DELORES Programmer's Guide

Tristan Miller
German Research Center for Artificial Intelligence
Erwin-Schrödinger-Straße 57
67663 Kaiserslautern
Tristan.Miller@dfki.de

13 December 2003

Contents

1 Introduction

1.1 About this document

DELORES is a defeasible logic interpreter which works in interactive or batch mode. The following document is intended to assist future developers and maintainers of the DELORES project in comprehending the structure of the source code. It is not intended to cover in great detail the inner workings of the individual functions, and as such should not be considered a replacement for the program's internal documentation.

1.2 Program history

The original DELORES was implemented from 13 October 1999 to 13 January 2000 at Griffith Uni-

versity by Tristan Miller, based on instructions and papers provided by Michael Maher. The system was revised for public release in December 2003 by Tristan Miller, then at the German Research Center for Artificial Intelligence (DFKI GmbH) in Kaiserslautern.

2 Overview of project files

Files in the DELORES distribution are organized into a number of subdirectories. `doc` contains the program documentation, `src` contains the program source, `examples` contains sample theories for DELORES, and the root directory contains various files related to project management.

2.1 Project management

The DELORES project is distributed using the GNU Build System; the distributed files therefore adhere more or less to the GNU coding standards.

2.2 Metadata files

The files `AUTHORS`, `ChangeLog`, `COPYING`, `INSTALL`, `README`, `NEWS`, and `THANKS` are all required by the GNU Build System. They are self-explanatory and should be manually updated when any major changes are made. As DELORES is maintained in a CVS repository, it is easiest to update the `ChangeLog` file with the GNU `rcs2log` program.

The `BUGS` and `TODO` files are not mandated by the GNU standards but are included for obvious reasons.

2.3 Build files

The human-created files for use with Autoconf and Automake are `configure.ac` and `Automake.am`.[1] Refer to the GNU Autoconf and Automake manuals for how to edit these files.

Every other file included in the distribution and not mentioned in this manual is automatically produced by the GNU Build System.

2.4 Documentation

2.4.1 `delores.1`

This is the Unix man page for DELORES, written with `troff` macros.

[1] Each subdirectory in the distribution may have its own `Automake.am`.

2.4.2 `delores.tex`

This is the LaTeX source for this document.

2.5 Cross-platform compatibility

2.5.1 `config.h`

This file is automatically generated by the Autoconf `configure` script distributed with DELORES; it `#define`s a number of C preprocessor macros informing DELORES which C features are available on the host system. For those systems which for whatever reason cannot run the `configure` script, `config.h` can be manually produced using `config.h.in` as a template.

2.5.2 `dl_stdbool.h`

DELORES makes use of C99's `bool` data type. This data type is not yet available on all C compilers, so `dl_stdbool.h` is provided as a wrapper to the official library header `<stdbool.h>`. If the user's compiler does not have the `bool` data type, then `dl_stdbool.h` will define it. Any source file using boolean variables must `#include "dl_stdbool.h"` instead of `<stdbool.h>`.

2.5.3 `dl_stdint.h`

DELORES makes extensive use of various preprocessor macros and data types defined in the `<stdint.h>` header, which is new in C99. Since C99 is not yet widely supported, DELORES's `dl_stdint.h` wrapper should be used in place of `<stdint.h>`. Files making use of the macros `PRIuMAX`, `SCNuMAX`, `SIZE_MAX`; the data type `uintmax_t`; or the function `strtoumax()` `#include` this wrapper.

2.5.4 `dl_strdup.h`, `dl_strdup.c`

The `strdup()` function is a non-standard extension to the C language provided by many compilers. For those compilers which do not include this function, we define our own version here. Any file which uses the `strdup()` function should therefore be sure to `#include "dl_strdup.h"`

2.6 Memory management

The following files were incorporated into DELORES over fears that the native memory management provided by the development environment and/or operating system (*i.e.*, `malloc()` and friends) would be too slow. With this code, developers now have the option of compiling DELORES with the default memory management

scheme, or with BGET, a public domain memory allocation package by John Walker. The original BGET source code has been modified somewhat for smoother integration with DELORES, to eliminate some unnecessary code and internal documentation, and to correct `brel()`'s incompatibility with `free()`.

2.6.1 `dl_malloc.h`

In this header is defined the macro `DL_USE_BGET`, whose presence indicates to use the BGET memory allocation package instead of the environment's native `malloc()` and `free()`. Note that all memory allocation in DELORES should be performed using `balloc()` and `bfree()`, which are wrappers for the BGET or native `malloc()` and `free()`. If BGET support is enabled, this header also gives the prototype for `bufferPoolInitialize()`, which must be called before the first use of `balloc()`. The `dl_malloc.h` file should be included in any file making use of dynamic memory allocation.

2.6.2 `dl_malloc.c`

This file defines the function `balloc()`, which is essentially a wrapper for either `malloc()` or BGET's `bget()`, depending on whether or not the `DL_USE_BGET` macro is defined. `balloc()` has the added functionality of checking the return value of the memory allocation function, and aborting execution if no further memory is available. If `DL_USE_BGET` is defined, this file also includes code for the `bufferPoolInitialize()` function and the two static functions it calls.

2.6.3 `bget.h`

This is the header file for the BGET memory allocator. It includes prototypes for functions (which are not used outside `bget.c` and `dl_malloc.*`) and some useful macros and a typedef. (For some reason, the author of BGET elected to use `long` instead of the standard `size_t` for his memory buffers; accordingly, BGET is not completely compatible with `malloc()` and friends.[2]) There is no need to include this file as it is already included in the `dl_malloc` files.

2.6.4 `bget.c`

This file contains the code for BGET, John Walker's memory allocation package which serves as a faster, mostly-compatible replacement for `malloc()`. Some of the code in `bget.c`, and its associated header `bget.h`, has been modified by the DELORES author.

[2] This may explain why DELORES compiled with BGET periodically crashes on some systems.

2.7 Hash table abstract data type

The hash table data type comes in a separate module, OHASH, comprising the two files `ohash.h` and `ohash.c`. OHASH is a public domain hash table module by Tristan Miller, based on public domain code by Jerry Coffin. OHASH comes with extensive internal documentation, and the reader is encouraged to consult it.

2.7.1 `ohash.h`

This header gives typedefs and function prototypes for the hash table ADT. Any DELORES source file making use of hash tables needs to include this file.

2.7.2 `ohash.c`

This file includes the source code for the functions to add to, delete from, and search hash tables.

2.8 Timer abstract data type

The timer ADTs provide an interface for creating timers to measure CPU time and, if possible, real time. The source code is extremely simple, usage of this module should be obvious from a cursory examination of the header file.

2.8.1 `timer.h`

This header gives the typedefs and function prototypes dealing with real-time and CPU-time timers. Any file making use of the `cpuTimer` and `realTimer` data types and associated functions should include this file.

2.8.2 `timer.c`

This file provides the definitions for functions which initialize, reset, read, and destroy timers. Some of the code is platform-specific; its inclusion is determined by preprocessor directives in timer.h. If the platform does not support real-time timers with subsecond resolution, then reading a `realTimer` will always return 0.

2.9 Command-line argument processing

Because of the inflexibility and nonportability of `getopt()`, the command line argument processor has been manually implemented as a Flex lexer.

2.9.1 cmd_line_args.h

This header file typedefs a struct, `cmdargs_t`, which is used to pass the parsed command line arguments from the lexer. It also contains the function prototype for `getCmdLineArgs()`, the function which does the actual processing.

2.9.2 cmd_line_args.l

This file contains the Flex lexer for processing the command-line arguments. It is responsible for determining whether the requested hash table sizes, *etc.*, are within the acceptable limits, and also for finding the closest prime to the arguments for the `-a` and `-r` options.

2.9.3 primes.txt

This text file lists the first 100 008 primes,[3] plus a few more. It is used to search for the smallest prime greater than or equal to the number given with the `-a` and `-r` command-line options.

2.9.4 cmd_line_args.c

This is the C source code produced when Flex is invoked on `cmd_line_args.l`. It is an intermediate file and, provided one has Flex, may be safely deleted after project compilation.

2.10 Main interpreter

2.10.1 dl.h

This header file contains data types, global variables, function prototypes, and macros shared among the lexer, parser, and interpreter.

2.10.2 lexer.l

This is the interpreter's lexer; its primary job is to recognize tokens (keywords, atoms, variables, and labels) and pass them onto the parser. It is also responsible for ignoring comments, and for processing `included` files. This file must be processed by Flex to yield a C source file, `lexer.c`. The lexer is invoked by the parser, `parser.y`.

2.10.3 parser.y

This file contains the interpreter's parser, which is processed by Bison to yield the C source files `parser.h` and `parser.c`. The parser is the main loop of the interpreter; it is responsible for recognizing the grammar of the defeasible logic language and deciding which functions to call.

2.10.4 dl.c

This file contains all the functions called by the parser to manipulate the program's data. It contains the core of the interpreter's functionality; there are functions to add and delete literals, rules, and atoms, as well as routines for executing the defeasible logic reasoning engines.

2.10.5 main.c

This file contains `main()` and is the point of entry to the interpreter. `main()` is responsible for calling the command line argument lexer and making the appropriate initializations based on its return value, and then invoking the parser. This file also contains the hash table variables, and the hashing functions used to initialize them. (The hashing functions are not included in the ohash module because it is meant to be an *abstract* data type; no one hashing function can work on all possible data so the user (in this case, DELORES) is expected to provide their own hashing functions.)

2.10.6 lexer.c

`lexer.c` is the intermediate source file produced when Flex is invoked on `lexer.l`. It may be safely deleted.

2.10.7 parser.h, parser.c

These are the intermediate source files produced when Bison is invoked on `parser.y`. Provided one has Bison to rebuild them, they may be safely deleted.

2.10.8 delores

This is the actual executable file of the DELORES interpreter.

3 Bugs and debugging

3.1 Known issues

The code for DELORES is believed to be bug-free, in the sense that it has not (yet) been made to crash or to produce unpredictable results. There is, however, one segment of code which is perhaps not as intelligent as it could be made. Namely, the lexer code for the `include` directive is written such that it interprets as a filename everything between

[3] Source: `http://www.utm.edu/research/primes/lists/small/`

the first left parenthesis after the `include` keyword and the last right parenthesis and period before a newline. Needless to say, this is problematic if one's program contains a line such as the following:

```
include(program1.dl). include(program2.dl).
```

In this case, DELORES aborts with an error indicating that the file `"program1.dl). include(program2.dl"` cannot be found. To rectify this behaviour is not a trivial feat, which is why it was not done by the original author in the first place. Until such time as it is fixed, users are cautioned to place their `include` directives on lines separate from any other statements.

3.2 Debugging tools

There are a number of debugging facilities built into DELORES and its associated modules. Typically, these facilities take the form of conditional compilation directives which, when certain macros are defined, compile in extra debugging information which will be printed to standard error when the interpreter is run.

3.2.1 Lexer

The lexer (that part of the interpreter which recognizes individual tokens) can be made to print out information on tokens as it recognizes them by defining the `DL_LEXER_DEBUG` macro. This macro can be found, commented out, in `dl.h`.

3.2.2 Parser

The parser (that part of the interpreter which recognizes the language grammar) can be made to print out information on syntactic constructs as it recognizes them by defining the `DL_PARSER_DEBUG` macro. This macro can be found, commented out, in `dl.h`.

3.2.3 Other functions

To enable other helpful debugging messages in the interpreter, there is the `DL_DEBUG` macro, which may be enabled in `dl.h`. When `DL_DEBUG` is enabled, the user also has access to the `?` command which, when appended to an atom name, will print out detailed information on that atom. (The information given is much more in-depth than that provided by the `print` directive.) The `dl.h` header also includes the `DL_PROFILE` macro which, when enabled, will time the execution of DELORES. (Obviously, the results will only be meaningful in batch mode.)

3.2.4 BGET

The only debugging facility in BGET is the use of `assert()` macros, which is turned on and off with the `NDEBUG` macro.

3.2.5 OHASH

When compiled, OHASH looks for the presence of a `HASH_PROFILE` macro (commented out in `ohash.h`). If it exists, it will compile in a special variable, `collisions`, in the hash table data structure. As its name suggests, `collisions` holds a running total of the number of insertion collisions for the associated hash table. Code in `main.c` will test for the presence of the `HASH_PROFILE` macro and, if it exists, will print out the number of atom table and rule table collisions before the interpreter exits. This information can be used to test the suitability of the hashing function.

4 Data structures

All custom data types in DELORES (with the exception of hash tables and timers) are defined in `dl.h`. (The reader is encouraged to consult this file directly, as many of the concepts are mutually referential and thus difficult to explain in prose.) There are two fundamental data types, atoms and rules, and two aggregate data types, rule lists and literals.

4.1 Atoms

Atoms are those unique, definite constants which correspond to proper nouns in natural language. Atoms, which are C variables of type `Atom`, are uniquely identified by a name (any sequence of alphanumeric characters, including the underscore, which begins with a lowercase letter) and are stored in a hash table called `atomTable`. The `Atom` data structure contains a pointer (`char *id`) to the atom's name; a series of boolean variables (`bool plus_delta`, `bool minus_sigma_neg`, *etc.*) giving information on what has been proved about the atom and its negation; two pointers to a list of rules (`RuleList *rule_heads` and `RuleList *rule_heads_neg`) whose head is the atom or its negation; and finally pointers (`Literal *strict_occ`, `Literal *defeater_occ_neg`, *etc.*) to lists of occurrences of that atom (or its negation) in strict, defeasible, and defeater rule bodies.

4.2 Rules

Every rule has a unique identifying label (`char *id`) comprised of alphanumeric characters and digits; the label must begin with a letter. Labels generated by the interpreter may also contain slashes. Rules are C variables of type `Rule`, and are stored in a hash table called `ruleTable`. A rule has three basic parts: the head, the arrow type, and the body. The head is comprised of a pointer to an atom (`Atom *head`), and a boolean flag (`bool neg`) indicating whether the atom is to be negated. The arrow type (`int arrow_type`) is one of `SARROW`, `DARROW`, or `DEFARROW`; these are preprocessor macros defined in `parser.h`. Finally, the body is a pointer to a list of literals. Also stored with the rule is a number indicating the order in which it was created; this is used by the `listing` directive to sort the rules.

4.3 Rule lists

As the name implies, a rule list (variable of type `RuleList`) is simply a doubly-linked list of pointers to rules. Their main use is in the `Atom` data structure; every atom `a` points to a list of rules whose head is `a`.

4.4 Literals

Literals (variables of type `Literal`) are, basically speaking, lists of atoms. They are used primarily as rule bodies. A literal contains a pointer (`Atom *atom`) to an atom, a boolean flag (`bool neg`) indicating whether the atom is negated, and pointers (`Literal *next` and `Literal *prev`) to the previous and next literals in the list. The literal will also contain a pointer (`Rule *rule`) to the rule whose body it is in, if applicable. For example, take the following rule:

```
r: h <= b1, neg b2, b3.
```

This rule's body is essentially a pointer to the literal, let's call it `L1`. Then `L1.rule` is a pointer to the rule r, `L1.atom` is a pointer to $b1$, `L1.neg` is false, `L1.prev` is `NULL`, and `L1.next` contains a pointer to the next literal in the body, say, `L2`. Then `L2.rule` is a pointer to r, `L2.atom` is a pointer to $b2$, `L2.neg` is true, and so on.

Literals also contain two more pointers, which the author has labelled "up" and "down" more for ease of visualization than for accurate semantics. Whereas the "prev" and "next" pointers point to the corresponding left or right literal in the rule body as it would be written, the "up" and "down" pointers point to the previous and next literals in that literal's *equivalence class*. The equivalence class for a literal `L` is the set of all literals `E` for which `L.atom == E.atom` and `L.neg == E.neg` and `L.rule->arrow_type == E.rule->arrow_type` (where `L.rule` and `E.rule` are both non-`NULL`). It is to these equivalence classes of literals that the atoms' `strict_occ`, `strict_occ_neg`, *et al.* pointers to point. Here is another example; assume that these are the only two rules in the theory:

```
r: h <= neg b1, b2.
s: i <= neg b1.
```

For simplicity's sake, assume that `b1` and `b2` are the C variable names for the atoms $b1$ and $b2$, respectively. Further, assume that `L1` and `L2` are `Literal` variables for the first two body elements in r, and `L3` is a `Literal` variable representing the body of s. Finally, say `r` is the variable for the rule r, and likewise `s` for s. Then we have the following:

```
b1.strict_occ == NULL
b1.strict_occ_neg == NULL
b1.defeater_occ == NULL
b1.defeater_occ_neg == NULL
b1.defeasible_occ == NULL
b1.defeasible_occ_neg == &L1
b2.strict_occ == NULL
b2.strict_occ_neg == NULL
b2.defeater_occ == NULL
b2.defeater_occ_neg == NULL
b2.defeasible_occ == &L2
L1.atom == &b1
L1.neg == true
L1.rule == &r
L1.prev == NULL
L1.next == &L2
L1.up == NULL
L1.down == &L3
L2.atom == &b2
L2.neg == false
L2.rule == &r
L2.prev == &L1
L2.next == NULL
L2.up == NULL
L2.down == NULL
L3.atom == &b1
L3.neg == true
L3.rule == &s
L3.prev == NULL
L3.next == NULL
L3.up == &L1
L3.down == NULL
```

5 Future improvements

5.1 Project maintenance

- The external documentation could probably be improved, especially the above section on data structures, as it was produced under somewhat rushed conditions.

5.2 Interpreter

- Though the grammar is in place, the semantics for term lists, `faild`/`failD` declarations, and variables/grounding has yet to be implemented. See `lexer.l` and `parser.y` for what code needs to be added—sections are marked conspicuously with the following comments:

  ```
  /* *** NOT IMPLEMENTED YET *** */
  ```

- Both the syntax and semantics for `private` declarations has yet to be implemented.

- When an error occurs interpreting a deeply-nested `include` file, it would be helpful to track back through the `include` stack much in the same way that gcc does. This functionality would be built into `lexer.l`.

- The well-founded defeasible logic inference engine is not complete. Those sections which still need work are conspicuously marked as such in the program's internal documentation. Briefly, however, the only procedures which are not yet in place are the mechanisms for "remembering" the state of the rules and for returning the rules to that state. The former requires changes to `inferRevisedAlgorithm()`, and the latter changes to `resetRules()`, both in `dl.c`.

A Copyright