of Datalog that represents the relations using binary decision diagrams (BDDs). BDDs are a data structure that can efficiently represent large relations and provide efficient set operations. This allows bddbddb to efficiently represent and operate on extremely large relations.

IRIS (Integrated Rule Inference System) [IRIS2008] is a Java implementation of an extensible reasoning engine for expressive rule-based languages provided as an API. Supports safe or un-safe Datalog with (locally) stratified or well-founded negation as failure, function symbols and bottom-up rule evaluation.

Coral [RSSS94] is a deductive system with a declarative query language that supports general Horn clauses augmented with complex terms, set–grouping, aggregation, negation, and relations with tuples that contain (universally quantified) variables. It only runs under Unix platforms. There is also a version which allows object–oriented features, called Coral++ [SRSS93].

FLORID (F-LOgic Reasoning In Databases) [KLW95] is a deductive object-oriented database system supporting F-Logic as data definition and query language. With the increasing interest in semistructured data, Florid has been extended for handling semistructured data in the context of Information Integration from the Semantic Web.

The NAIL! project delivered a prototype with stratified negation, well–founded negation, and modularity stratified negation. Later, it added the language Glue, which is essentially single logical rules, with SQL statements wrapped in an imperative conventional language [PDR91, DMP93]. The approach of combining two languages is similar to the aforementioned Coral, which uses C++. It does not run on Windows platforms.

Another deductive database following this combination of declarative and imperative languages is Rock&Roll [BPFWD94].

ADITI 2 [VRK+91] is the current version of a deductive database system which uses the logic/functional programming language Mercury. It does not run on Windows platforms. There is no further development planned for Aditi.

See also the Datalog entry in Wikipedia (**`http://en.wikipedia.org/wiki/Datalog`**).

8.2 Technological Transfers

Datalog has been extensively studied and is gaining a renowned interest thanks to their application to ontologies [FHH04], semantic web [CGL09], social networks [RS09], policy languages [BFG07], and even for optimization [GTZ05]. Companies as LogicBlox, Exeura, Semmle, and Lixto embody Datalog-based deductive database technologies in the solutions they develop. The high-level expressivity of Datalog and its extensions has therefore been acknowledged as a powerful feature to deal with knowledge-based information.

The first commercial oriented deductive database system was the Smart Data System (SDS) and its declarative query language Declarative Reasoning (DECLARE) [KSSD94], with support for stratified negation and sets. Currently, XSB and DLV have been projected to spin-off companies and they develop deductive solutions to real-world problems.

9. Future Enhancements

The following list (in order of importance) suggests some points to address for enhancing DES:

- Multiple DB connections
- Hypothetical queries
- Disjunctive heads
- Information about cycles involving negation in the loaded program
- Complete algorithm for finding undefined information
- Constraints (reals, integers, enumerated types)
- Precise error reporting for SQL and Datalog syntax errors

If you find worthwhile for your application either some of the points above, or others not listed, please inform the author for trying to guide the implementation to the most demanded points.

10. Caveats and Limitations

- Datalog:

  - No database updates via Datalog rules are allowed
  - No compound terms as arguments in user relations
- SQL:
  - Table and column identifiers are case sensitive
  - Some incorrect SQL statements are not rejected (as those containing a GROUP BY clause and columns in the projection list which do not occur in the grouping list)
  - Computable SQL statements follow the grammar in Section 4.2.7 of this manual. The current grammar parses extra clauses which cannot be computed yet (e.g., **`ORDER BY`**, **`ALL`**, **`ANY`**, ...)
  - See also Section 5.1.1 regarding ODBC connections
- Test case generator:
  - Source distribution for Ciao does not suppor this feature since there is no provision for an FD library with required operators
  - Source distribution for GNU Prolog does not support negative integers
- Miscellanea:
  - Line numbers of the consulted programs are not reported for the source distribution of GNU Prolog since this system does not provide this information through **`read_term`**.
  - GNU Prolog source distribution does not detect the ISO arithmetic error **`float_overflow`**, **`int_overflow`**, and **`int_underflow`**, so that it is possible to get erroneous results when computations involve large numbers.
  - Ciao Prolog, GNU Prolog and SWI Prolog source distributions do not allow arithmetic expressions involving **`log/2`**
  - The source distribution of GNU Prolog does not support the test case generator since a CLP(FD) constraint library is required.
  - The source distribution of Ciao Prolog does not support the test case generator since a CLP(FD) constraint library is required.
  - **`/abolish`** commands remove Datalog rules, including those coming from SQL-to-Datalog compilations. Therefore, if a view is defined and Datalog rules are removed, only the view definition remains. This view becomes thus incomplete
  - Batch processing cannot be nested
  - Last line in a processed file has to end with a carriage return

11. Release Notes History

This section lists release notes of all software releases in reverse chronological order.

11.1 Version 2.0 of DES (released on August, 31st, 2010)

- Enhancements:
  - Connection to RDBs via ODBC connections (DSN providers as MySQL, MS Access, Oracle, ...) RDB tables and views can be queried both from SQL and Datalog
  - Duplicates are allowed in answers, both for Datalog and SQL
  - Datalog and SQL tracers
  - New commands:
    - **`/open_db`** ***Name*** **`[`*Options*`]`** Open and set the current ODBC connection to ***Name***, where ***Options***=[**`user(`*Username*`)`**] [**`password(`*Password*`)`**]. This connection must be already defined at the OS layer
    - **`/close_db`** Close the current ODBC connection
    - **`/current_db`** Display the current ODBC connection name and DSN provider
    - **`/duplicates`** Display whether duplicates are enabled
    - **`/duplicates`** ***Switch*** Enable or disable duplicates (**`on`** or **`off`**, resp.)
    - **`/trace_sql`** ***View*** **`[`*Order*`]`** Trace a SQL view in the given order (**`postorder`** or the default **`preorder`**)
    - **`/trace_datalog`** ***Goal*** **`[`*Order*`]`** Trace a Datalog basic goal in the given order (**`postorder`** or the default **`preorder`**)
    - **`/output Switch`** Enable or disable display output (**`on`** or **`off`**, resp.)
    - **`/save_ddb Filename`** Save the current Datalog database to a file
    - **`/restore_ddb Filename`** Restore the Datalog database in the given file (same as **`consult`**)
  - Results from **`SELECT SQL`** statements (those sent to an ODBC connection) can contain duplicates
  - Added **`UPDATE`** SQL statement
  - Added **`varchar2`** Oracle SQL datatype
  - Remarks can now start with '**`--`**', as in Oracle SQL
  - Both **`EXCEPT`** and **`MINUS`** are allowed to express SQL set difference
  - SQL user identifiers can be enclosed between quotation marks (either double quotes `""`, or square brackets `[]`, or backquotes ` `` `)
  - Closing the opened log file, if any, on quitting
  - Added timing information to SQL query processing, including listings which may include view processing from RDBs
  - Some dead code removal

- Changes:
  - New port to SICStus 4.*x*, which replaces the old port to SICStus 3.*x*
  - The command **`debug`** is renamed as **`debug_datalog`**
  - Executables have been built with SWI-Prolog, instead of SICStus Prolog
- Fixed bugs:
  - Asserting rules with a number as atom/string changed the type to number, as in **`/assert t('1')`**, which asserted **`t(1)`** instead of **`t('1')`**
  - Disjunctions in aggregate goals might lead to missing answers, as in **`group_by((p(X,Y),(Y=a;Y=b)),[X],C=count)`**
  - Some infix builtins were accepted without delimiting blanks, as **`Xis1`**, posed as a goal, and interpreted as **`X is 1`**
- Caveats and limitations:
  - See Section 10 of the user manual
- Known bugs:
  - The projection list of a natural outer join is not correct in all cases
  - Disjunctions in having conditions in the **`group_by`** clause may display errors which are not
  - Operator precedence in SQL conditions are not correctly handled. Use parentheses to ensure correct operator applications

11.2 Version 1.8.1 of DES (released on March, 17th, 2010)

- Fixed bugs:
  - The Windows and Linux executable distributions lacked some libraries regarding test case generation, which have been added in the current distributions
- Caveats and limitations:
  - See Section 10 of the user manual
- Known bugs:
  - The projection list of a natural outer join is not correct in all cases
  - Disjunctions in having conditions in the **`group_by`** clause may display errors which are not

11.3 Version 1.8.0 of DES (released on December, 18th, 2009)

- Enhancements:
  - An advanced test case generator supporting positive-negative, positive and negative test cases for views, ranging over integer and string data types
  - New command:

        - **/tc_size *Min Max*** Sets the minimum and maximum number of tuples generated for a test case
    - New use for existing command:
        - **/test_case *View [Options]*** Generates test case classes for the view ***View***. ***Options*** may include a class and/or an action parameters. The test case class is indicated by the values **all** (positive-negative), **positive**, or **negative** in the class parameter. The action is indicated by the values **display** (only display tuples), **replace** (replace contents of the involved tables by the computed test case), or **add** (add the computed test case to the contents of the involved tables) in the action parameter. Default parameters are **all** and **display**
    - More precise type inferring system
    - Enhanced syntax error reporting when consulting Datalog programs. An offending rule which is a valid term but is not a valid Datalog rule is listed together with location information
    - Enhanced pretty-print:
        - Rules: disjunctive bodies and quoted constants
        - SQL: indentation
        - **/dbschema**: bullets and expanded indentation
    - Informing that a goal cannot be debugged when its predicate is not defined
    - New switch for existing command:
        - **/timing detailed** Displays detailed elapsed time (parsing, computation, display and total elapsed times)
    - Line number information of consulted files is available also for the source distributions of both Ciao and SWI Prolog
- Changes:
    - The displayed integer type for tables and views has changed from **int** to **integer**
    - Any sequence of characters enclosed between quotes are allowed as a constant, as **'2*3'**
    - A bit more precise verbose output messages
- Fixed bugs:
    - Select statements with empty relations and **group_by** gave incorrect results
    - Translations of disjunctions in **group_by** conditions involving shared variables were incorrect
    - Some output displays were not logged via the command **/log**
    - Rule retraction may behave incorrectly when compiled rules cannot be differentiated
    - When a set of tables were dropped, their foreign keys were not

  - A renaming in the projection list of a SQL statement with the same identifier as input relations was incorrectly translated
  - Dropping and recreating a view failed to delete the defining Datalog rules for the rule, raising a warning
  - Removed meaningless warning message when redefining a table
  - Consulting a datalog program with syntax errors when safety is enabled yielded a loop
  - When asserting a rule and simplification enabled, the correct variable names where not displayed in the translation in some cases
- Caveats and limitations:
  - See Section 10 of the user manual
- Known bugs:
  - The projection list of a natural outer join is not correct in all cases
  - Disjunctions in having conditions in the **`group_by`** clause may display errors which are not

11.4 Version 1.7.0 of DES (released on October, 30th, 2009)

- Enhancements:
  - Extended SQL grammar and processor to cope with types as well as table and column constraints (primary key and foreign key)
  - Type system for SQL. Primitive types include: **`char`**, **`char(n)`**, **`varchar(n)`**, **`varchar`**, **`string`**, **`int`**, **`integer`**, and **`real`**
  - Basic type checking/inferring system for SQL views. Inferred types for views are displayed via **`/dbschema`** and, for autoviews, in the answer relation. Inferring precision is low (the types of expressions and numbers are not inferred)
  - Domain, primary key, and referential integrity constraints for tables created with SQL statements
  - Datalog aggregate predicates: **`group_by/3`**, **`count/3`**, **`count/2`**, **`sum/3`**, **`times/3`**, **`avg/3`**, **`min/3`**, and **`max/3`**
  - Datalog aggregate functions: **`count/0`**, **`count/1`**, **`sum/1`**, **`times/1`**, **`avg/1`**, **`min/1`**, and **`max/1`**
  - Datalog predicate builtins: **`is_null/1`** and **`is_not_null/1`** for determining whether their single argument is a null value or not, respectively
  - Test case generation for views
  - New commands:
    - **`/test_case`** ***`View`*** Generates all test case classes of for the given view
    - **`/p`** ***`Filename`*** Shorthand for **`/process`** ***`Filename`***

- Upgraded commands:
  - **/listing *Head*** Lists all rules whose heads are subsumed by ***Head***
  - **/listing *Head:-Body*** Lists all rules that are subsumed by ***Head:-Body***
- The command **process** looks for its input filename, allowing to omit the extensions **.sql** and **.ini**
- Comparison operators can include arithmetic expressions, as in **A<2*B**. This also means that equality behaves more generally than **is/2**, as shown in the query **sqrt(2)=X**, which returns **{ sqrt(2) = 1.4142135623730951 }**
- Some arithmetic expressions are precomputed when translating SQL statements to Datalog rules
- Displaying the number of tuples in rule listings, retracts, and abolishes
- Adding development flag status to the listing of **/status**

* Changes:
  - A table definition with a CREATE TABLE statement must include a type for each attribute. Former table definitions (up to version 1.6.2) are no longer valid
  - Evaluation of an arithmetic expression including a null value returns a null, instead of raising an exception
  - Operands of comparison operators are evaluated. Only arithmetic expressions are allowed, up to now. So, **X=Y+2** is allowed whenever Y is bound
  - The distribution files **des1.pl**, **dessql.pl**, and **desdebug.pl** have been renamed to **des_glue.pl**, **des_sql.pl**, and **des_debug.pl**, respectively
* Fixed bugs:
  - Development listings via **/dbschema** were not displaying compiled Datalog rules
  - String constants including only digits were incorrectly parsed as numbers
  - Failed to parse SQL set statements involving constants in the projection list
  - Nulls were not correctly read from files
  - **IS NULL** and **IS NOT NULL** in SQL statements were not behaving correctly
  - Safety checks involving disjunctions were not always properly performed, as in **p(X) :- q(X);r(X)**
  - The command **/operators** was never implemented but listed via **/help**. It has been removed
  - Listings of exploded rules were not displaying the correct source variable names in bodies
  - Some rules could not been asserted under simplification, as **p(X):-X=1;X=2**
  - Error when a multiply renamed table occurs in a SQL statement, as in **select * from t t1,t t2 where t1.a=t2.a**

- Caveat:
    - Batch processing cannot be nested
- Known bugs:
    - The projection list of a natural outer join is not correct in all cases
    - Disjunctions in having conditions in the `group_by` clause may yield to errors which are not

11.5 Version 1.6.2 (released on March, 10th, 2009)

- Enhancements:
    - Null values has been included both for Datalog programs and SQL statements
    - Novel outer join Datalog functions: `lj/3`, `rj/3`, and `fj/3`
    - Outer join SQL clauses added: `LEFT [OUTER] JOIN`, `RIGHT [OUTER] JOIN`, and `FULL [OUTER] JOIN`
    - Solving algorithm enhanced for stratified queries. Partial recomputations of lower-stratum predicates are avoided
    - Compilation of SQL `WHERE` conditions to Datalog rules now provides shorter and more efficient programs
    - Disjunctions in Datalog rule bodies
    - New commands:
        - `/development` ***`Switch`*** Enables/Disables development listings. These listings show the source-to-source translations needed to handle null values, Datalog outer join built-ins, and disjunctive literals
        - `/development` Displays whether development listings are enabled
        - `/simplification` ***`Switch`*** Enables/Disables simplification of Datalog rules. Rules with equalities, `true`, and `not(`***`BooleanValue`***`)` are simplified
        - `/simplification` Displays whether rule simplification is enabled
    - `WHERE` conditions accept arithmetic expressions (e.g., `1+t.a>3`)
    - Display of the number of undefined computed tuples, and the number of tuples in the extension table answer and call sets
    - Parentheses in Datalog rule bodies, not only in arithmetic expressions, are allowed
    - Parenthesered listings of Datalog rule bodies, making more readable bodies with conjunctions and disjunctions
    - Simplification of rules containing equalities
- Changes:

- Rule listings are grouped by predicate name and arity. For a given predicate name and arity, facts come first, followed by rules with right hand sides. The order of facts and rules follows Prolog standard order between terms
- Datalog rules resulting from translating views change the naming convention to (the more readable) *ViewName_Arity_Number* in lieu of *ViewName*`$p`*Number*
- Results from Datalog autoviews are given the relation name **`answer`** instead of **`autoview`**
- Pretty-print is applied to all Datalog rule listings
- Safety warnings are not hidden by computability warnings

- Fixed bugs:
  - Unformatted SQL statement display for certain conditions and joins
  - Parsing error for **`EXISTS`** clause (no blanks between **`EXISTS`** and opening parenthesis were allowed)
  - SQL arithmetic functions could only be written in lowercase
  - Some **`WHERE`** conditions incorrectly translated into Datalog conditions (bug introduced in version 1.6.1)
  - Some **`WHERE`** conditions involving parentheses incorrectly parsed
  - Correlated SQL queries with non-basic conditions were incorrectly translated into Datalog rules
  - **`DELETE`** SQL statements failed to be parsed (copy-paste bug introduced in version 1.6.1)
  - Some unsafe Datalog queries were not rejected for computation (as **`X=Y`**)
  - During startup batch processing of **`des.ini`**, some tasks upon exceptions were not performed
  - Typing **`des.`** in a Prolog interpreter after abnormally quitting the system did not result in exception catching anymore
  - A class of unsafe rules was not be able to be preprocessed for reordering body goals, yielding non-termination
  - Incomplete error message

- Known bugs:
  - The projection list of a natural outer join is not correct in all cases

- Caveat:
  - Computable SQL statements follow the grammar in the manual. The current grammar parses extra clauses which cannot be computed yet (e.g., **`ORDER BY`**, ...)

11.6 Version 1.6.1 (released on November, 10th, 2008)

- Enhancements:

- Arithmetic expressions are allowed in the projection list of **`SELECT`** statements
- Subqueries in comparisons (**=**, **<**, **>**, ...), in either side or even in both sides of the comparison operator (read as **`ANY`**, not **`ALL`**, which is unsupported up to now)
- Display of the number of computed, inserted and deleted tuples
- Commands are case-insensitive
- Some tweaks on the SQL parsing code for making it hopefully more understandable and efficient
- The answer to a SQL query is a relation with name **`'answer'`**, and its schema is displayed when solving it
- A new use for the **`/dbschema`** command: Now, it accepts an optional argument (a database object, which can be a view or a table name) for restricting the displayed schema
- The **`/dbschema`** command informs about local view definitions for each view
- A new SQL DDL statement: **`drop database`**, which drops the database (including tables, views, and rules)
- Stratifications are not computed during building a view that involves local views. As a consequence, several messages are suppressed (as 'undefined' and 'non stratifiable')

* Changes:
  - Inserted and deleted tuples are not shown
* Fixed bugs:
  - Complex left-hand-side relations in joins failed to be parsed
  - Conjunctive Prolog goals failed to be parsed (bug introduced in version 1.6.0)
  - Natural joins now return common attributes only once
  - Datalog rules involving expressions with (prefix) unary operators were incorrectly displayed as infix
  - Parsing of Datalog bodies failed is some situations were arithmetic operators were involved (as in **`/assert p(X) :- X is -(1)`**)
  - Parsing of projection lists failed in some situations where ***`table.*`*** was intermixed with references to single table attributes
  - Program transformation for obtaining safe rules yielded incorrect results in some cases
  - When dropping a view, its local view definitions (if any) were not dropped as well
  - Different views could define the same local view name

- **`/listing`** ***`Name`*** failed to list rules of different arities (bug introduced in version 1.6.0)

11.7 Version 1.6.0 (released on July, 28th, 2008)

- Enhancements:
  - SQL query language added to the system: DDL (Data Definition Language), DML (Data Manipulation Language), and DQL (Data Query Language)
  - Common database for different query languages. Relations defined via SQL or via Datalog can be interchangeably accessed by queries in any language
  - Pretty-print listings for Datalog programs and SQL statements
  - Processing of batch files via the new command **`/process`** ***`File`***
  - Display of 'File not found' errors
  - Lexicographically ordered listings
  - New commands:
    - **`/datalog`** Switches to Datalog interpreter
    - **`/datalog`** ***`Query`*** Executes a Datalog query
    - **`/prolog`** Switches to Prolog interpreter
    - **`/sql`** Switches to SQL interpreter
    - **`/sql`** ***`Query`*** Executes a SQL query
    - **`/dbschema`** Displays the database schema
    - **`/pretty_print`** Displays whether pretty print for listings is enabled
    - **`/pretty_print`** ***`Switch`*** Enables/Disables pretty print
    - **`/process`** ***`File`*** Processes the contents of File as if they were typed at the system prompt
- Changes:
  - Changed some output formatting for the debugger
  - Some tweaks on system messages, mainly referring to safety/ computability
  - Initial status: Program transformation and time display are disabled by default
  - System status is listed at start-up
  - Listings of Datalog rules are ordered
- Fixed bugs:
  - The debugger in SICStus-based releases yielded incorrect results
  - Asserting/Consulting some unsafe clauses without program transformation yielded failure, raising an input error /failing to consult

11.8 Version 1.5.0 (released on December, 30th, 2007)

- Enhancements:
  - A more fine-grained debugging as long as individual clauses can be inspected
  - Warning and error messages provided for:
    - Undefined predicates which are called by rules each time the database is changed
    - Unsafe rules
    - Execution exceptions known at compile-time
  - Exception messages provided for:
    - Execution exceptions unknown at compile-time
  - Rule transformation for allowing computation of safe rules which may raise run-time exceptions due to built-ins
  - Rejection of unsafe or uncomputable queries, views and autoviews
  - Catching of instantiation errors
  - Rule source annotated for debugging and informative errors, i.e., file and lines in the program (if consulted) or assertion time (if manually asserted)
  - Elapsed time display
  - New basic, simpler (although less efficient than the already implemented) algorithm for computing stratified negation, following [SD91]
  - Fresh variables are given new variable names instead of numbers
  - New commands:
    - **`/negation`** Displays the selected algorithm for solving negation
    - **`/negation`** ***`Algorithm`*** Sets the required Algorithm for solving negation (**`strata`** or **`et_not`**)
    - **`/timing`** Displays whether elapsed time display is enabled
    - **`/timing`** ***`Switch`*** Enables or disables elapsed time display (**`on`** or **`off`**, resp.)
    - **`/safe`** Displays whether program transformation is enabled
    - **`/safe`** ***`Switch`*** Enables or disables program transformation (**`on`** or **`off`**, resp.)
  - Changed commands:
    - **`/verbose`** Displays whether verbose output is enabled
    - **`/verbose`** ***`Switch`*** Enables or disables verbose output messages (**`on`** or **`off`**, resp.)
  - Deprecated commands:
    - **`/noverbose`**

- Slight modifications on existing commands:
  - **`/debug`** ***`Goal Level`*** The inspection level can be set with the second optional argument with **`p`** for predicate level and **`c`** for clause level
  - **`/status`** Now, it also displays the selected algorithm for negation and whether program transformation is enabled
  - **`/version`** For matching the 'standard' display
- New examples added to the directory **`examples`**
- The Prolog database corresponding to the Datalog loaded programs has been discarded, therefore using only one representation for them
- Revised and upgraded user's manual

- Changes:
  - Inequality built-ins cause an error and stops execution whenever they are computed with any non-ground argument (formerly, they silently failed)
- Fixed bugs:
  - The Linux version did not work. Now, it has been fixed and tested on Ubuntu 6.10, Kubuntu 7.04 (Feisty), and Mandriva Linux 2007 Spring
  - The parser did not detect that the argument of **`not`** could be a variable
  - Name clashes when loading programs and asserting rules are avoided

11.9 Version 1.4.0 (released on September, 2nd, 2007)

- Enhancements:
  - Arithmetic has been added. The infix builtin 'is' allows the evaluation of arithmetic expressions
  - Arithmetic operators:
    - `\` Bitwise negation
    - `-` Negative value of its single argument
    - `**` Power
    - `^` Synonym for power
    - `*` Multiplication
    - `/` Real division
    - `+` Addition
    - `-` Subtraction
    - `//` Integer quotient
    - **`rem`** Integer remainder
    - `\/` Bitwise disjunction between integers
    - `#` Bitwise exclusive or between integers
    - `/\` Bitwise conjunction between integers

    - **<<** Shift left the first argument the number of places indicated by the second one
    - **>>** Shift right the first argument the number of places indicated by the second one
- Arithmetic functions:
    - **sqrt** Square root
    - **log** Natural logarithm of its single argument
    - **ln** Synonym for **log/1**
    - **log** Logarithm of the second argument in the base of the first one
    - **sin** Sine
    - **cos** Cosine
    - **tan** Tangent
    - **cot** Cotangent
    - **asin** Arc sine
    - **acos** Arc cosine
    - **atan** Arc tangent
    - **acot** Arc cotangent
    - **abs** Absolute value
    - **float** Float value of its argument
    - **integer** Closest integer between 0 and its argument
    - **sign** Returns -1 if its argument is negative, 0 otherwise
    - **gcd** Greatest common divisor
    - **min** Least of two numbers
    - **max** Greatest of two numbers
    - **truncate** Integer part as a float
    - **float_integer_part(*X*)** Integer part as a float
    - **float_fractional_part(*X*)** Fractional part as a float
    - **round** Closest integer
    - **floor** Greatest integer less or equal to its argument
    - **ceiling** Least integer greater or equal to its argument
- Arithmetic constants:
    - **pi** Archimedes' constant
    - **e** Euler's number
- Scientific notation supported

  - Autoviews (automatic temporary views) for conjunctive queries on the fly
  - Parsing of programs, queries, and asserted rules
  - New command:
    - **`/status`** Displays the current status of the system
  - Output from the command **`/builtins`** has been rearranged
  - Upgraded input error message
  - Prolog goals (submitted via the command **`/prolog`**) can be conjunctive goals
  - Revised and upgraded user's manual
  - Revised and homogeneized input processing
  - Line comments (starting with `%`) are allowed as prompt inputs (useful for commenting lines in batch files)
  - File and path names enclosed between single quotes for error reporting in OS commands (therefore clarifying misusing of blanks)
- Fixed bugs:
  - Underscores in variables were incorrectly parsed
  - Asserted rules had missing program variable names
  - The output stream was not flushed when prompting user input in the debugger and when prompting new Prolog solutions using **`/prolog`**
  - File and directory names as numbers threw an exception in OS commands
  - Incorrect goal when abolishing no rules
  - Some commands did not admit blanks between arguments
  - Fixed some disarranged displays
  - Batch processing tried to open both **`.ini`** and **`.pl`** files
  - Dangling choice points in several places
  - Anonymous variables were incorrectly parsed
  - Debugging was not possible during batch processing

11.10 Version 1.3.0 (released on May, 2nd, 2007)

- Enhancements:
  - Declarative debugger
- Fixed bugs:
  - The output stream was not flushed before waiting the user input. This presented a connection problem with the configurable IDE ACIDE (See Contributions)
- Contributions:

  - ACIDE (A Configurable Development Environment). Authors: Diego Cardiel Freire, Juan José Ortiz Sánchez, and Delfín Rupérez Cañas, leaded by Fernando Sáenz. 3/2007. Description: This project is aimed to provide a multiplatform configurable integrated development environment which can be configured in order to be used with any development system such as interpreters, compilers and database systems. Features of this system include: project management, multifile editing, syntax colouring, and parsing on-the-fly (which informs of syntax errors when editing programs prior to the compilation). Status: alpha.
  - Emacs development environment. Author: Markus Triska. 22/2/2007. Description: Provides an integration of DES into Emacs. Once a Datalog file has been opened, you can consult it by pressing F1 and submit queries and commands from Emacs.

11.11 Version 1.2.0 (released on February, 9th, 2007)

- Enhancements:
  - Solving-by-stratum algorithm
  - Temporary views, which allow to write a temporary rule whose head is solved as a query
  - Program variable names are kept to allow more readable program listings
  - Syntax error reports when loading programs in standalone applications and SICStus source distribution
  - Handling and reporting of Prolog exceptions in standalone applications, SWI and SICStus source distribution
  - New commands:
    - **`/verbose`** (default option) for verbose output
    - **`/noverbose`** for abbreviated messages
    - **`/strata`** displays the current stratification
    - **`/pdg`** displays the current predicate dependency graph
    - **`/dir`** synonym of **`/ls`**
    - **`/log`** ***`FileName`*** sets the current log to ***`FileName`***
    - **`/log`** displays the current log file, if any
    - **`/nolog`** disables logging
    - **`/version`** for displaying the current system version
  - New uses for existing command: **`/abolish`** ***`Name`***, and **`/abolish`** ***`Name/Arity`***
  - Batch processing
  - Rearranged and revised help
  - Reworked command and query-related messages
  - Consulting/Reconsulting files avoids duplicates

  - Added examples
- Fixed bugs:
  - Loading an incorrect Datalog program exited standalone applications (**`des.exe`** and **`deswin.exe`** applications)
  - Evaluating Prolog goals via /prolog failed for programs containing negation
  - For several commands, blanks between a command and its arguments were not consumed but the first one
  - Non existent directory errors were not catched in command **`/ls`**

11.12 Version 1.1.2 (released on December, 20th, 2006)

- Enhancements:
  - New uses for existing commands: **`/list_et`** ***`Name`***, **`/listing`** ***`Name`***
- Fixed bugs:
  - The commands **`/list_et`** and **`/clear_et`** were not properly parsed
  - Infix operators allowed a variable argument

11.13 Version 1.1.1 (released on February, 21st, 2005)

- A new executable version for Linux
- Enhancements:
  - Atoms can contain blanks
- Fixed bugs:
  - When using /prolog, DES1.1 did not find predicates defined without facts.

11.14 Version 1.1 (released on March, 4th, 2004)

- Full recursion
- Memoization techniques
- Gathering of undefined facts under non stratified programs (incomplete algorithm)
- Several new commands:
  - **`/listing`** ***`Name/Arity`***. Lists Datalog rules matching the pattern
  - **`/retractall`** ***`Head`***. Deletes all Datalog rules matching head
  - **`/list_et`**. Lists contents of the extension table
  - **`/list_et`** ***`Name/Arity`***. Lists contents of the extension table matching the pattern
  - **`/clear_et`**. Clears the extension table
  - **`/builtins`**. Lists built-in operators
  - **`/cd`** ***`Path`***. Sets the current directory

  - **`/cd`**. Sets the current directory to the directory where DES was started from
  - **`/pwd`**. Displays the current directory
  - **`/ls`**. Displays the contents of the current directory
  - **`/ls`** ***`Path`***. Displays the contents of the given absolute or relative directory
  - **`[`*`FileNames`*`]`**. Consults a list of Datalog files abolishing previous rules
  - **`[+`*`FileNames`*`]`**. Consults a list of Datalog files keeping previous rules
  - **`/shell`** ***`Command`***. Submits a command to the operating system shell
- Cosmetic changes:
  - Commands start with a slash
  - Command arguments are no longer enclosed in brackets
  - Both commands and queries may end with a dot
- Fixed bugs:
  - Primitives fail adequately when they should do it, instead of exiting from the interpreter

11.15 Version 1.0 (released on December, 2003)

Version 1.0 of DES, the first public release of the system, featured:

- Naïve Datalog system intended to be complete w.r.t. relational algebra
- Limited support for recursion: Termination is not guaranteed for some recursive programs
- Basic Negation
- Built-in Operators
  - **`=`** Syntactic equality
  - **`\=`** Syntactic disequality
  - **`>`** Greater than
  - **`>=`** Greater than or equal to
  - **`<`** Less than
  - **`=<`** Less than or equal to
  - **`not(`*`Goal`*`)`** Negation
- Commands
  - **`consult(`*`File`*`)`** Loads a Datalog program abolishing current rules
  - **`reconsult(`*`File`*`)`** Loads a Datalog program keeping current rules
  - **`assert((`*`Head`*`:-`*`Body`*`))`** Asserts a new rule
  - **`retract((`*`Head`*`:-`*`Body`*`))`** Retracs a rule
  - **`abolish`** Abolishes the loaded program

- **`listing`** Lists the loaded rules.
- **`prolog(`*`Goal`*`)`** SLD execution of Goal.
- **`halt`** Quits the system
- **`help`** Displays the help

12. Acknowledgements

The author wishes to thank the Clip group for providing their free Ciao system, and in particular to F. Bueno for his help in porting DES to the Ciao system. Also thanks to J. Wielemaker and D. Diaz for providing their free Prolog systems. Also, thanks to all the people providing feedback, since they are guiding DES to suit more demanded requirements. Contributors are specially acknowledged: Markus Triska, for developing the Emacs IDE, and the students Diego Cardiel Freire, Juan José Ortiz Sánchez, Delfín Rupérez Cañas, and Miguel Martín, who developed and improved ACIDE. Thanks to Yolanda García and Rafael Caballero for making declarative debugging true for deductive databases. Both are also key authors in the inclusion of test case generation for views. In particular, Yolanda took the implementation effort supported by Rafael. Enrique Martín Martín fixed the Linux distribution. Finally, thanks to the projects TIN2005-09207-C03-03 (MERIT-FORMS), TIN2008-06622-C03-01 (FAST-STAMP), S-0505/TIC/0407 (PROMESAS-CAM), and UCM-BSCH-GR58/08-910502 (GPD-UCM) which supported this work.

Appendix A. GNU General Public License

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License.

Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

one line to give the program's name and an idea of what it does.

Copyright (C) *yyyy name of author*

This program is free software; you can redistribute it and/or

modify it under the terms of the GNU General Public License

as published by the Free Software Foundation; either version 2

of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

GNU General Public License for more details.

You should have received a copy of the GNU General Public License

along with this program; if not, write to the Free Software

Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) *year name of author*

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details

type `show w'. This is free software, and you are welcome

to redistribute it under certain conditions; type `show c'

for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright

interest in the program `Gnomovision'

(which makes passes at compilers) written

by James Hacker.

signature of Ty Coon, 1 April 1989

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

Bibliography

[AO08] P. Ammann and J. Offutt, "Introduction to Software Testing", Cambridge University Press, 2008.

[AOTWZ03] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo, "The deductive database system LDL++", TPLP, 3(1):61–94, 2003.

[BCC97] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López–García, and G. Puebla. "The Ciao Prolog system. Reference manual", School of Computer Science, Technical University of Madrid (UPM), 1997. http://www.clip.dia.fi.upm.es.

[BFG07] M. Becker, C. Fournet, and A. Gordon. Design and Semantics of a Decentralized Authorization Language. In CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium, pages 3–15, Washington, DC, USA, 2007. IEEE Computer Society.

[BPFWD94] M.L. Barja, N.W. Paton, A. Fernandes, M.H. Williams, A. Dinn, "An Effective Deductive Object–Oriented Database Through Language Integration", In Proc. of the 20th VLDB Conference, 1994.

[Caba05] Caballero, R., A declarative debugger of incorrect answers for constraint functional-logic programs, in: WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming (2005), pp. 8–13.

[CGL09] A. Calì, G. Gottlob, and T. Lukasiewicz. Datalog+-: a unified approach to ontologies and integrity constraints. In ICDT '09: Proceedings of the 12th International Conference on Database Theory, pages 14–30, New York, NY, USA, 2009. ACM.

[CGS06b] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez, "Towards a Set Oriented Calculus for Logic Programming", Programación y Lenguajes, P. Lucio y F. Orejas (editors), CIMNE, pp. 41-50, Barcelona, Spain, September, 2006.

[CGS07] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez, "A New Proposal for Debugging Datalog Programs", 16th International Workshop on Functional and (Constraint) Logic Programming, 2007.

[CGS08] R. Caballero, Y. García-Ruiz and F. Sáenz-Pérez, "A Theoretical Framework for the Declarative Debugging of Datalog Programs" In International Workshop on Semantics in Data and Knowledge Bases (SDKB 2008), LNCS 4925, pp. 143-159, Springer, 2008.

[CGS10a] R. Caballero, Y. García-Ruiz and F. Sáenz-Pérez, "Applying Constraint Logic Programming to SQL Test Case Generation", In 10th International Symposium on Functional and Logic Programming (FLOPS 2010), 2010.

[Chan78] C.L. Chang, "Deduce 2: Further Investigations of Deduction in Relational Databases", H. Gallaire and J. Minker (eds.), Logic and Databases, Plenum Press, 1978.

[Diaz] D. Diaz, `http://www.gnu.org/software/prolog`.

[Diet87] S.W. Dietrich, “Extension Tables: Memo Relations in Logic Programming”, IV IEEE Symposium on Logic Programming, 1987.

[DMP93] M. Derr, S. Morishita, and G. Phipps, “Design and Implementation of the Glue–NAIL Database System”, In Proc. of the ACM SIGMOD International Conference on Management of Data, pp. 147–167, 1993.

[FD92] C. Fan and S. W. Dietrich, "Extension Table Built-ins for Prolog", Software - Practice and Experience Vol. 22 (7), pp. 573-597, July 1992.

[FHH04] R. Fikes, P.J. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the Semantic Web. J. Web Sem., 2(1):19–29, 2004.

[FP96] Wolfgang Faber and Gerald Pfeifer. DLV homepage, since 1996. url **`http://www.dlvsystem.com/`**.

[GR68] C.C. Green and B. Raphael, “The Use of Theorem–Proving Techniques in Question–Answering Systems”, Proceedings of the 23rd ACM National Conference, Washington D.C., 1968.

[GTZ05] S. Greco, I. Trubitsyna, and E. Zumpano. NP Datalog: A Logic Language for NP Search and Optimization Queries. Database Engineering and Applications Symposium, International, 0:344–353, 2005.

[GUW02] H. Garcia-Molina, J. D. Ullman, J. Widom, “Database Systems: The Complete Book”, Prentice-Hall, 2002.

[IRIS2008] IRIS-Reasoner, **`http://iris-reasoner.org`**.

[JGJ+95] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, S. Eherer: ConceptBase - a deductive object base for meta data management. In Journal of Intelligent Information Systems, Special Issue on Advances in Deductive Object-Oriented Databases, Vol. 4, No. 2, 167-192, 1995. System available at: **`http://www-i5.informatik.rwth-aachen.de/CBdoc/`**

[KLW95] M. Kifer, G. Lausen, J. Wu, "Logical Foundations of Object Oriented and Frame Based Languages", Journal of the ACM, vol. 42, p. 741-843, 1995.

[KSSD94] W. Kiessling, H. Schmidt, W. Strauss, and G. Dünzinger, “DECLARE and SDS: Early Efforts to Commercialize Deductive Database Technology”, VLDB Journal, 3, pp. 211–243, 1994.

[KT81] C. Kellogg and L. Travis, “Reasoning with Data in a Deductively Augmented Data Management System”, H. Gallaire, J. Minker, and J. Nicolas (eds.), Advances in Data Base Theory, Volume 1, Plenum Press, 1981.

[Lloy87] J. Lloyd, “Foundations of Logic Programming”, Springer Verlag, 1987.

[Mink87] J. Minker, “Perspectives in Deductive Databases”, Technical Report CS–TR–1799, University of Maryland at College Park, March 1987.

[MN82] J. Minker and J.–M. Nicolas, “On Recursive Axioms in Deductive Databases, Information Systems”, 16(4):670–702, 1991.

[PDR91] G. Phipps, M. A. Derr, and K.A. Ross, “Glue–NAIL!: A Deductive Database System”. In Proc. of the ACM SIGMOD Conference on Management of Data, pp. 308–317, 1991.

[Robi65] J.A. Robinson, “A Machine–Oriented Logic Based on the Resolution Principle”, Journal of the ACM, 12:23–41, 1965.

[RS09] R. Ronen and O. Shmueli. Evaluating very large Datalog queries on social networks. In EDBT ’09: Proceedings of the 12th International Conference on Extending Database Technology, pages 577–587, New York, NY, USA, 2009. ACM.

[RSSS94] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The Coral deductive system. VLDB Journal, 3(2):161–210, 1994.

[RSSWF97] P. Rao, Konstantinos F. Sagonas, Terrance Swift, David Scott Warren, and Juliana Freire, “XSB: A System for Efficiently Computing WFS", Logic Programming and Non–monotonic Reasoning, 1997.

[RU95] R. Ramakrishnan and J.D Ullman, “A Survey of Research on Deductive Database Systems”, Journal of Logic Programming, 23(2): 125–149, 1995.

[SD91] C. Shih and S. W. Dietrich, "Extension Table Evaluation of Datalog Programs with Negation", Proceedings of the IEEE International Phoenix Conference on Computers and Communications, Scottsdale, AZ, March 1991, pp. 792-798.

[Shap83] Shapiro, E., “Algorithmic Program DeBugging”, ACM Distinguished Dissertation, MIT Press, 1983.

[SICStus] SICS, **`http://www.sics.se/sicstus`**.

[Silv07] Silva, J., A Comparative Study of Algorithmic Debugging Strategies, in: Proc. of International Symposium on Logic-based Program Synthesis and Transformation LOPSTR 2006, 2007, pp. 134–140.

[SRSS93] D. Srivastava, R. Ramakrishnan, S. Sudarshan, and P. Seshadri, “Coral++: Adding Object–Orientation to a Logic Database Language”, Proceedings of the International Conference on Very Large Databases, 1993.

[Tang99] Z. Tang, "Datalog++: An Object-Oriented Front-End For The Xsb Deductive Database Management System", http://citeseer.ist.psu.edu/tang99datalog.html.

[TS86] H. Tamaki and T. Sato, “OLD Resolution with Tabulation”, Proceedings of ICLP’86, Lecture Notes on Computer Science 225, Springer–Verlag, 1986.

[Ullm95] J.D. Ullman. Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies), Computer Science Press, 1995.

[VRK+91] J. Vaghani, K. Ramamohanarao, D.B. Kemp, Z. Somogyi, and P.J. Stuckey, “Design Overview of the Aditi Deductive Database System”, In Proc. of the 7th Intl. Conf. on Data Engineering, pp. 240–247, 1991.

[Wiele] J. Wielemaker, **`http://www.SWI Prolog.org`**.

[WL04] J. Whaley and M. Lam, Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In: Prog. Lang. Design and Impl., 2004.

[ZCF+97] C. Zaniolo, S. Ceri, C. Faloutsos, T.T. Snodgrass, V.S. Subrahmanian, and R. Zicari, "Advanced Database Systems", Morgan Kauffmann Publishers, 1997.

[ZF97] U. Zukowski and B. Freitag, “The Deductive Database System LOLA”, In: J. Dix and U. Furbach and A. Nerode (Eds.). Logic Programming and Nonmonotonic Reasoning. LNAI 1265, pp. 375–386. Springer, 1997.