Boolean Expression Diagrams Library
version 2.5 (DRAFT)

Poul Frederick Williams

IT University in Copenhagen
Glentevej 67, DK-2400 Copenhagen NV, Denmark
E-mail: pfw@it-c.dk

October 11, 2000

Abstract

This paper describes the Boolean Expression Diagram library written in C by Henrik Reif Andersen, Henrik Hulgaard, and Poul Frederick Williams between 1996 and 2000. We describe the library from a user's point of view.

1 Introduction

This paper gives an overview of how to use the Boolean Expression Diagram (BED) library in C and C++ programs. We assume that the reader has a prior knowledge of the BED data structure.

2 Getting Started

You can obtain the newest BED package by writing to one the authors at {hra,henrik,pfw}@it-c.dk. The BED webpage is http://www.it-c.dk/research/bed/. The README file in the distribution describes how to compile the package.

Once you have successfully compiled the package, you will find the following libraries:

- lib/libutil.a
- lib/libbed.a
- lib/libshell.a

The first one contains utility functions for use in the other libraries. The second one contains the BED data structures and the algorithms for manipulating them. The third one contains functions for constructing a shell-interface to the BED package. Only the first two libraries are needed to use BEDs in your own programs.

To use BEDs in your own C or C++ program, you should do the following:

- Include bed.h.
- Start your BED session by calling bed_init.
- End your BED session by calling bed_done.
- Link your C program with libutil.a and libbed.a.
- Be sure that the compiler can find the include files which are located in libutil and libbed.

Variables of the type `bed_var` represent boolean variables. Variables of the type `bed_node` (`bed` in C++) represent vertices in the BED data structure. The constants `bed_false` and `bed_true` represent the two terminal vertices. Variables of the type `bed_op` represent boolean connectives.

The types `bed_var_list` and `bed_node_list` represent lists of `bed_var`s and `bed_node`s, respectively. Use the functions in `libutil/ilist.h` to manipulate them.

Return values of type `bed_node` are *only* valid until the next call into the BED package. Use `bed_ref` to preserve them across calls. Remember to free those nodes again using `bed_deref` when you no longer need them. If you use the C++ class `bed` you do not have to worry about reference counting the nodes. The class constructors and destructor handle it for you.

3 An Example

This section contains a very simple example of how to use the BED library in your own C program.

3.1 The `main.c` File

```
#include "bed.h"
#include "bedio.h"      /* get access to I/O functions */

void doit()
{
  /* Get 16 new boolean variables -- v is the first, v+1 the second, etc. */
  bed_var v = bed_new_variables( 16 );

  /* Create BEDs for the variables v and v+1 */
  bed_node n1 = bed_ref( bed_mk_var( v, bed_false, bed_true ) );
  bed_node n2 = bed_ref( bed_mk_var( v+1, bed_false, bed_true ) );

  /* Create a BED for the conjunction of n1 and n2 */
  bed_node n3 = bed_ref( bed_mk_op( BED_AND, n1, n2 ) );

  /* Transform n3 into a BDD using upall */
  bed_node n4 = bed_ref( bed_upall( n3 ) );

  /* Transform n3 into a BDD using upone_iter */
  bed_node n5 = bed_ref( bed_upone_iter( n3 ) );

  if ( n4 == n5 )
    printf( "n4 and n5 are identical\n" );

  if ( bed_is_bdd( n5 ) )
    printf( "n5 is a BDD\n" );

  /* Clean up */
  bed_deref( n1 );
  bed_deref( n2 );
  bed_deref( n3 );
  bed_deref( n4 );
  bed_deref( n5 );
}


int main()
```

```
{
  /* Initialize the BED library: 10 kb for the data structure and 5 kb for caches */
  bed_init( 10*1024, 5*1024 );

  doit();

  /* Close down the BED library */
  bed_done();

  return 0;
}
```

3.2 The Makefile

```
CC              = gcc

SRC             = main.c
OBJ             = $(subst .c,.o,$(SRC))

TARGET          = main

BEDHOME         = $(HOME)/bed/src       # change to match your system

INCLUDE         = -I$(BEDHOME)/libbed -I$(BEDHOME)/libutil
LIBS            = -L$(BEDHOME)/libbed -L$(BEDHOME)/libutil -lbed -lutil

CFLAGS          = $(INCLUDE)
LDFLAGS         = $(LIBS) $(INCLUDE)


%.o :           %.c
                $(CC) -c $(CFLAGS) $< -o $@

all:            $(OBJ)
                $(CC) $(CFLAGS) $(OBJ) $(LDFLAGS) -o $(TARGET)

clean:
                rm -f $(OBJ)
```

4 The bed.h Header File

```
/*
 *  Copyright (c) 1996 - 2000 Technical University of Denmark
 *  Copyright (c) 1999 - 2000 IT University of Copenhagen
 *
 * by Poul Frederick Williams, Henrik Hulgaard, Henrik Reif Andersen
 * IT University of Copenhagen
 * Glentevej 67
 * DK-2400 Copenhagen NV, Denmark
 *
 * e-mail: {pfw,henrik,hra}@it-c.dk
 */
```

```
#ifndef bed_h_
#define bed_h_

#ifdef __cplusplus
#define CPLUSPLUS
#endif

#ifdef CPLUSPLUS
extern "C" {
#endif

#include "bool.h"
#include "ilist.h"

#ifdef CPLUSPLUS
}
#endif


/*=== BED types ================================================*/


typedef unsigned int    bed_var;
typedef unsigned int    bed_node;

#ifndef CPLUSPLUS
typedef bed_node bed;
#endif /* CPLUSPLUS */

typedef enum {
  BED_NOP       = 0,           /* No ( binary Boolean ) operator */
  BED_VAR       = 0,           /* Variable ( same as BED_NOP ) */
  BED_ITE       = 0,           /* If-Then-Else ( same as BED_NOP ) */
  BED_K0        = 1,           /* Constant 0 */
  BED_PI1       = 2,           /* left-child */
  BED_PI2       = 3,           /* right-child */
  BED_NOR       = 4,
  BED_NLIMP     = 5,           /* Not left-imply */
  BED_NPI1      = 6,           /* Not left-child */
  BED_NIMP      = 7,           /* Not imply */
  BED_NPI2      = 8,           /* Not right-child */
  BED_XOR       = 9,
  BED_NAND      = 10,
  BED_AND       = 11,
  BED_BIIMP     = 12,          /* Logical equality */
  BED_IMP       = 13,          /* Imply */
  BED_LIMP      = 14,          /* left-imply */
  BED_OR        = 15,
  BED_K1        = 16,          /* Constant 1 */
  BED_EXISTS    = 17,          /* Existential quantifier */
  BED_FORALL    = 18,          /* Universal quantifier */
```

```
  BED_SUBST     = 19,           /* Substitution */
  BED_ESUB      = 20            /* Exists & Subst of all variables */
} bed_op;

typedef iList           bed_var_list;
typedef iList           bed_node_list;

typedef struct {
  unsigned int          node_count;             /* #nodes in use right now */
  unsigned int          highest_node_count;     /* highest #nodes in use   */
  unsigned int          number_of_variables;    /* #variables declared     */
  unsigned int          number_of_nodes;        /* #nodes declared         */
  unsigned int          number_of_gc;           /* #garbage collections    */
  unsigned long         gc_time;                /* Total GC time in msec   */
} bed_info;


#ifndef CPLUSPLUS
#define                 BED_UNDEFINED           ( (unsigned int) -1 )
#else
class bed;
extern const bed        BED_UNDEFINED;
#endif

#define BED_UPONE_MODE_ORDER    1
#define BED_UPONE_MODE_STRICT   2


/*=== Global variables ========================================*/

extern const bed_node   bed_zero;
extern const bed_node   bed_one;

#define                 bed_true                bed_one
#define                 bed_false               bed_zero

extern Boolean          bed_do_reductions;      /* Default: true */


/*=== BED interface prototypes ================================*/

#ifdef CPLUSPLUS
extern "C" {
#endif

/* Initialization, clearing, and freeing of BEDs */

void            bed_init( unsigned long n, unsigned long c );
void            bed_clear(void);
void            bed_done(void);
void            bed_clear_cache(void);
Boolean         bed_is_running(void);
```

```
/* Building and manipulating BEDs */

bed_var         bed_new_variables( unsigned int number );

bed_node        bed_mk( bed_var var, bed_op op, bed_node low, bed_node high );

#define bed_ith_var( var )              bed_mk( var, BED_VAR, bed_false, bed_true )
#define bed_mk_var( var, low, high )    bed_mk( var, BED_VAR, low, high )
#define bed_mk_op( op, low, high )      bed_mk( 0, op, low, high )
#define bed_mk_not( node )              bed_mk( 0, BED_NPI1, node, node )
#define bed_mk_exists( var, node )      bed_mk( var, BED_EXISTS, node, node )
#define bed_mk_forall( var, node )      bed_mk( var, BED_FORALL, node, node )
#define bed_mk_subst( var, in_node, with_node ) \
                        bed_mk( var, BED_SUBST, in_node, with_node )
#define bed_mk_esub( node )             bed_mk( 0, BED_ESUB, node, node )

bed_node        bed_upall( bed_node node );
bed_node        bed_upone( bed_var var, bed_node node );
bed_node        bed_upone_iter( bed_node node );
bed_node        bed_upsome( bed_var_list *vars, bed_node node );

bed_node        bed_apply( bed_op op, bed_node low, bed_node high );
bed_node        bed_restrict( bed_node u, bed_var var, Boolean val );
bed_node        bed_simplify( bed_node u, bed_node c );
bed_node        bed_quantdown( bed_node u );


/* Reference counting */

bed_node        bed_ref( bed_node node );
void            bed_deref( bed_node node );


/* Ordering */

void            bed_set_ordering( bed_var_list *var_list );
bed_var_list*   bed_get_ordering( void );


/* Garbage collection */

void            bed_gc();
extern void     (*bed_external_gc)();           /* Function which is called */
                                                /* before garbage collection */
extern void     (*bed_external_gc_post)();      /* Function which is called */
                                                /* after garbage collection */


/* Examining BEDs */

Boolean         bed_is_reachable( bed_node node, bed_node from_node );
Boolean         bed_is_bdd( bed_node node );
```

```
double          bed_sat_count( bed_node node );

bed_var_list*   bed_any_sat( bed_node node );
bed_var_list*   bed_any_nonsat( bed_node node );
Boolean         bed_evaluate( bed_node node, bed_var_list *assignment);

unsigned int    bed_node_count( bed_node node );
bed_var_list*   bed_support( bed_node node );


/* Mappings */

extern char*    (*bed_var2name)( bed_var );     /* Mappings of bed_vars & */
extern char*    (*bed_node2name)( bed_node );   /* bed_nodes to names     */
char*           bed_op2name( bed_op op );       /* bed_op to names        */


/* Functions for traversing BEDs */

bed_node        bed_get_low( bed_node node );
bed_node        bed_get_high( bed_node node );
bed_var         bed_get_var( bed_node node );
bed_op          bed_get_op( bed_node node );


/* Functions for reading, writing, & viewing BEDs */

/*
 *      Refer to "bedio.h"
 */

void            bed_io_view( bed_node node );


/* Internal settings */

void            bed_set_upone_mode( unsigned char tag );
unsigned char   bed_get_upone_mode();


/* Info */

bed_info        bed_get_info();

const char*     bed_version();
const char*     bed_author();
const char*     bed_addr();
const char*     bed_copyright();

#ifdef CPLUSPLUS
}
#endif
```

```
/*************************************************************************
  C++ part
*************************************************************************/

#ifdef CPLUSPLUS


/*=== BED class ========================================================*/

class bed
{
 public:

   bed(void)         { root=0; }
   bed(const bed &r) { bed_ref(root=r.root); }
   ~bed(void)        { bed_deref(root); }

   int id(void) const;

   bed operator=(const bed &r);

   bed operator&(const bed &r) const;
   bed operator&=(const bed &r);
   bed operator^(const bed &r) const;
   bed operator^=(const bed &r);
   bed operator|(const bed &r) const;
   bed operator|=(const bed &r);
   bed operator!(void) const;
   bed operator>>(const bed &r) const;
   bed operator>>=(const bed &r);
   bed operator-(const bed &r) const;
   bed operator-=(const bed &r);
   bed operator>(const bed &r) const;
   bed operator<(const bed &r) const;
   bed operator<<(const bed &r) const;
   bed operator<<=(const bed &r);
   int operator==(const bed &r) const;
   int operator!=(const bed &r) const;

private:
   bed_node root;

   /* Construct bed from bed_node */
   bed(bed_node r) { bed_ref(root=r); }
   bed operator=(bed_node r);


   /* Building and manipulating BEDs */

   friend bed   bed_mk_true(void);
```

```
   friend bed   bed_mk_false(void);
   friend bed   bed_mk_undefined(void);

   friend bed   bed_mk( bed_var var, bed_op op,
                        const bed &low, const bed &high );

   friend bed   bed_upall( const bed &node );
   friend bed   bed_upone( bed_var var, const bed &node );
   friend bed   bed_upone_iter( const bed &node );
   friend bed   bed_upsome( bed_var_list *vars, const bed &node );

   friend bed   bed_apply( bed_op op, const bed &low, const bed &high );
   friend bed   bed_restrict( const bed &u, bed_var var, Boolean val );
   friend bed   bed_simplify( const bed &u, const bed &c );
   friend bed   bed_quantdown( const bed &u );


   /* Examining BEDs */

   friend Boolean       bed_is_reachable( const bed &node, const bed &from );
   friend Boolean       bed_is_bdd( const bed &node );

   friend double        bed_sat_count( const bed &node );

   friend bed_var_list* bed_any_sat( const bed &node );
   friend bed_var_list* bed_any_nonsat( const bed &node );
   friend Boolean       bed_evaluate( const bed &node, bed_var_list *assignment);

   friend unsigned int  bed_node_count( const bed &node );
   friend bed_var_list* bed_support( const bed &node );


   /* Functions for traversing BEDs */

   friend bed           bed_get_low( const bed &node );
   friend bed           bed_get_high( const bed &node );
   friend bed_var       bed_get_var( const bed &node );
   friend bed_op        bed_get_op( const bed &node );


   /* Functions for viewing BEDs */

   friend void          bed_io_view( const bed &node );


   /* Hacks -- don't use unless you know what you are doing */

   friend bed           bed_from_int( bed_node r );
};


/*=== BED constants ====================================================*/
```

```
extern const bed bed_zero_pp;
extern const bed bed_one_pp;

#undef bed_true
#undef bed_false

#define bed_true        bed_one_pp
#define bed_false       bed_zero_pp
#define bed_one         bed_one_pp
#define bed_zero        bed_zero_pp


/*=== C++ interface ==================================================*/


inline bed bed_mk_true(void)
{ return 1; }

inline bed bed_mk_false(void)
{ return 0; }

inline bed bed_mk_undefined(void)
{ return (bed_node)-1; }

inline bed bed_mk( bed_var var, bed_op op, const bed &low, const bed &high )
{ return bed_mk( var, op, low.root, high.root ); }

inline bed bed_upall( const bed &node )
{ return bed_upall( node.root ); }

inline bed bed_upone( bed_var var, const bed &node )
{ return bed_upone( var, node.root ); }

inline bed bed_upone_iter( const bed &node )
{ return bed_upone_iter( node.root ); }

inline bed bed_upsome( bed_var_list *vars, const bed &node )
{ return bed_upsome( vars, node.root ); }

inline bed bed_apply( bed_op op, const bed &low, const bed &high )
{ return bed_apply( op, low.root, high.root ); }

inline bed bed_restrict( const bed &u, bed_var var, Boolean val )
{ return bed_restrict( u.root, var, val ); }

inline bed bed_simplify( const bed &u, const bed &c )
{ return bed_simplify( u.root, c.root ); }

inline bed bed_quantdown( const bed &u )
{ return bed_quantdown( u.root ); }

inline Boolean bed_is_reachable( const bed &node, const bed &from )
{ return bed_is_reachable( node.root, from.root ); }
```

```
inline Boolean bed_is_bdd( const bed &node )
{ return bed_is_bdd( node.root ); }

inline double bed_sat_count( const bed &node )
{ return bed_sat_count( node.root ); }

inline bed_var_list* bed_any_sat( const bed &node )
{ return bed_any_sat( node.root ); }

inline bed_var_list* bed_any_nonsat( const bed &node )
{ return bed_any_nonsat( node.root ); }

inline Boolean bed_evaluate( const bed &node, bed_var_list *assignment)
{ return bed_evaluate( node.root, assignment ); }

inline unsigned int bed_node_count( const bed &node )
{ return bed_node_count( node.root ); }

inline bed_var_list* bed_support( const bed &node )
{ return bed_support( node.root ); }

inline bed bed_get_low( const bed &node )
{ return bed_get_low( node.root ); }

inline bed bed_get_high( const bed &node )
{ return bed_get_high( node.root ); }

inline bed_var bed_get_var( const bed &node )
{ return bed_get_var( node.root ); }

inline bed_op bed_get_op( const bed &node )
{ return bed_get_op( node.root ); }

inline void bed_io_view( const bed &node )
{ bed_io_view( node.root ); }


/*=== Inline C++ functions ============================================*/

inline int bed::id(void) const
{ return root; }

inline bed bed::operator&(const bed &r) const
{ return bed_mk_op(BED_AND,*this,r); }

inline bed bed::operator&=(const bed &r)
{ return (*this=bed_mk_op(BED_AND,*this,r)); }

inline bed bed::operator^(const bed &r) const
{ return bed_mk_op(BED_XOR,*this,r); }

inline bed bed::operator^=(const bed &r)
```

```
{ return (*this=bed_mk_op(BED_XOR,*this,r)); }

inline bed bed::operator|(const bed &r) const
{ return bed_mk_op(BED_OR,*this,r); }

inline bed bed::operator|=(const bed &r)
{ return (*this=bed_mk_op(BED_OR,*this,r)); }

inline bed bed::operator!(void) const
{ return bed_mk_not(*this);}

inline bed bed::operator>>(const bed &r) const
{ return bed_mk_op(BED_IMP,*this,r); }

inline bed bed::operator>>=(const bed &r)
{ return (*this=bed_mk_op(BED_IMP,*this,r)); }

inline bed bed::operator-(const bed &r) const
{ return bed_mk_op(BED_NIMP,*this,r); }

inline bed bed::operator-=(const bed &r)
{ return (*this=bed_mk_op(BED_NIMP,*this,r)); }

inline bed bed::operator>(const bed &r) const
{ return bed_mk_op(BED_NIMP,*this,r); }

inline bed bed::operator<(const bed &r) const
{ return bed_mk_op(BED_NLIMP,*this,r); }

inline bed bed::operator<<(const bed &r) const
{ return bed_mk_op(BED_LIMP,*this,r); }

inline bed bed::operator<<=(const bed &r)
{ return (*this=bed_mk_op(BED_LIMP,*this,r)); }

inline int bed::operator==(const bed &r) const
{ return r.root==root; }

inline int bed::operator!=(const bed &r) const
{ return r.root!=root; }


#endif /* CPLUSPLUS */

#endif /* bed_h_ */
```

5 Library Functions

bed_any_nonsat – Returns a non-satisfying assignment

```
bed_var_list* bed_any_nonsat( bed_node node )
```

Description

Returns a non-satisfying assignment for the BED rooted at *node*. The result is a list of variables which are assigned the value true.

Return value

bed_var_list*

See also

bed_any_sat

bed_any_sat – Returns a satisfying assignment

```
bed_var_list* bed_any_sat( bed_node node )
```

Description

Returns a satisfying assignment for the BED rooted at *node*. The result is a list of variables which are assigned the value true.

Return value

bed_var_list*

See also

bed_any_nonsat

bed_apply – Connects two BDDs with a Boolean connective

```
bed_node bed_apply( bed_op op, bed_node low, bed_node high )
```

Description

The standard BDD apply. This function assumes that both *low* and *high* are BDDs. The result is a BDD. The ordering used is the global ordering set by bed_set_ordering.

Return value

bed_node which is the root of a BDD

See also

bed_upall, bed_upone_iter, bed_set_ordering

bed_clear – Clears any existing BED in memory

```
void bed_clear()
```

Description

Clears any existing BED in memory and empties the cache.

Return value

See also

bed_init, bed_done, bed_clear_cache

bed_clear_cache – Clears the internal cache

```
void bed_clear_cache()
```

Description

Clears the internal cache.

Return value

See also

bed_clear

bed_deref – Dereference count a vertex

```
void bed_deref( bed_node node )
```

Description

Unmark the vertex *node* which has previously been marked by bed_ref.

Return value

See also

bed_ref

bed_done – Terminates the use of the BED package

```
void bed_done()
```

Description

Terminates the use of the BED package by freeing memory.

Return value

See also

bed_init, bed_clear, bed_is_running

bed_evaluate – Evaluates a truth assignment on a BED

```
Boolean bed_evaluate( bed_node node, bed_var_list* assignment )
```

Description

Returns true if the truth assignment *assignment* evaluates to true on the BED *node*. The assignment is a list of variables assigned the value true.

Return value

Boolean

See also

bed_external_gc – Function called by the garbage collector

```
void (*bed_external_gc)()
```

Description

The function pointed to by bed_external_gc is called by the garbage collecter before any garbage collection is done. Default is NULL meaning no function is called.

Return value

See also

bed_gc, bed_external_gc_post

bed_external_gc_post – Function called by the garbage collector

```
void (*bed_external_gc_post)()
```

Description

The function pointed to by bed_external_gc is called by the garbage collecter after the garbage collection has been done. Default is NULL meaning no function is called.

Return value

See also

bed_gc, bed_external_gc

bed_gc – Garbage collection

```
void bed_gc()
```

Description

Performs a garbage collection. All vertices which are not marked by bed_ref and cannot be reached from a marked vertex are garbage collected. Garbage collection happens automatically when the BED package runs out of memory.

Return value

See also

bed_external_gc

bed_get_high – Returns the high child

```
bed_node bed_get_high( bed_node node )
```

Description

Returns the high child of vertex *node*.

Return value

bed_node

See also

bed_get_low, bed_get_op, bed_get_var

bed_get_info – Return info about the BED

```
bed_info bed_get_info()
```

Description

Returns info about the current state of the BED package.

Return value

bed_info

See also

bed_get_low – Returns the low child

```
bed_node bed_get_low( bed_node node )
```

Description

Returns the low child of vertex *node*.

Return value

bed_node

See also

bed_get_high, bed_get_op, bed_get_var

bed_get_op – Returns the operator

```
bed_op bed_get_op( bed_node node )
```

Description

Returns the operator of vertex *node*.

Return value

bed_op

See also

bed_get_low, bed_get_high, bed_get_var

bed_get_ordering – Returns the current variable ordering

```
bed_var_list* bed_get_ordering()
```

Description

Returns the current variable ordering.

Return value

bed_var_list*

See also

bed_set_ordering

bed_get_upone_mode – Returns the current upone mode

```
unsigned char bed_get_upone_mode()
```

Description

Returns the current upone mode. It is either BED_UPONE_MODE_ORDER or BED_UPONE_MODE_STRICT. The former mode pulls variables up until they reach their place in the ordering. The latter mode pulls variables up to the root regardless of the current ordering.

Return value

See also

bed_set_upone_mode, bed_upone

bed_get_var – Returns the variable

```
bed_var bed_get_var( bed_node node )
```

Description

Returns the variable of vertex *node*.

Return value

bed_var

See also

bed_get_low, bed_get_high, bed_get_op

bed_init – Initializes the BED package

```
void bed_init( unsigned long n, unsigned long c )
```

Description

Initializes the BED package. The parameter n specifies the number of bytes to allocate for vertices while c specifies the number of bytes to allocate for the cache.

Return value

See also

bed_done, bed_clear, bed_is_running

bed_io_dimacs – Outputs a BED in the DIMACS format

```
void bed_io_dimacs( FILE *fp, bed_node node )
```

Description

Outputs the BED *node* to the file fp in the DIMACS format.

Return value

See also

bed_io_graph – Outputs BEDs as graphs in the DOT format

```
void bed_io_graph( FILE *fp, bed_node_list *nodes )
```

Description

Outputs the BEDs in the list *nodes* to the file *fp* in the DOT format.

Return value

See also

bed_io_view

bed_io_print_node – Outputs a BED vertex as text

```
void bed_io_print_node( FILE *fp, bed_node node )
```

Description

Outputs the BED *node* as text to the file *fp*.

Return value

See also

bed_io_read – Retrieves BEDs from file

```
bed_io_read_info* bed_io_read( char *filename )
```

Description

Retrieves BEDs from the file named *filename*. The result is a pointer to a `bed_io_read_info` structure.

Return value

bed_io_read_info*

See also

bed_io_read_done, bed_io_write

bed_io_read_done – Cleans up after bed_io_read

```
void bed_io_read_done( bed_io_read_info *info )
```

Description

Cleans up the bed_io_read_info structure after bed_io_read. All memory are deallocated.

Return value

See also

bed_io_read

bed_io_view – Shows a graphical view of a BED

```
void bed_io_view( bed_node node )
```

Description

Shows a graphical view of the BED *node*. This command uses the DOT and GHOSTVIEW programs.

Return value

See also

bed_io_graph

bed_io_write – Writes BEDs to a file

```
void bed_io_write( FILE *fp, bed_io_root_entry* nodes )
```

Description

Writes BEDs in *nodes* to the file *fp* for later retrieval.

Return value

See also

bed_io_read

bed_is_bdd – Tests whether a BED is a BDD

```
Boolean bed_is_bdd( bed_node node )
```

Description

Returns true if the BED at vetex *node* is a BDD.

Return value

Boolean

See also

bed_is_reachable – Reachability of a vertex from another vertex

```
Boolean bed_is_reachable( bed_node node, bed_node from_node )
```

Description

Returns true if vertex *node* is reachable from vertex $from\_node$.

Return value

Boolean

See also

bed_is_running – Check if the BED package is running

```
Boolean bed_is_running()
```

Description

Returns true if the BED package is running. The BED package is running after the function bed_init has been called and until the function bed_done has been called.

Return value

Boolean

See also

bed_init, bed_done

bed_mk – Creates a BED vertex

```
bed_node bed_mk( bed_var var, bed_op op, bed_node low, bed_node high )
```

Description

Creates a BED vertex with variable *var*, operator *op*, low child *low*, and high child *high*. Special care should be taken to avoid building BEDs which are not free.

Return value

bed_node

See also

Other bed_mk functions

bed_mk_esub – Creates a BED esub vertex

```
bed_node bed_mk_esub( bed_node node )
```

Description

Creates a vertex for the Boolean function existentially quantifying all variables with an even number and substitute all odd variables with variables having a number which is one less. This is useful in fixed-point computations.

Return value

bed_node

See also

Other bed_mk functions

bed_mk_exists – Creates a BED existential quantification vertex

```
bed_node bed_mk_exist( bed_var var, bed_node node )
```

Description

Creates a vertex for the Boolean function $\exists\ var\ :\ node$.

Return value

bed_node

See also

Other bed_mk functions

bed_mk_forall – Universal quantification of a BED variable

```
bed_node bed_mk_forall( bed_var var, bed_node node )
```

Description

Creates a BED for the Boolean function $\forall$ *var* : *node*.

Return value

bed_node

See also

Other bed_mk functions

bed_mk_not – Creates a BED negation vertex

```
bed_node bed_mk_not( bed_node node )
```

Description

Creates a vertex for the Boolean function *not node*.

Return value

bed_node

See also

Other bed_mk functions

bed_mk_op – Creates a BED operator vertex

```
bed_node bed_mk_op( bed_op op, bed_node low, bed_node high )
```

Description

Creates a vertex for the Boolean function *low op high*.

Return value

bed_node

See also

Other bed_mk functions

bed_mk_subst – Creates a BED substitution vertex

```
bed_node bed_mk_subst( bed_var var, bed_node in_node, bed_node with_node )
```

Description

Creates a vertex for the Boolean function $in\_node[var := with\_node]$.

Return value

bed_node

See also

Other bed_mk functions

bed_mk_var – Creates a BED variable vertex

```
bed_node bed_mk_var( bed_var var, bed_node low, bed_node high )
```

Description

Creates a vertex for the Boolean function *if var then high else low*. Special care should be taken to avoid building BEDs which are not free.

Return value

bed_node

See also

Other bed_mk functions

bed_new_variables – Declares new variables

```
bed_var bed_new_variables( unsigned int number )
```

Description

Declares *number* new Boolean variables. The variables are identified by *number* successive numbers starting from the return value.

Return value

bed_var

See also

bed_node_count – Counts the number of vertices

```
unsigned int bed_node_count( bed_node node )
```

Description

Counts the number of vertices in the BED rooted at *node*. Terminal vertices are not counted.

Return value

unsigned int

See also

bed_op2name – Converts an operator to a string

```
char* bed_op2name( bed_op op )
```

Description

Returns a string which the name of the operator *op*.

Return value

char*

See also

bed_quantdown – Pushes quantifiers down into a formula

```
bed_node bed_quantdown( bed_node u )
```

Description

Pushes quantifiers down into a formula. This works best if the reduction rules are turned on.

Return value

bed_node which is the root of a BED

See also

bed_ref – Reference count a vertex

```
bed_node bed_ref( bed_node node )
```

Description

Mark the vertex *node* and all vertices reachable from it so that they will not be garbage collected. Use bed_deref to unmark a vertex. All functions expect their input to be marked and return unmarked vertices. It is possible to call bed_ref multiple times for a vertex, in which case bed_deref should be called the same number of times.

Return value

The input vertex *node*

See also

bed_deref

bed_restrict – Restricts a variable to a Boolean value

```
bed_node bed_restrict( bed_node u, bed_var var, Boolean val )
```

Description

The standard BDD restrict. This function returns a BED in which all occurences of *var* are replaced with either $low(var)$ or $high(var)$ depending on whether *val* is **false** or **true**.

Return value

bed_node which is the root of a BED

See also

bed_sat_count – Counts satisfying assignments

```
double bed_sat_count( bed_node node )
```

Description

Returns the number of satisfying assignments of the Boolean function represented by vertex *node*. *node* must be a BDD.

Return value

double. A negative number indicates an error.

See also

bed_set_ordering – Sets the variable ordering

```
void bed_set_ordering( bed_var_list *var_list )
```

Description

Sets the variable ordering to the order in which the variables occur in *var_list*. Variables which are not mentioned are placed last in the ordering. The default ordering is the ordering in which the variables have been declared.

Return value

See also

bed_get_ordering

bed_set_upone_mode – Sets the upone mode

```
void bed_set_upone_mode( unsigned char tag )
```

Description

Sets the upone mode to BED_UPONE_MODE_ORDER or BED_UPONE_MODE_STRICT. The former mode pulls variables up until they reach their place in the ordering. The latter mode pulls variables up to the root regardless of the current ordering. Default is BED_UPONE_MODE_ORDER.

Return value

See also

bed_get_upone_mode, bed_upone

bed_simplify – Simplifies a formula under a condition

```
bed_node bed_simplify( bed_node u, bed_node c )
```

Description

The simplify function from Coudert, Berthet, and Madre (1989). The BED u is simplified to a BED u' such that $f^c \rightarrow (f^u = f^{u'})$, where f^v is the formula represented by BED v.

Return value

bed_node which is the root of a BED

See also

bed_support – Returns the support of a BED

```
bed_var_list* bed_support( bed_node node )
```

Description

Returns the support of the BED *node*. The support is a list of all the variables occurring in *node* and the vertices reachable from it.

Return value

bed_var_list*

See also

bed_upall – Transforms a BED to a BDD

```
bed_node bed_upall( bed_node node )
```

Description

Transforms a BED to a BDD by removing all operator vertices. The transformation closely resembles the standard BDD apply algorithm for building BDDs. The variable ordering of the resulting BDD is the global variables ordering set by bed_set_ordering.

Return value

bed_node

See also

bed_upone, bed_upone_iter, bed_upsome, bed_set_ordering

bed_upone – Lifts one variable up

```
bed_node bed_upone( bed_var var, bed_node node )
```

Description

Lifts the variable *var* up. How far the variable is lifted depends on the current mode. In BED_UPONE_MODE_ORDER mode, upone lifts *var* to either vertex *node* or to just below a variable vertex with a variable which comes before *var* in the global ordering. In BED_UPONE_MODE_STRICT mode, upone lifts *var* to vertex *node*.

Return value

bed_node which is the root of a BED

bed_upone_iter – Transforms a BED to a BDD

```
bed_node bed_upone_iter( bed_node node )
```

Description

Transforms a BED to a BDD by lifting all variables up above the operators vertices. The variables are lifted up in the order set by bed_set_ordering.

Return value

bed_node which is the root of a BDD

See also

bed_upone, bed_upall, bed_upsome, bed_set_ordering

bed_upsome – Lifts a set of variables up

```
bed_node bed_upsome( bed_var_list *vars, bed_node node )
```

Description

Lifts the variables *vars* up to vertex *node*. Uses the global ordering, however, variables in *vars* are always lifted above variables outside *vars*.

Return value

bed_node which is the root of a BED

See also

bed_upall, bed_upone_iter, bed_set_ordering

il_append – Appends an integer to an integer list

```
void il_append( iList *l, int e )
```

Description

Appends the integer e to the list l. e is appended at the end of the list.

Return value

See also

il_clear – Clears a list

```
void il_clear( iList *l )
```

Description

Clears list l.

Return value

See also

il_copy – Copies an integer list

```
iList*  il_copy( iList *a )
```

Description

Returns a copy of the integer list a

Return value

iList*

See also

il_elem – Gives access to an element in the list

```
#define il_elem( a, n )
```

Description

Gives access to element number n in list a. Can be used on both left and right hand sides of '='.

Return value

See also

il_empty – Check if a list is empty

```
Boolean il_empty( iList *l )
```

Description

Returns true if list *l* is empty.

Return value

Boolean

See also

il_free – Frees an integer list

```
void il_free( iList *a )
```

Description

Frees the integer list *a*

Return value

See also

il_new

il_iter – Iterates through all elements of a list

```
#define IL_ITER( l, i )
```

Description

Iterates through all elements of the list l. The integer i starts at 0 and ends at il_length(l)-1. Use il_elem(l,i) to access each element.

Return value

See also

il_length – Gives the length of a list

```
unsigned int il_length( iList *l )
```

Description

Returns the length of list l.

Return value

unsigned int

See also

il_new – Creates a new and empty integer list

```
iList* il_new()
```

Description

Creates a new and empty interger list

Return value

iList*

See also

il_free