



Datalog Educational System V1.5.0

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 12/30/2007



Copyright (C) 2004-2007 Fernando Sáenz

Every reader or user of this document acknowledges that is aware that no guarantee is given regarding its contents, on any account, and specifically concerning veracity, accuracy and fitness for any purpose.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation¹, 59 Temple Place – Suite 330, Boston, MA 02111, USA.

¹ <http://www.fsf.org/>



Contents

1. Introduction	5
1.1 Deductive Databases	5
1.2 Referring to DES	6
2. Installation	6
2.1 Downloading DES.....	6
2.1.1 Source Distribution	6
2.1.2 Executable Distribution	7
2.1.2.1 Windows	7
2.1.2.2 Linux	7
2.1.2.3 Unix	8
2.2 Installing and executing DES.....	8
2.2.1 MS Windows	8
2.2.1.1 Executable Distribution	8
2.2.1.2 Source Distribution.....	8
2.2.2 Unix/Linux	9
2.2.2.1 Executable Distribution	9
2.2.2.2 Source Distribution.....	9
2.2.3 Starting DES from a Prolog interpreter.....	9
3. Getting Started	9
3.1 Getting Help	12
4. System Description.....	12
4.1 Syntax	13
4.2 Rules.....	14
4.3 Programs	14
4.4 Queries	15
4.5 Temporary Views.....	15
4.6 Automatic Temporary Views	15
4.7 Batch Processing	16
4.8 Messages	16
4.9 Negation.....	17
5. Commands	19
5.1 Rule Database Commands	19
5.2 Debugging	21
5.3 Extension Table Commands	21
5.4 Operating System Commands	21
5.5 Log Commands	22
5.6 Informative Commands.....	22
5.7 Miscellanea	22
6. Built-ins.....	23
6.1 Comparison Operators	23
6.2 Arithmetic.....	23
6.2.1 Arithmetic Operators	24
6.2.2 Arithmetic Constants	25
6.2.3 Arithmetic Functions.....	25
6.3 Negation.....	26
7. Safety and Computability	27
8. Examples.....	29
8.1 Relational Operations (file relop.dl).....	29



8.2	Paths in a Graph (file paths.dl)	30
8.3	Family Tree (file family.dl).....	31
8.4	Basic Recursion Problem (file recursion.dl).....	32
8.5	Transitive Closure (file transclosure.dl).....	32
8.6	Mutual Recursion (file mutrecursion.dl).....	32
8.7	Farmer-Wolf-Goat-Cabbage Puzzle (file puzzle.dl).....	33
8.8	Paradoxes (file russell.dl).....	36
8.9	Parity (file parity.dl).....	39
8.10	Grammar (file grammar.dl)	40
8.11	Fibonacci (file fib.dl).....	40
8.12	Hanoi Towers (file hanoi.dl).....	41
8.13	Other Examples.....	42
9.	Declarative Debugger	42
10.	Notes about the Implementation of DES.....	43
10.1	Tabling	43
10.2	Finding Stable Models.....	44
10.3	Dealing with Negation	45
10.4	Porting to Unsupported Systems.....	45
10.5	Differences among Platforms.....	46
11.	Contributions.....	46
12.	Related Work	47
13.	Future Enhancements	48
14.	Release Notes History.....	48
15.	Acknowledgements.....	51
Appendix A. GNU General Public License.....		51
Bibliography		51

1. Introduction

The Datalog Educational System (DES) is a free, multiplatform, Prolog-based implementation of a basic deductive database with stratified negation and declarative debugging which uses Datalog as a query language. The system is implemented on top of Prolog and it can be used from a Prolog interpreter running both on Windows and Unix/Linux. Moreover, Windows, Linux and Unix executables are also provided.

We have developed it 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 as a query language. In addition, since it is implemented on top of Prolog and students are assumed to know Prolog programming, they can study its implementation. Other known systems are not fully suited due to the absence of some characteristics DES does offer for our educational purposes (see Section 11).

DES 1.5.0 is the current implementation, which enjoys full recursive evaluation with memoization techniques, arithmetic, stratified negation and a novel approach to declarative debugging. Our approach to debug a Datalog query relies on program semantics rather than on the computation mechanism. The debugging process starts when the user detects an unexpected answer. By asking questions about the intended semantics, the debugger looks for incorrect program relations. See Section 8.12 for further details.

In this release there are many new features and fixes (e.g., the Linux version is fixed) as listed in Section 14.

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 recently added recursion, it only allows 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 11 for references to other current deductive 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 a source distribution for several Prolog interpreters and operating systems, and executable distributions for Windows, Linux and Unix.

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, with minor changes to a small selected piece of code (found in the file **des1.pl**), it is likely to run on any other Prolog system and operating system it is installed, since the core (found in the file **des.pl**) was implemented following standard Prolog. (See Section 10.3 for porting to unsupported systems.) We have tested DES under several Prolog systems (Ciao Prolog 1.10p5, GNU Prolog 1.4.0, SICStus Prolog 3.11.0, and SWI–Prolog 5.6.27), and several operating systems (MS Windows 98 and later, SunOS, and Linux).

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

- **des.pl**. Contains the core of DES
- **desdebug.pl**. Contains the declarative debugger
- **des1.pl**. Contains particular code for the selected Prolog system

- **systems/{ciao,gnu,sicstus,swi}**. Contains the same two 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 [□](#)
- **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:

- **des.exe**. Console executable file
- **deswin.exe**. Windows executable file
- **des.pl**. Contains the core of DES
- **desdebug.pl**. Contains the declarative debugger
- **des1.pl**. Contains SICStus dependent code
- **systems/{ciao,gnu,sicstus,swi}**. Contains the same two previous files for all of the supported Prolog systems (these directories can be erased if desired, they are included only for reference)
- **main.sav**. Saved state of DES
- ***.dll**. DLL libraries for the runtime system
- **doc/manualDES<version>.pdf**. This manual
- **examples/*.dl** Example files which will be discussed in Section [□](#)
- **license/license** A verbatim copy of the GNU Public License for this distribution
- **sp311/*** Directory containing Prolog libraries

2.1.2.2 Linux

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

- **des**. Console executable file. It may require to set the execution permission
- **desdebug.pl**. Contains the declarative debugger
- **des.pl**. Contains the core of DES
- **des1.pl**. Contains SICStus dependent code
- **systems/{ciao,gnu,sicstus,swi}**. Contains the same two previous files for all of the supported Prolog systems (these directories can be erased if desired, they are included only for reference)
- **des.sav**. Saved state of DES
- **doc/manualDES<version>.pdf**. This manual

- **examples/*.dl** Example files which will be discussed in Section [□](#)
- **license/license** A verbatim copy of the GNU Public License for this distribution
- **sp-3.11.1/*** Directory containing Prolog libraries

2.1.2.3 Unix

From the same URL above, you can download a Unix executable distribution in a single archive file containing the same files found in the Linux distribution, targeted at the SunOS5.4 Sparc platform.

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 the way the system starts for commodity. 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.

2.2.1.2 Source Distribution

Perform the following steps:

1. Create a shortcut in the desktop for running the Prolog interpreter.
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]"`

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



2.2.2 Unix/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.

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.1.5.0           *
*
*
* Type "/help" for help about commands                       *
* Type "des." to continue if you get out of DES             *
*   from a Prolog interpreter                               *
*
*                               Fernando Sáenz (c) 2004-2007 *
*                               DISIA UCM                    *
*   Please send comments, questions, etc. to:             *
*                               fernan@sip.ucm.es          *
*                               Visit the Web site at:     *
*                               http://des.sourceforge.net/ *
*****
```

DES>

This last line (**DES>**) 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.** at the Prolog prompt to continue.

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 you use the following command in order to consult the Datalog program:²

```
DES> /consult Filename
```

Where **FileName** is the name of the file, as **relop.dl** (the default extension **.dl** can be omitted). If the file is located in the distribution directory, you can consult the file with:

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

or simply:

```
DES> /consult relop
```

Otherwise, when the file is located at another path, you can firstly change to the new path using the command **/cd Path**, where **Path** is the new directory (relative or absolute). Assuming that we are in the system directory and we have installed the distribution at **c:\des1.5.0**, we can submit the following inputs (see Section 7 for an explanation of the consulted program):

```
DES> /verbose off
```

```
Info: Abbreviated output.
```

```
DES> /consult relop
```

² See section □ for more details about commands.



Info: Consulting relop...

DES> /listing

```
a(a1).
a(a2).
a(a3).
b(b1).
b(b2).
b(a1).
c(a1,b2).
c(a1,a1).
c(a2,b2).
projection(X):-c(X,Y).
selection(X):-a(X),X=a2.
cartesian(X,Y):-a(X),b(Y).
join(X):-a(X),b(X).
union(X):-a(X).
union(X):-b(X).
difference(X):-a(X),not(b(X)).
```

DES> a(a3)

```
{
  a(a3)
}
```

DES> a(a5)

```
{
}
```

DES> /assert a(a4)

DES> a(X)

```
{
  a(a1),
  a(a2),
  a(a3),
  a(a4)
}
```

DES> /prolog a(X)

```
a(a1)
? (type ; for more solutions, <Intro> to continue) ;
a(a2)
? (type ; for more solutions, <Intro> to continue) ;
a(a3)
? (type ; for more solutions, <Intro> to continue) ;
a(a4)
? (type ; for more solutions, <Intro> to continue) ;
```

no

This last command allows to note the basic difference between Datalog and Prolog execution. The former gives the whole meaning of the relation **a** with the query **a(X)** at a time, whereas the latter searches solutions, one-by-one, that satisfy the goal **a(X)**. A meaning of a relation is the set of facts inferred both extensionally and intensionally from the program.

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:

```
DES> /list_et
```

```
Answers:
```

```
{
  a(a1),
  a(a2),
  a(a3),
  a(a4)
}
```

```
Calls:
```

```
{
  a(_12905)
}
```

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 10.1 and 10.2). Unless a temporary view (see Section 4.6) or the command **/clear_et** is submitted, the extension table keeps those results, otherwise it is cleared.

3.1 Getting Help

You can get useful information with the following commands:

- **/help**. Shows the list of available commands, which are explained in Section [4](#).
- **/builtins**. Shows the list of built-ins, which are explained in Section [6](#).

Also, visit the URL for last information:

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

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

```
fernans@sip.ucm.es
```

4. System Description

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 database user.

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

4.1 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 plus sign can be used as a prefix unary operator in arithmetical expressions (cf. Section 6.2.1) and also following the symbol **E** in scientific notation, as already seen.

- Alphanumeric constants. An alphanumeric constant is a sequence of characters written in any of the following forms:
 - Any sequence of alphanumeric characters (including the underscore **_**), starting with a lowercase letter
 - Any sequence of alphanumeric characters and blanks delimited by single quotes.

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

- Constants. A constant is either an alphanumeric constant or a number (integer or float).
- Variables. Variables are written with alphanumeric characters, and alternatively start with 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**, and **_var**, and **_**.

- Terms. Terms can be:
 - Noncompound. Variables or constants.

- Compound. As in Prolog, they have the form $\mathbf{t}(\mathbf{t1}, \dots, \mathbf{tn})$, where \mathbf{t} is a function symbol (functor), and \mathbf{ti} ($1 \leq i \leq n$) are terms.

Up to the current version, compound terms can only occur in arithmetic expressions, and their function symbols can be any of the built-in arithmetic operators and functions (cf. Section 6.2). These operators can be:

- Infix, as the addition (e.g., $\mathbf{1+2}$)
- Prefix, as bitwise negation (e.g., $\mathbf{\backslash 1}$)

Examples of terms are: $\mathbf{r(p)}$, and $\mathbf{p(x,y)}$, and $\mathbf{x > y}$.

- Atoms. An atom has the form $\mathbf{a(t1, \dots, tn)}$, where \mathbf{a} is a predicate (relation) symbol, and \mathbf{ti} ($0 \leq i \leq n$) are terms. If \mathbf{i} is $\mathbf{0}$, then the atom is simply written as \mathbf{a} .

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

There are several built-in predicates: \mathbf{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, $\mathbf{1 < 2}$ is a positive atom built from an infix built-in comparison operator (see Section 6.1).

Examples of atoms are: \mathbf{p} , $\mathbf{r(a,x)}$, $\mathbf{1 < 2}$, and $\mathbf{x is 1+2}$.

Note that $\mathbf{p(1+2)}$ and $\mathbf{p(t(a))}$ are not valid atoms.

- Literals. Literals can be:
 - Positive. An atom.
 - Negative. A negated atom of the form $\mathbf{not(a(t1, \dots, tn))}$, where $\mathbf{a(t1, \dots, tn)}$ 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).

Examples of literals are: \mathbf{p} , $\mathbf{r(a,x)}$, $\mathbf{not(q(x,b))}$, $\mathbf{1 < 2}$, and $\mathbf{x is 1+2}$.

Note that $\mathbf{not(p,q)}$ is not a valid literal.

- A literal can be a positive atom or a negative atom. They occur in rule bodies, queries, and view bodies.

4.2 Rules

Datalog rules have the form $\mathbf{head :- body}$, or simply \mathbf{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 6.

4.3 Programs

DES programs consist of a set of rules. Programs may contain remarks. A remark starts with the symbol $\mathbf{\%}$, and ends at the end of line.

4.4 Queries

A (positive) query is the name of a relation with as many arguments as the arity of the relation, following the syntax of atoms (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. A negative query is written as **not(PositiveQuery)**, i.e., a negated atom (a negative literal)

Queries are typed at the DES system prompt. The answer to a query is the set of facts 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 Sections 0 and 4.6). Built-in comparison operators (listed in Section 6.1) can be safely used in queries whenever their arguments are ground at evaluation time (excepting equality, which performs unification).

4.5 Temporary Views

Temporary views allows 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> 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> a(X) :- b(X)
```

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

4.6 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> a(X),b(Y)
```

```
Info: Processing:
```

```
  autoview(X,Y) :- a(X),b(Y)
{
  autoview(a1,a1),
  autoview(a1,b1),
  autoview(a1,b2),
  autoview(a2,a1),
  autoview(a2,b1),
  autoview(a2,b2),
  autoview(a3,a1),
  autoview(a3,b1),
  autoview(a3,b2)
}
```

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).
```

4.7 Batch Processing

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. To this end, prompt inputs starting with the symbol `%` are interpreted as comments. This way, the batch file `des.ini` may contain comments. Other way is useless although allowed.

4.8 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.

4.9 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 program³ 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 5.7). The second one works for stratified programs (possibly yielding non-termination for non-stratified programs) and is less efficient than the first one, which, in addition can even terminate for most non-stratifiable programs. Let's consider the following session:

```
DES> /assert p :- not(q)
Warning: Undefined predicate(s): [q/0]
DES> /assert q :- not(p)
Warning: Non stratifiable program.
DES> /negation strata
DES> p
Warning: Unable to ensure correctness for this query.
{
}
Info: Elapsed time: 0 ms.
DES> /negation et_not
Info: Algorithm 'et_not' selected for solving negation.
DES> p
... Non termination!
```

If you are interested in how programs with negation are solved for the algorithm **strata**, you can find useful the following commands (cf. Section 5.6):

```
DES> /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> /strata

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

³ Located at the **examples** distribution directory. Adapted from [RSSWF97].

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 $\text{predicate}/\text{arity}$. Next, arcs in the graph are shown in a list whose elements 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.

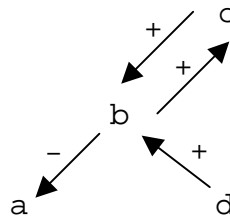


Figure 3. Predicate Dependency Graph for **negation.dl**

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> a
```

```
Info: Computing by stratum of [b].
{
  a
}
```

```
DES> 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.

5. 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 4.2 for information about DES queries.

5.1 Rule Database Commands

- `/consult FileName`

Loads 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> /consult examples/mutrecursion
```

which behaves the same as the following:

```
DES> /consult examples/mutrecursion.dl
```

```
DES> /consult ./examples/mutrecursion
```

```
DES> /consult c:/des1.5.0/examples/mutrecursion.dl
```

This last command assumes that the distribution directory is `c:/des1.5.0`.

Synonyms: /c.

- **/[FileNames]**

Loads 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> /[mutrecursion, family]
```

See also /consult Filename.

- **/reconsult FileName**

Loads 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.

- **/[+FileNames]**

Loads 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].

- **/assert Head:-Body**

Adds a Datalog rule. Rule order is irrelevant for Datalog computation. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/retract Head:-Body**

Deletes a Datalog rule. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/retractall Head**

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

- **/abolish**

Deletes all the loaded rules. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/abolish Name**

Deletes all the loaded rules for the predicates matching **Name**. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/abolish Name/Arity**

Deletes all the loaded rules for the predicate matching the pattern **Name/Arity**. The extension table is cleared, and the predicate dependency graph and strata are recomputed.

- **/listing**

Lists the loaded rules.

- **/listing Name**

Lists the loaded rules matching **Name**.

- **/listing Name/Arity**

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

5.2 Debugging

- **/debug *Goal***

Starts the debugger for the goal ***Goal*** at predicate level.

- **/debug *Goal Level***

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

5.3 Extension Table Commands

- **/list_et**

Lists the contents of the extension table in alphabetical order. First, answers are displayed, then calls.

- **/list_et *Name***

Lists the contents of the extension table matching ***Name***.

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

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

- **/clear_et**

Deletes the contents of the extension table.

5.4 Operating System Commands

- **/cd *Path***

Sets the current directory to ***Path***.

- **/cd**

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

- **/pwd**

Displays the absolute filename for the current directory.

- **/ls**

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

Synonym: /dir.

- **/ls *Path***

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

*Synonym: /dir *Path*.*

- **/shell *Command***

Submits ***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.

5.5 Log Commands

- **/log**
Displays the current log file, if any.
- **/log filename**
Sets the current log to filename.
- **/nolog**
Disables logging.

5.6 Informative Commands

- **/help**
Displays the help on commands.
Synonyms: /h.
- **/builtins**
Lists predefined operators.
- **/negation**
Displays the selected algorithm for solving negation (**strata** or **et_not**).
- **/safe**
Displays whether program transformation is enabled.
- **/timing**
Displays whether elapsed time display is enabled.
- **/timing Switch**
Enables or disables elapsed time display (**on** or **off**, resp.)
- **/verbose**
Displays whether verbose output is enabled.
- **/verbose Switch**
Enables or disables verbose output messages (**on** or **off**, resp.)
- **/strata**
Displays the stratification for the loaded program.
- **/strata**
Displays the current stratification as a list of pairs (PredName/Arity, Stratum).
- **/pdg**
Displays the current predicate dependency graph.
- **/version**
Displays the current system version.
- **/status**
Displays the current system status, i.e., verbose mode, the selected negation algorithm, logging, elapsed time display, program transformation, and system version.

5.7 Miscellanea

- **/prolog Goal**
Triggers Prolog's SLD resolution for the goal *Goal*.
- **/negation Algorithm**
Sets the required *Algorithm* for solving negation (**strata** or **et_not**).
- **/safe Switch**
Enables or disables program transformation (**on** or **off**, resp.)

- **/halt**

Quits the system.

Synonyms: /quit, /q, /exit, /e.

6. Built-ins

6.1 Comparison Operators

All comparison operators are infix. 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. These operators need ground arguments since they are not constraints, but test operators, and argument domains are infinite. Also, they always raise an exception if at least one of its arguments is a variable.

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

- **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**.

6.2 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> 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> 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> X is e

{
  2.718281828459045 is exp(1)
}
```

```
DES> 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 $\langle V, E \rangle$ such that V is the equivalent value to the evaluation of the arithmetical expression E .

6.2.1 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 .

- **$x \wedge y$** (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.
- **$x \text{ rem } y$** (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.
- **$x \setminus y$** (Bitwise disjunction) *ISO*
Bitwise disjunction of the integers **x** and **y** .
- **$x /\setminus y$** (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** .
- **$x \ll y$** (Shift left) *ISO*
 x shifted left **y** places.
- **$x \gg y$** (Shift right) *ISO*
 x shifted right **y** places.

6.2.2 Arithmetic Constants

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

6.2.3 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.

6.3 Negation

- **not(Relation)** (Stratified negation)
It stands for the complement of the relation **Relation** wrt. the meaning of the program (i.e., closed world assumption). See Sections 4.9 and 10.3.

7. Safety and Computability

Built-in predicates are appealing, but they come at a cost, already noticed in Section 6. 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 strong, 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.

8. 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 command `/verbose off` has been executed.

8.1 Relational Operations (file `relop.d1`)

This (classic) program is intended to show how to mimic the basic relational operations with Datalog rules. It contains three relations (**a**, **b**, and **c**), which are used as arguments of relational operations.

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

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

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

% Relational 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
join(X) :- a(X), b(X).

% a U b
union(X) :- a(X).
union(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> projection(X)
```

```
{
  projection(a1),
  projection(a2)
}
```

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 4.6 returns:

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

```
{
  a(a1),
  a(a2),
  a(a3),
  a(b1),
  a(b2)
}
```

8.2 Paths in a Graph (file `paths.d1`)

This program⁴ 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.

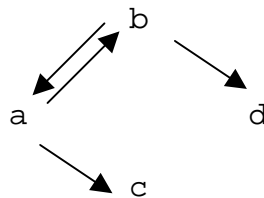


Figure 1. Paths in a Graph

```
% 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),

```

⁴ Adapted from [TS86].

```
path(a,b),
path(a,c),
path(a,d),
path(b,a),
path(b,b),
path(b,c),
path(b,d)
}
```

8.3 Family Tree (file family.d1)

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`).

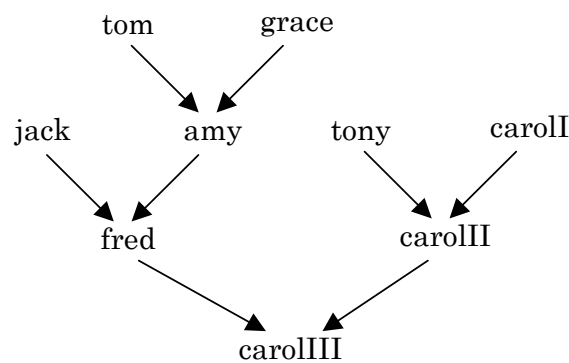


Figure 2. Family Tree

```
father(tom,amy).
father(jack,fred).
father(tony,carolII).
father(fred,carolIII).
mother(graceI,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)
}
```

Solving the view:

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

yields the following answer, computing the set of known sons:

```
{
  son(amy,tom,graceI),
  son(carolIII,tony,carolI),
  son(carolIII,fred,carolIII),
  son(fred,jack,amy)
}
```

8.4 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)`.)

8.5 Transitive Closure (file `tranclosure.dl`)

With this example, we show a possible use of mutual recursion by means of a program that defines the transitive closure of the relations `p` and `q`.⁵

```
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.

8.6 Mutual Recursion (file `mutrecursion.dl`)

The following program shows a basic example about mutual recursion:

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

⁵ Taken from [Diet87].

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

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

```
{
  p(a) ,
  p(b) ,
  p(c) ,
  p(d)
}
```

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.

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

This example⁶ 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) ,
  state(X1,Y,U,X1) .
% Farmer goes by himself
state(X,Y,U,V) :-
```

⁶ Adapted from [Diet87].

```
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)
}
```

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)
}
```

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:⁷

```
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
```

Now, let's turn our attention to the query `state(X,Y,U,V)`. If we empty the extension table with `/clear_et` or we submit this query from an empty extension table (as when we consult a program), we may expect to get all the possible states as before, but we only get:

```
{
  state(n,n,n,n)
}
```

However, this is a reasonable answer given the program above. Note that, in contrast to previous examples, we have a non ground fact:

```
% Farmer is with Goat
safe(X,Y,X,V) .
```

What is the meaning of this rule? The relation `safe/4` is true whenever the first and third arguments are the same, whatever the arguments are. This means that we can find an infinite ground set representing the meaning of `safe/4`. The answer set for this relation is represented with ground and non ground facts, as follows:

```
{
  safe(_8432,_8431,_8432,_8433) ,
  safe(n,n,s,n) ,
  safe(s,s,n,s)
}
```

The meaning of this relation is not completely defined with ground facts, which are needed to also complete the meaning of `state/4`. If we want to get a finite meaning for relations we are ought to write programs with only ground facts. Therefore, it turns out to be reasonable for this example to restrict the relation `safe/4` to only the possible values its arguments can take; in other words, to define finite domains (types) for them. Incidentally, the problem we have met is that of an unsafe Datalog rule in the nonground fact (cf. Section 7). When loading this program, a warning notices that this rule is actually unsafe.

So, we rewrite this relation as:

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

⁷ Remember that the system returns *all* of the possible solutions.

```
safe(X,X,X1,X) :- opp(X,X1) .
```

where the new relation **shore/1** is intended to represent the two possible values for an argument of type **shore**.

```
% Possible shores (n/s)  
shore(n) .  
shore(s) .
```

Now, all of the relation arguments have finite domains provided they are implemented with safe rules. For instance:

```
% Farmer takes Wolf  
state(X,X,U,V) :-  
  safe(X,X,U,V) ,  
  opp(X,X1) ,  
  state(X1,X1,U,V) .
```

The first two arguments have known domains because the goal **opp(X,X1)**, and also the next two arguments because of the goal **safe(X,X,U,V)**.

Back to the meaning of **safe/4**, now we get, as expected:

```
{  
  safe(n,n,n,n) ,  
  safe(n,n,n,s) ,  
  safe(n,n,s,n) ,  
  safe(n,s,n,n) ,  
  safe(n,s,n,s) ,  
  safe(s,n,s,n) ,  
  safe(s,n,s,s) ,  
  safe(s,s,n,s) ,  
  safe(s,s,s,n) ,  
  safe(s,s,s,s)  
}
```

That is, we get the intended intensional meaning in the former answer set as extensional information.

The file **typedpuzzle.dl** contains the rewritten program which yields the intended complete meaning to the query **state(X,Y,U,V)**.

8.8 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> shaves (X,Y)
```

```
Info: Computing strata...
```

```
Warning: Unable to ensure correctness for this query.
```

```
{
  shaves (barber,mayor)
}
```

```
Undefined:
```

```
{
  shaves (barber,barber)
}
```

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

```
{
  man (barber) ,
  man (mayor) ,
  not (shaves (mayor,mayor)) ,
  shaves (barber,mayor)
}
```

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:

```
Info: Computing strata...
```

```
Warning: Unable to ensure correctness for this query.
```

```
{
  shaved (barber) ,
  shaved (mayor)
}
```

```
Undefined:
```

```
{
  shaves (barber,barber)
}
```

That is, the system is unable to prove that `shaved(barber)` is undefined (it proves that the fact is positive but it is unable to prove that the fact is also negative).

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

```
DES> /pdg
```

```
Nodes: [man/1,shaved/1,shaves/2]
```

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

```
DES> /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.)

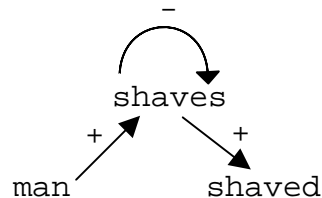


Figure 4. Predicate Dependency Graph for **russell.dl**

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> man(X)
```

```
Info: Computing strata...
```

```
Info: Stratifiable subprogram found for the given query.
```

```
{
  man(barber) ,
  man(mayor)
}
```

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 (cfr. Sections 8.9 and 8.10).

Stratification may be needed for programs without negation as long as a temporary view contains a negated goal. Consider the first view in Section 4.6 under the program of Section 7 (rules in the program with negation are not present in the subgraph for the query **d(X)**):

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

```
Info: Computing predicate dependency graph...
```

```
Info: Computing strata...
```

```
Info: Computing by stratum of [b(_9174)].
```

```
{
  d(a2) ,
  d(a3)
}
```

In this view, the query **d(X)** is solved with a solve-by-stratum algorithm, described in Section 10.3. In this case, this means that the goal **b(_9174)** 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 $p:-\text{not}(p)$ can be found in the file `paradox.dl`, whose model is undefined as you can test with the query `p`.

8.9 Parity (file `parity.dl`)

This example program⁸ 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 8.12 to show how it is caught by the declarative debugger.

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

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

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

⁸ Adapted from [ZCF+97].

```
br(a).  
br(b).
```

8.10 Grammar (file `grammar.d1`)

Parsers can also be coded as Datalog programs. In this example⁹, 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).  
DES> a(1,6)  
{  
  a(1,6)  
}  
Info: Elapsed time: 16 ms.
```

8.11 Fibonacci (file `fib.d1`)

The all-time classics Fibonacci program¹⁰ 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> fib(1000,F)  
{
```

⁹ Taken from [FD92].

¹⁰ Taken from [FD92].



```
fib(1000,7033036771142281582183525487718354977018126983635873274
2604905087154537118196933579742249494562611733487750449241765991
0881863632654502236471060120533741212738673391111981393731255987
67690091902245245323403501)
}
```

Info: Elapsed time: 28719 ms.

8.12 Hanoi Towers (file hanoi.d1)

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> hanoi(10,a,b,c)
{
    hanoi(10,a,b,c)
}
Info: Elapsed time: 16 ms.
```

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> /list_et
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)
}
```

8.13 Other Examples

The files `parity.dl` (computing the even/odd cardinality of a relation), `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).

9. 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 has 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. Therefore, the user can start a typical debugging session, as follows:

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

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

10. 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 and undefined (although incomplete) information. 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.

10.1 Tabling¹¹

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`

¹¹ For a complementary understanding of this section, the reader is advised to read [Diet87].

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 subsumes¹² 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, cfr. Section 5.2). 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 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.¹³ 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.

10.2 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,

¹² 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**).

¹³ The contents of the extension table in this case should be restored instead of being cleared; left for further improvements.

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 $s^k(\mathbf{z})$). 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 $\mathbf{p} :- \mathbf{p}$.

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 8.8), the algorithm for determining undefinedness is incomplete.

10.3 Dealing with Negation

Each time a program is consulted or modified (i.e., via submitting a temporary view or changing the database), a predicate dependency graph is built [ZCF+97]. This graph shows the dependencies, through positive and negative literals, among predicates in the program. It 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 wrt. a naïve bottom-up implementation.

10.4 Porting to Unsupported Systems

DES is implemented with three Prolog files: `des.pl`, `desdebug.pl`, and `des1.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 debugger code, also following ISO. 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.)

10.5 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.

11. Contributions

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

- ACIDE (A Configurable Development Environment).
Authors: Diego Cardiel Freire, Juan José Ortiz Sánchez, and Delfín Rupérez Cañas, led by Fernando Sáenz.
Date: 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).
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.

12. 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 to students. Nevertheless, we can comment some representative 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.

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 efficient represent and operate on extremely large relations.

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 employing 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].

The only commercial oriented deductive database system has been the Smart Data System (SDS) and its declarative query language Declarative Reasoning (DECLARE) [KSSD94], with support for stratified negation and sets.

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.

13. Future Enhancements

The following list (in order of importance) suggests some points to address for enhancing DES:

- Aggregated functions (**count**, **sum**, **avg**, ...) and multiset handling.
- Database integration (relational, deductive).
- Hypothetical queries.
- Disjunctive heads.
- Information about cycles involving negation in the loaded program.
- Complete algorithm for finding undefined information.
- Constraint solver for sound computation of primitives.
- Precise error reporting for 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.

14. Release Notes History

This section lists release notes of all software releases in reverse chronological order:

Version 1.5.0 of DES adds to previous version (1.4.0):

- 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
-

Version 1.4.0 of DES adds to previous version (1.3.0):

- 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
-

Version 1.3.0 of DES adds to previous version (1.2.0):

- 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, leded 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.

Version 1.2.0 of DES adds to previous version (1.1.2):

- 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 caught in command `/ls`
-

Version 1.1.2 of DES adds to previous version (1.1.1):

- 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
-

Version 1.1.1 of DES adds to previous version (1.1):

- 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.
-

Version 1.1 of DES adds to previous version (1.0):

- 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
-

Version 1.0 of DES, the first 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))` Retracts 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

15. 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, and Delfín Rupérez Cañas, who developed ACIDE. Thanks to Yolanda García and Rafael Caballero Roldán for making declarative debugging true for deductive databases. In particular, Yolanda took the implementation effort supported by Rafael expertisement. Enrique Martín Martín fixed the Linux distribution. Finally, thanks to the projects TIN2005-09207-C03-03, and S-0505/TIC0407 which supported this work.

Appendix A. GNU General Public License

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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

- [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>.
- [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.
- [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.
- [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.
- [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.
- [JGJ+95] M. Jarke, R. Gellersdörfer, 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.
- [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.