NAME

bdd − a binary decision diagram (BDD) package

SYNOPSIS

#include <bdduser.h>

DESCRIPTION

The **libbdd** library provides a set of routines for manipulating binary decision diagrams (BDDs). Some support is also provided for multi-terminal BDDs (MTBDDs). Programs designed to be used with the library should use the **-lbdd -lmem** options to **cc** when linking.

LIST OF FUNCTIONS

Name	*Function*
bdd_init	Initialize the library
bdd_version	Get BDD version string
bdd_quit	Finish using the library
bdd_new_var_first	Create a variable first in the order
bdd_new_var_last	Create a variable last in the order
bdd_new_var_before	Create a variable before an existing one
bdd_new_var_after	Create a variable after an existing one
bdd_var_with_index	Obtain an existing variable
bdd_var_with_id	Obtain an existing variable
bdd_one	Constant TRUE
bdd_zero	Constant FALSE
bdd_and	Logical AND
bdd_nand	Logical NAND
bdd_or	Logical OR
bdd_nor	Logical NOR
bdd_xor	Logical XOR
bdd_xnor	Logical XNOR
bdd_identity	Logical identity
bdd_not	Logical NOT
bdd_ite	Logical IF-THEN-ELSE
bdd_if	Get the variable of the top node in a BDD
bdd_then	Get the THEN branch of the top node in a BDD
bdd_else	Get the ELSE branch of the top node in a BDD
bdd_if_index	Get the index of the top variable in a BDD
bdd_if_id	Get a unique ID number for the top variable
bdd_intersects	Check intersection
bdd_implies	Check boolean implication
bdd_new_assoc	Make a new variable association
bdd_free_assoc	Free a variable association
bdd_temp_assoc	Set the temporary variable association
bdd_augment_temp_assoc	Set the temporary variable association
bdd_assoc	Set the current variable association
bdd_exists	Existential quantification
bdd_forall	Universal quantification
bdd_rel_prod	Relational product
bdd_compose	Substitute for a variable
bdd_substitute	Substitute for a set of variables
bdd_reduce	Simplify given a constraint
bdd_cofactor	Generalized cofactor
bdd_depends_on	Determine if a BDD depends on a variable
bdd_support	Find the support of a BDD
bdd_satisfy	Find a satisfying assignment
bdd_satisfy_support	Find a satisfying assignment

bdd_satisfying_fraction	Fraction of valuations satisfying a BDD
bdd_swap_vars	Swap two variables in a BDD
bdd_apply2	Generic apply routine
bdd_apply1	Generic apply routine
bdd_size	Number of nodes in a BDD
bdd_size_multiple	Number of nodes in multiple BDDs
bdd_profile	Node profile of a BDD
bdd_profile_multiple	Node profile of multiple BDDs
bdd_function_profile	Function profile of a BDD
bdd_function_profile_multiple	Function profile of multiple BDDs
bdd_print_bdd	Print a BDD in human-readable form
bdd_print_profile	Print a node profile of a BDD
bdd_print_profile_multiple	Print a profile of multiple BDDs
bdd_print_function_profile	Print a function profile of a BDD
bdd_dump_bdd	Write a BDD to a file
bdd_undump_bdd	Load a BDD from a file
bdd_type	Classify a BDD
bdd_free	Decrease the reference count of a BDD
bdd_unfree	Increase the reference count of a BDD
bdd_clear_refs	Set all BDD reference counts to zero
bdd_gc	Garbage collect unused BDD nodes
bdd_total_size	Total number of BDD nodes in use
bdd_vars	Total number of variables in existence
bdd_cache_ratio	Get/set operation result cache size
bdd_node_limit	Get/set the number of BDD nodes allowed
bdd_overflow	Get/clear overflow flag
bdd_overflow_closure	Set a closure to invoke on overflow
bdd_abort_closure	Used to abort operations in progress
bdd_stats	Print statistics
bdd_dynamic_reordering	Specify dynamic reordering technique
bdd_reorder	Invoke dynamic reordering
bdd_new_var_block	Create variable block
bdd_var_block_reorderable	Set block reorderability
mtbdd_free_terminal_closure	Called when freeing an MTBDD terminal
mtbdd_get_terminal	Get an MTBDD terminal node
mtbdd_terminal_value	Get the value of an MTBDD terminal node
mtbdd_ite	IF-THEN-ELSE operation for MTBDDs
mtbdd_equal	Equality operation for MTBDDs
mtbdd_transform	Applies the current transform to an MTBDD
mtbdd_transform_closure	Called to set the MTBDD transform
mtbdd_one_data	Sets the MTBDD data value for TRUE

BASIC CONCEPTS

For a general overview of BDDs, see the original article by Bryant [1].

Almost all of the BDD library routines require a BDD manager as one of their arguments. A BDD manager is a structure which holds various variables used by the BDD routines. The type **bdd_manager** is a pointer to this structure. BDDs themselves are also represented internally as structures. The type **bdd** is a pointer to one of these structures.

There is a global ordering on the boolean variables which may appear in a BDD. The variable at the root of a BDD is earlier in the ordering than all other variables in the BDD. Each variable has an index which represents its position in the ordering; *v1* appears before *v2* in the ordering if and only if the index for *v1* is less than the ordering for *v2*. Each variable is also assigned a unique ID number that is invariant. Since variables can be created at any position within the order, this is not true for the index. Also, the library

supports dynamic variable reordering. With dynamic variable reordering, variables may be shuffled around in the middle of an operation in order to reduce the number of BDD nodes in use.

Some routines such as **bdd_substitute** require a mapping from variables to BDDs to operate. This mapping is supplied in the form of a variable association which is a set of pairs. The first element of each pair is the variable, and the second element is the BDD that the variable is associated with. Multiple associations may exist at any one time. Other routines such as **bdd_exists** require sets of variables. Sets of variables are represented by variable associations where only the fact that a variable is associated with some BDD is significant. There is one association, called the temporary variable association, which is special in two ways. First, this association always exists. Second, results are not cached across calls when this association is used. The temporary association is intended for when an association will not be reused. The advantage of using it is that setting the temporary association does not require scanning the result cache to flush out-of-date results.

The results returned by the library represent canonical forms and may be checked for equivalence using the standard C comparison operators. For example:

```
{
 bdd_manager bddm;
 bdd f;
 ...
 if (f == bdd_one(bddm))  /* Tautology check */
   ...
}
```

For checking for relations such as boolean implication, use **bdd_intersects** and **bdd_implies**.

Multi-terminal BDDs are like BDDs, except an MTBDD may have more than just the constants TRUE and FALSE at the leaves. Passing an MTBDD to a routine expecting a BDD will give undefined results, except where noted below. MTBDDs are built up using **mtbdd_get_terminal** and **mtbdd_ite**.

STORAGE MANAGEMENT

Each BDD node has an associated reference count which records the number of references to the BDD (internal and external). Whenever a BDD is returned from a function, the reference count for its top node is incremented. (If the BDD did not exist before, the reference count will be 1.) Each time a garbage collection occurs, either internally or because of a call to **bdd_gc**, all nodes which are not referenced are reclaimed. The reference count of a BDD may be decremented by calling **bdd_free**. This should be done whenever possible for maximum space efficiency. You may also specify a limit for the total number of BDD nodes using **bdd_node_limit**. If it is not possible to complete an operation without exceeding this limit, the operation is aborted and (by default) a null pointer is returned. Whenever this happens, the reference counts of all nodes are restored to what they were before the operation. If a null pointer is passed to a routine, the routine simply returns null. Thus, it is not necessary to check for overflows after each operation. There is also an internal flag that indicates whether any operation has caused an overflow. It may be read and reset by **bdd_overflow**. Optionally, a user-defined closure may be invoked when an overflow occurs; see **bdd_overflow_closure**. Also see **bdd_free**, **bdd_unfree**, **bdd_clear_refs**, **bdd_node_limit** and **bdd_gc**. The library also includes high-performance replacements for **malloc** and **free**. See the discussion at the end of the section on adding new routines.

DETAILED DESCRIPTION

bdd_manager
bdd_init()

Creates and initializes a new BDD manager. Multiple BDD managers may exist at any time.

char *
bdd_version()

Returns a string identifying the version number of the BDD library.

void
bdd_quit(bddm)
bdd_manager bddm;

Deallocates the BDD manager given by **bddm** and all the storage associated with it.

bdd
bdd_new_var_first(bddm)
bdd_manager bddm;

Creates a new variable at the start of the BDD variable ordering and returns the BDD for it.

bdd
bdd_new_var_last(bddm)
bdd_manager bddm;

Creates a new variable at the end of the BDD variable ordering and returns the BDD for it.

bdd
bdd_new_var_before(bddm, var)
bdd_manager bddm;
bdd var;

Creates a new variable which is before **var** in the BDD variable ordering and returns the BDD for the new variable.

bdd
bdd_new_var_after(bddm, var)
bdd_manager bddm;
bdd var;

Creates a new variable which is after **var** in the BDD variable ordering and returns the BDD for the new variable.

bdd
bdd_var_with_index(bddm, i)
bdd_manager bddm;
long i;

If a variable with index **i** has been created, returns the BDD for the variable. If no such variable exists, returns null. See also **bdd_if_index**.

bdd
bdd_var_with_id(bddm, i)
bdd_manager bddm;
long i;

If a variable with ID **i** has been created, returns the BDD for the variable. If no such variable has been created, returns null. See also **bdd_if_id**.

bdd
bdd_one(bddm)
bdd_manager bddm;

Returns the BDD for the constant TRUE.

bdd
bdd_zero(bddm)
bdd_manager bddm;

Returns the BDD for the constant FALSE.

bdd
bdd_and(bddm, f, g)
bdd_manager bddm;
bdd f, g;

Returns the BDD for the logical AND of **f** and **g**.

bdd

bdd_nand(bddm, f, g)
bdd_manager bddm;
bdd f, g;

Returns the BDD for the logical NAND of **f** and **g**.

bdd
bdd_or(bddm, f, g)
bdd_manager bddm;
bdd f, g;

Returns the BDD for the logical OR of **f** and **g**.

bdd
bdd_nor(bddm, f, g)
bdd_manager bddm;
bdd f, g;

Returns the BDD for the logical NOR of **f** and **g**.

bdd
bdd_xor(bddm, f, g)
bdd_manager bddm;
bdd f, g;

Returns the BDD for the logical XOR of **f** and **g**.

bdd
bdd_xnor(bddm, f, g)
bdd_manager bddm;
bdd f, g;

Returns the BDD for the logical XNOR of **f** and **g**.

bdd
bdd_identity(bddm, f)
bdd_manager bddm;
bdd f;

Returns the BDD for **f**. The only real effect of this function is to increase the reference count of **f**. Also works with MTBDDs.

bdd
bdd_not(bddm, f)
bdd_manager bddm;
bdd f;

Returns the BDD for the logical NOT of **f**.

bdd
bdd_ite(bddm, f, g, h)
bdd_manager bddm;
bdd f, g, h;

Returns the BDD for the logical operation IF **f** THEN **g** ELSE **h**.

bdd
bdd_if(bddm, f)
bdd_manager bddm;
bdd f;

Returns the BDD for the variable which labels the root of the BDD given by **f**. Also works with MTBDDs. The result is undefined if **f** is one of the constants TRUE or FALSE or an MTBDD terminal node.

bdd
bdd_then(bddm, f)
bdd_manager bddm;
bdd f;

Returns the BDD for the THEN branch of the root of the BDD given by **f**. Also works with MTBDDs. The result is undefined if **f** is one of the constants TRUE or FALSE or an MTBDD terminal node.

bdd
bdd_else(bddm, f)
bdd_manager bddm;
bdd f;

Returns the BDD for the ELSE branch of the root of the BDD given by **f**. Also works with MTBDDs. The result is undefined if **f** is one of the constants TRUE or FALSE or an MTBDD terminal node.

long
bdd_if_index(bddm, f)
bdd_manager bddm;
bdd f;

Returns the index of the variable which labels the root of the BDD given by **f**. Also works with MTBDDs. The result is undefined if **f** is one of the constants TRUE or FALSE or an MTBDD terminal node. The variable at the start of variable ordering has index 0, the next has index 1, etc. Note that creating new variables may change the index of existing variables. Dynamic reordering may also change the index of variables.

long
bdd_if_id(bddm, f)
bdd_manager bddm;
bdd f;

Returns a unique ID number for the variable which labels the root of the BDD given by **f**. Also works with MTBDDs. The result is undefined if **f** is one of the constants TRUE or FALSE or an MTBDD terminal node. The ID for a variable is fixed at the time the variable is created and never changes after that.

bdd
bdd_intersects(bddm, f, g)
bdd_manager bddm;
bdd f, g;

Computes a BDD that implies the conjunction of **f** and **g**. If the conjunction is not FALSE, then the BDD returned will not be FALSE. Also, the function tries to construct as few new nodes as possible. This routine is intended for cases where you need to test for a FALSE conjunction, and, when it the conjunction is not FALSE, to obtain just one valuation satisfying both **f** and **g**. A non-FALSE result from **bdd_intersects** can be passed directly to a routine like **bdd_satisfy_support**.

bdd
bdd_implies(bddm, f, g)
bdd_manager bddm;
bdd f, g;

This is equivalent to calling **bdd_intersects** with **f** and NOT **g**.

int
bdd_new_assoc(bddm, assoc, pairs)
bdd_manager bddm;
bdd *assoc;
int pairs;

Creates or finds a variable association. The association is specified by **assoc** and should be a null-terminated array of BDDs. If **pairs** is 0, the array is assumed to be an array of variables. In this case, each variable is paired with the BDD for TRUE. Such an association may essentially be viewed as specifying a set of variables for use with routines such as **bdd_exists**. If **pairs** is nonzero, then the even numbered array elements should be variables and the odd numbered

elements should be the BDDs which they are mapped to. In both cases, the return value is an integer identifier for this association. Note: if the given association is equivalent to one which already exists, the same identifier is used for both, and the reference count of the association is increased by one.

void
bdd_free_assoc(bddm, id)
bdd_manager bddm;
int id;

Decrements the reference count of the variable association with identifier **id**, and frees it if the reference count becomes zero.

void
bdd_temp_assoc(bddm, assoc, pairs)
bdd_manager bddm;
bdd *assoc;
int pairs;

Sets the temporary variable association. The arguments **assoc** and **pairs** are as in **bdd_new_assoc**.

void
bdd_augment_temp_assoc(bddm, assoc, pairs)
bdd_manager bddm;
bdd *assoc;
int pairs;

Add to the temporary variable association. The arguments **assoc** and **pairs** are as in **bdd_new_assoc**. Any existing associations are overwritten. This is mainly used when doing things like substituting for all variables in a BDD. It isn't necessary to clear out the temporary association in such cases, so you can save a little time by using this routine.

int
bdd_assoc(bddm, id)
bdd_manager bddm;
int id;

Sets the current variable association to the one identified by **id**. The identifier for the old current association is returned. The temporary variable association has identifier -1.

bdd
bdd_exists(bddm, f)
bdd_manager bddm;
bdd f;

Returns the BDD for **f** with all the variables that are paired with something in the current variable association existentially quantified out.

bdd
bdd_forall(bddm, f)
bdd_manager bddm;
bdd f;

Returns the BDD for **f** with all the variables that are paired with something in the current variable association universally quantified out.

bdd
bdd_rel_prod(bddm, f, g)
bdd_manager bddm;
bdd f, g;

Returns the BDD for the logical AND of **f** and **g** with all the variables that are paired with something in the current variable association existentially quantified out. If **f** and **g** are viewed as boolean relations, this operation corresponds to relational product. This routine is generally much more efficient than doing the operations separately.

bdd
bdd_compose(bddm, f, g, h)
bdd_manager bddm;
bdd f, g, h;

Returns the BDD for the substitution of **h** for the variable **g** in **f**. When **h** does not depend on **g**, the operation may be viewed as composition of boolean functions. If **h** does depend on **g**, it corresponds to instantaneous substitution in a boolean formula.

bdd
bdd_substitute(bddm, f)
bdd_manager bddm;
bdd f;

Returns the BDD for **f** under a substitution defined by the current variable association. Each variable is replaced by its associated BDD. The substitution is effectively simultaneous.

bdd
bdd_reduce(bddm, f, g)
bdd_manager bddm;
bdd f, g;

Returns a BDD which agrees with **f** for all valuations which satisfy **g**. The result is usually smaller in terms of number of BDD nodes than **f**. This operation is typically used in state space searches to simplify the representation for the set of states which will be expanded at each step.

bdd
bdd_cofactor(bddm, f, g)
bdd_manager bddm;
bdd f, g;

Returns a BDD for the generalized cofactor of **f** by **g**. The BDD indicated by **g** should not be the constant FALSE. For some properties of this operation, see Touati *et al.* [2].

int
bdd_depends_on(bddm, f, g)
bdd_manager bddm;
bdd f;
bdd g;

Returns 1 if the BDD or MTBDD **f** depends on the variable given by the BDD **g**, and returns 0 otherwise.

void
bdd_support(bddm, f, support)
bdd_manager bddm;
bdd f;
bdd *support;

Stores the support of **f** as a null-terminated sequence of variables in **support**. Works for MTBDDs also.

bdd
bdd_satisfy(bddm, f)
bdd_manager bddm;
bdd f;

Returns a BDD which is not false, implies **f**, and has at most one BDD node at each level. The BDD indicated by **f** should not be the constant FALSE.

bdd
bdd_satisfy_support(bddm, f)
bdd_manager bddm;
bdd f;

Returns a BDD which is not false, implies **f**, has at most one BDD node at each level, and has a node labeled with each variable which is paired with something in the current variable

association. If **f** is the constant FALSE, the result is undefined.

double
bdd_satisfying_fraction(bddm, f)
bdd_manager bddm;
bdd f;

Returns the fraction of valuations which satisfy **f**. If **f** is a function of *n* variables, then 2 to the power *n* times this fraction is the number of valuations which satisfy **f**.

bdd
bdd_swap_vars(bddm, f, g, h)
bdd_manager bddm;
bdd f;
bdd g;
bdd h;

Returns the BDD for **f** with **g** substituted for **h** and **h** substituted for **g**. The substitution is effectively simultaneous.

bdd
bdd_apply2(bddm, terminal_fn, f, g, env)
bdd_manager bddm;
bdd (*terminal_fn)();
bdd f;
bdd g;
pointer env;

This is a generic two-argument operation. The behavior of the operation on terminal values is given by **terminal_fn**. It should take as arguments: the BDD manager, pointers to two BDDs (the arguments for the call), and the pointer given by **env**. If the value of the call can be determined immediately from the arguments, it should return that value. Otherwise, it should return a null pointer. In this case, it may also use the BDD pointers that it received to alter the arguments to the call. A typical use for this ability is to put the arguments in a canonical order for commutative operations. The function should not alter the reference counts of either the arguments or the returned value. Also, the returned value (if non-null) has its temporary reference count incremented once automatically. If your function always returns one of the arguments or TRUE or FALSE, this is the right thing and you don't have to worry about it. If you want to call other routines to determine the return value, you should read the section on adding new routines below. Works with MTBDDs.

bdd
bdd_apply1(bddm, terminal_fn, f, env)
bdd_manager bddm;
bdd (*terminal_fn)();
bdd f;
pointer env;

This is a generic one-argument operation. It is basically like **bdd_apply2**, except that **terminal_fn** takes a single BDD pointer argument instead of the pair of pointers in the two-argument case. Works with MTBDDs.

long
bdd_size(bddm, f, negout)
bdd_manager bddm;
bdd f;
int negout;

Returns the number of nodes in **f**. The parameter **negout** is a flag indicating whether negative output pointers should be considered. The library uses this type of pointer flag internally, so if the flag is nonzero, the actual number of nodes used is returned. If the flag is zero, the return value is the number of nodes which would be needed to represent **f** using a basic BDD. Works

for MTBDDs too.

long
bdd_size_multiple(bddm, fs, negout)
bdd_manager bddm;
bdd *fs;
int negout;

Returns the number of nodes in the set of BDDs or MTBDDs given by **fs**, which should be a null-terminated array. Nodes which are shared among the BDDs are only counted once. The parameter **negout** is as in **bdd_size**.

void
bdd_profile(bddm, f, level_counts, negout)
bdd_manager bddm;
bdd f;
long *level_counts;
int negout;

Returns the ''node profile'' of **f**, i.e., the number of nodes at each level in **f**. The parameter **level_counts** should be an array of longs of size one plus the number of variables in existence (see **bdd_vars**). On return, this array holds the profile; the *i*th entry is the number of nodes labeled with the variable of index *i*. The last entry corresponds to the nodes for TRUE and FALSE. The parameter **negout** is as in **bdd_size**. Works for MTBDDs too; in this case, the last entry corresponds to the MTBDD terminal nodes.

void
bdd_profile_multiple(bddm, fs, level_counts, negout)
bdd_manager bddm;
bdd* fs;
long *level_counts;
int negout;

Returns the ''node profile'' of the set of BDDs or MTBDDs given by **fs**, which should be a null-terminated array. The parameters **level_counts** and **negout** are as in **bdd_profile**.

void
bdd_function_profile(bddm, f, func_counts)
bdd_manager bddm;
bdd f;
long *func_counts;

Returns the ''function profile'' of **f**, i.e., the number of functions at or below each level in **f**. The parameter **func_counts** should be an array of longs of size one plus the number of variables in existence (see **bdd_vars**). On return, this array holds the profile. The *i*th entry corresponds to the number of functions which can be obtained by restricting those variables of index less than *i*, provided that **f** has at least one node labeled with the variable of index *i*. If **f** has no nodes labeled with the variable of index *i*, then the *i*th entry of the profile is 0. Works for MTBDDs also.

void
bdd_function_profile_multiple(bddm, fs, func_counts)
bdd_manager bddm;
bdd *fs;
long *func_counts;

Returns the ''function profile'' of the set of BDDs or MTBDDs given by **fs**, which should be a null-terminated array. The parameter **func_counts** is as in **bdd_function_profile**.

void
bdd_print_bdd(bddm, f, naming_fn, terminal_id_fn, env, fp)
bdd_manager bddm;
bdd f;
char *(*naming_fn)();

char *(*terminal_id_fn)();
pointer env;
FILE *fp;

Prints a human-readable representation of the BDD or MTBDD **f** to the file given by **fp**. The **naming_fn** should be a pointer to a function taking a **bdd_manager**, a **bdd** and the pointer given by **env**. This function should return either a null pointer or a string that is the name of the supplied variable. If it returns a null pointer, a default name is generated based on the index of the variable. It is also legal for **naming_fn** to be null; in this case, default names are generated for all variables. The macro **bdd_naming_fn_none** is a null pointer of suitable type. **terminal_id_fn** should be a pointer to a function taking a **bdd_manager** and two longs, plus the pointer given by **env**. It should return either a null pointer or a string representing the MTBDD terminal represented by the given value. If it returns a null pointer, or if **terminal_id_fn** is null, then default names are generated for the terminals. The macro **bdd_terminal_id_fn_none** is a null pointer of suitable type.

void
bdd_print_profile(bddm, f, naming_fn, env, width, fp)
bdd_manager bddm;
bdd f;
char *(*naming_fn)();
pointer env;
int width;
FILE *fp;

Prints a node profile of a BDD in histogram form. The argument **naming_fn** should be as described in **bdd_print_bdd**. The width of the output stream is specified by **width**. This is used to determine how to scale the histogram.

void
bdd_print_profile_multiple(bddm, fs, naming_fn, env, width, fp)
bdd_manager bddm;
bdd *fs;
char *(*naming_fn)();
pointer env;
int width;
FILE *fp;

Prints a node profile of a set of BDDs, which should be given as a null-terminated array. The other arguments are as in **bdd_print_profile**.

void
bdd_print_function_profile(bddm, f, naming_fn, env, width, fp)
bdd_manager bddm;
bdd f;
char *(*naming_fn)();
pointer env;
int width;
FILE *fp;

Prints a function profile of a BDD in histogram form. The arguments are the same as those to **bdd_print_profile**.

int
bdd_dump_bdd(bddm, f, vars, fp)
bdd_manager bddm;
bdd f;
bdd *vars;
FILE *fp;

Writes an encoded description of the BDD or MTBDD **f** to the file given by **fp**. The argument **vars** should be a null-terminated array of variables that include the support of **f**. These variables

need not be in order of increasing index. The function returns a nonzero value if **f** was written to the file successfully.

bdd
bdd_undump_bdd(bddm, vars, fp, error)
bdd_manager bddm;
bdd *vars;
FILE *fp;
int *error;

Loads an encoded description of a BDD or MTBDD from the file given by **fp**. The argument **vars** should be a null-terminated array of variables that will become the support of the BDD. As in **bdd_dump_bdd**, these need not be in order of increasing index. If the same array of variables is used in dumping and undumping, the BDD returned will be equal to the one that was dumped. More generally, if the array **v1** is used when dumping, and the array **v2** is used when undumping, the BDD returned will be equal to the original BDD with the *i*th variable in **v2** substituted for the *i*th variable in **v1** for all *i*. Null is returned if the operation fails for some reason (node limit reached, I/O error, invalid file format, etc.). In this case, an error code is stored in **error**. The code will be one of the following.

Value	*Meaning*
BDD_UNDUMP_FORMAT	Invalid file format
BDD_UNDUMP_OVERFLOW	Node limit exceeded
BDD_UNDUMP_IOERROR	File I/O error
BDD_UNDUMP_EOF	Unexpected EOF

int
bdd_type(bddm, f)
bdd_manager bddm;
bdd f;

Returns an integer classifying the BDD or MTBDD **f**. The possible return values and their meanings are as follows.

Value	*Meaning*
BDD_TYPE_OVERFLOW	**f** is a null pointer
BDD_TYPE_ZERO	**f** is the constant FALSE
BDD_TYPE_ONE	**f** is the constant TRUE
BDD_TYPE_CONSTANT	**f** is an MTBDD constant
BDD_TYPE_POSVAR	**f** is a variable
BDD_TYPE_NEGVAR	**f** is the negation of a variable
BDD_TYPE_NONTERMINAL	**f** is not one of the above

void
bdd_free(bddm, f)
bdd_manager bddm;
bdd f;

Decreases the reference count of **f** by one. When the reference count of a BDD or MTBDD node reaches 0, the node and any of its children that are not otherwise referenced may eventually be garbage collected and reused. Intermediate results and unused BDDs and MTBDDs should be freed whenever possible. For example:

```
bdd
f_or_g_and_h(bddm, f, g, h)
    bdd_manager bddm;
    bdd f, g, h;
{
 bdd temp, result;
 temp=bdd_and(bddm, g, h);
 result=bdd_or(bddm, f, temp);
```

```
    bdd_free(bddm, temp);    /* Free intermediate */
    return (result);
}
```

void
bdd_unfree(bddm, f)
bdd_manager bddm;
bdd f;

Increases the reference count of **f** by one. This is usually used in conjunction with **bdd_clear_refs**. Works with MTBDDs.

void
bdd_clear_refs(bddm)
bdd_manager bddm;

Sets the reference counts of all BDD and MTBDD nodes (except for the node for TRUE/FALSE) to 0. Calling this routine and then immediately calling **bdd_unfree** on a set of BDDs has the effect of disposing of all BDDs except those in the set.

void
bdd_gc(bddm)
bdd_manager bddm;

Forces a BDD garbage collection; all nodes not reachable from a node with a nonzero reference count are disposed of. (Garbage collections also occur internally at various times.)

long
bdd_total_size(bddm)
bdd_manager bddm;

Returns the number of BDD and MTBDD nodes in existence (including those which are eligible for garbage collection).

long
bdd_vars(bddm)
bdd_manager bddm;

Returns the number of variables in existence.

int
bdd_cache_ratio(bddm, ratio)
bdd_manager bddm;
int ratio;

Sets the BDD operation cache size ratio to **ratio** and returns the old cache size ratio. The number of cache entries is constrained to be (roughly) less than the cache size ratio divided by 16 times the number of BDD nodes in existence. The default size ratio is 4, which gives about 1 cache entry per 4 BDD nodes. The amount of memory required per node will be about 17+(**ratio**/16)*20 bytes on a machine with 32-bit words.

void
bdd_node_limit(bddm, limit)
bdd_manager bddm;
long limit;

Sets the number of allowed BDD nodes to **limit** and returns the old limit. A value of 0 specifies no limit. If in the course of an operation, the number of nodes reaches the limit, an internal garbage collection takes place. If this does not free enough nodes to continue, the operation is aborted and a null value is returned. When dynamic reordering is used to shift around large variable block, this limit may be exceeded during reordering.

int
bdd_overflow(bddm)
bdd_manager bddm;

Returns 1 if any operation has caused an overflow in the number of nodes, and 0 otherwise.

Calling this routine clears the internal overflow flag, so subsequent calls will return 0 until the next overflow occurs.

void
bdd_overflow_closure(bddm, overflow_fn, overflow_env)
bdd_manager bddm;
void (*overflow_fn)();
pointer overflow_env;

Sets the closure to invoke when an overflow occurs. The function given by **overflow_fn** will be invoked as the last stage in the cleanup after the overflow. The function is passed the BDD manager and the pointer given by **overflow_env**. Typically, the function will jump to a user-provided error recovery routine.

void
bdd_abort_closure(bddm, abort_fn, abort_env)
bdd_manager bddm;
void (*abort_fn)();
pointer abort_env;

Sets a closure to invoke when the next node creation is attempted. All temporary results will be cleaned up just before the function given by **abort_fn** is called. The function is passed the BDD manager and the pointer given by **abort_env**. Typically, the function will jump to a user-provided error recovery routine. This functionality is intended to be used to cleanly interrupt BDD operations. Typically, **bdd_abort_closure** will be called within a signal handler.

void
bdd_stats(bddm, fp)
bdd_manager bddm;
FILE *fp;

Prints some statistics to the file given by **fp**.

void
bdd_dynamic_reordering(bddm, reorder_fn)
bdd_manager bddm;
void (*reorder_fn)();

Selects the method for dynamic reordering. When dynamic reordering is being used, the library may attempt to rearrange the BDD variable ordering in the midst of an operation so as to reduce the number of nodes in use. There are currently two available reordering methods. The first, **bdd_reorder_stable_window3**, permutes the variables within windows of three adjacent variables so as to minimize the overall BDD size. This process is repeated until no more reduction in size occurs. The second method, **bdd_reorder_sift**, moves each variable throughout the order to find an optimal position for that variable (assuming all other variables are fixed). This generally achieves greater size reductions than the window-based method, but is slower. The **reorder_fn** may also be **bdd_reorder_none** (an appropriately cast null pointer), in which case dynamic reordering is turned off. Also see the discussion on variable blocks in **bdd_new_var_block**.

void
bdd_reorder(bddm)
bdd_manager bddm;

Invoke the current dynamic reordering method.

block
bdd_new_var_block(bddm, v, n)
bdd_manager bddm;
bdd v;
long n;

Groups the variable **v** and the **n**-1 variables after it in the ordering into a single block for purposes of dynamic reordering. The purpose of blocks is to provide control over the possible orders that dynamic reordering will consider. In general, the variable blocks form a hierarchy. For example,

a block consisting of the variables with indexes 0 through 3 might be made up of two sub-blocks, one for the variables with index 0 and 1, and one for the variables with index 2 and 3. When dynamic reordering is invoked, it is actually applied to each block within the hierarchy. Reordering a block involves shuffling around the sub-blocks within it. Thus, dynamic reordering actually moves groups of variables rather than single variables. If you know that a group of variables should be together in the ordering, you should collect them together into a block. As an example, in BDD-based sequential verification algorithms, the variables representing the current state and next state of a state-holding element should generally be adjacent in a good ordering. By grouping these variables into a block, we can ensure that only orderings with this property are considered. After a block has been reordered, each sub-block within it is recursively reordered as well. You can also specify that certain blocks should not be reordered (see **bdd_var_block_reorderable** below).

void
bdd_var_block_reorderable(bddm, b, reorderable)
bdd_manager bddm;
block b;
int reorderable;

If **reorderable** is non-zero, turns on reordering for the given block, otherwise turns it off. By default, blocks are not reorderable. As an example, suppose we are building the BDDs representing a circuit with distinct control and data path. In such a case, we typically want to have the control variables at the top of the ordering. For the data path, we probably want to have the variables for each bit slice grouped together, and we want the bit slices to be ordered from most-significant to least-significant. However, we want to allow reordering within the control part and within each slice. To do this, we create the variables in the following order: control variables first, down to LSB slice variables. Then we create separate variable blocks for the control part and for each slice. We then turn on reordering for these blocks. Next, we create a block containing all of the variables, and we leave reordering off for this block. When dynamic reordering is invoked, it will rearrange the control variables and the variables within each slice, but will not move the control variables or the slices in relation to each other.

void
bdd_free_terminal_closure(bddm, free_terminal_fn, free_terminal_env)
bdd_manager bddm;
void (*free_terminal_fn)();
pointer free_terminal_env;

Sets a closure to invoke when freeing an MTBDD terminal node. The function receives the BDD manager, two longs representing the value of the terminal, and the pointer given by **free_terminal_env**. If you using the terminal value to hold pointers to other data structures, you can set up this routine to free those structures.

bdd
mtbdd_get_terminal(bddm, value1, value2)
bdd_manager bddm;
long value1;
long value2;

Creates an MTBDD terminal node corresponding to the value given by **value1** and **value2**. If a terminal node with the value already exists, its reference count is increased. See also **bdd_free_terminal_closure**.

void
mtbdd_terminal_value(bddm, f, value1, value2)
bdd_manager bddm;
bdd f;
long *value1;
long *value2;

f should be an MTBDD terminal node. The value of the node is stored in **value1** and **value2**.

bdd
mtbdd_ite(bddm, f, g, h)
bdd_manager bddm;
bdd f;
bdd g;
bdd h;

f should be a BDD and **g** and **h** should be MTBDDs. Returns the MTBDD for the operation IF **f** THEN **g** ELSE **h**.

bdd
mtbdd_substitute(bddm, f)
bdd_manager bddm;
bdd f;

Does the analog of **bdd_substitute** for the MTBDD **f**. The elements in the variable association must be BDDs.

bdd
mtbdd_equal(bddm, f, g)
bdd_manager bddm;
bdd f;
bdd g;

Returns the BDD which is true for those valuations on which the MTBDDs **f** and **g** are equal. That is, this is the analog of a logical XNOR for MTBDDs.

bdd
mtbdd_transform(bddm, f)
bdd_manager bddm;
bdd f;

Conceptually applies the user-defined transform to all terminals of the specified MTBDD. (This is actually done by just flipping the pointer flag, so this routine is really a macro for **bdd_not**.) See **mtbdd_transform_closure**.

void
mtbdd_transform_closure(bddm, canonical_fn, transform_fn, env)
bdd_manager bddm;
int (*canonical_fn)();
void (*transform_fn)();
pointer env;

Sets the MTBDD terminal transformation closure. Currently in the library, the pointer representing a boolean function and the pointer representing the negation of that function are identical except for the low-order bit. Complementing a function is done by simply toggling that bit. The MTBDD terminal transformation allows this mechanism to be extended to MTBDDs. Whenever a terminal is created, **canonical_fn** will be called. It is passed the BDD manager, two longs representing the terminal being created, and the pointer given by **env**. The function should return zero if the value is already canonical, and a non-zero result if it needs to be transformed. If the value needs to be transformed, then **transform_fn** will be called, with the BDD manager, two longs representing the value to be transformed, pointers to two longs to hold the result, and the pointer given by **env**. The actual terminal node that is created will contain the transformed value. The original terminal requested will be represented by a pointer to this node, with the low-order bit of the pointer set. Also see **mtbdd_one_data**. If you are going to call this function, you should do it before creating any MTBDD terminals.

void
mtbdd_one_data(bddm, value1, value2)
bdd_manager bddm;
long value1;
long value2;

If you are planning to use MTBDDs that contain TRUE and FALSE as well as other values, you may need to use this function to set the MTBDD value for the node representing TRUE. In this case, also keep in mind that the when the transformation function is applied to this value, it should yield the value that you want for FALSE. Also, the value for TRUE should be regarding as canonical, i.e., TRUE must be represented by a pointer with the low-order bit cleared. As an example, suppose that we are planning to use MTBDDs to represent spectral transforms of boolean functions [4]. In this case, the MTBDD terminal values will conceptually be integers. Further, it is convenient for TRUE to be represented by the value -1, and FALSE to be represented by +1. We will represent terminal values using two longs, with the first long representing the most-significant part of the integer. We will also assume a 2's complement representation, so TRUE should be represented by the data values -1 and -1. Since the value for FALSE is the negation of that for TRUE, we will have our transform function represent integer negation. Also, since we want the value for TRUE to be canonical, we will regard nonnegative values as canonical. Thus, we define

```
int
canonical_fn(bddm, value1, value2, env)
    bdd_manager bddm;
    long value1;
    long value2;
    pointer env;
{
 return (value1 > 0 || (!value1 && value2 > 0));
}

void
transform_fn(bddm, value1, value2, result1, result2, env)
    bdd_manager bddm;
    long value1;
    long value2;
    long *result1;
    long *result2;
    pointer env;
{
 if (!value2)
   /* Will be a carry when taking 2's complement of value2.  Thus, */
   /* take 2's complement of high part. */
   value1= -value1;
 else
   {
    value2= -value2;
    value1= ~value1;
   }
 *result1=value1;
 *result2=value2;
}
```

We then call **mtbdd_transform_closure** to register these functions, and use

```
bdd_one_data(bddm, -1l, -1l);
```

to set the value for TRUE to -1. (The default canonical checking and transformation functions and the default MTBDD values for TRUE and FALSE are actually as given in this example.) If you are going to call **bdd_one_data**, you should do it before creating any MTBDD terminals.

ADDING NEW ROUTINES

If you want to add new routines to the library, you would be well-advised to look at some of the existing ones to get a feel for how they operate. Good ones include **bdd_ite** (the basic logical operation) and **bdd_exists** (a routine using variable associations). Some basic points are explained below. To get the declarations of the internal library data structures and routines, you should **#include <bddint.h>** instead of using **bdduser.h**. You will probably want to study this file to become familiar with the data structures.

Pointers to BDD nodes and cache entries are tagged using the low three bits of the pointer. Because of this, all structures must be aligned on eight byte boundaries. The storage allocation routines guarantee this alignment. The tag field of a tagged pointer is extracted with the **TAG** macro. The **POINTER** macro masks off the tag to get the actual pointer. If the pointer is a pointer to a BDD node, you can use **BDD_POINTER** instead; this just casts the result to a **bdd** after masking off the tag. The tag can be set using **SET_TAG**, and individual tag bits can be manipulated with **TAG0**, **FLIP_TAG0** and **SET_TAG0** for tag bit 0, and the analogous macros for tag bits 1 and 2. More commonly, slightly higher level macros are used for manipulating tags. For BDD nodes, there is only one tag bit that is actually used. When it is set, it indicates the pointer should be interpreted as representing the complement of the node that it points to. (Or for MTBDDs, that it should be interpreted as transformed using the user-definable transformation function). There are macros for testing, clearing, and flipping the negation flag.

Before using the macros below on a pointer **f**, you need to use **BDD_SETUP(f)**. This actually declares a new variable to hold the masked pointer **BDD_POINTER(f)**. Hence, it needs to be placed at some point where a variable declaration could legally go. If you change **f**, you can reset this internal variable using **BDD_RESET**.

BDD pointers are generally manipulated using the following macros. Below, ‘‘node’’ refers to the node referenced by the pointer.

BDD_IS_CONST
Tests if the node represents the constant TRUE or FALSE or an MTBDD terminal node.

BDD_INDEX
Returns the index of the node, or **BDD_MAX_INDEX** if given a constant node.

BDD_INDEXINDEX
Returns the index index of a node. This field is the value returned by **bdd_if_id** and is invariant; when you create a new variable, the index of old nodes may change, but the index index stays the same. When you call **bdd_find**, you pass the desired index index of the new node, not the index.

BDD_NOT
Flips the negation flag on a pointer.

BDD_THEN, BDD_ELSE
Return the THEN and ELSE pointers of a node, taking proper account of pointer flags. These are used for doing Shannon expansions on a node.

BDD_TOP_VAR2 Takes a **bdd_manager**, a variable that can hold an index index, and two **bdd**s. Sets the index index variable to the index index of the variable with the lowest index among the variables at the roots of the BDDs. This index index can then be used with...

BDD_COFACTOR Takes an index index, a BDD, and two variables of type **bdd**, and sets the two variables either to the original BDD or to the cofactors of the original BDD with respect to its top variable, depending on whether the index index of the first BDD matches that specified. You can do a Shannon expansion on the top variable of two BDDs by using **BDD_TOP_VAR2** to get the index index of the highest variable and then using **BDD_COFACTOR** to take the appropriate cofactors.

BDD_MARK
Accesses the mark field of a node. This expands to a l-value, so you can set the mark with this as well. (But see BDD_TEMP_REFS below.)

BDD_ONE, BDD_ZERO

Take a BDD manager and give back the BDDs for TRUE and FALSE.

BDD_REFS
Accesses the reference count field of a node.

BDD_INCREFS, BDD_DECREFS
Increment and decrement the reference count.

BDD_TEMP_REFS
Accesses the temporary reference count field of a node. The temporary reference count and the mark actually share storage, so you can't use both at once! That is, unless you are very clever, you can't write a routine that builds temporary nodes and uses the marks.

BDD_TEMP_INCREFS, BDD_TEMP_DECREFS
Increment and decrement the temporary reference count.

New BDD nodes are created using **bdd_find**. This routine takes a BDD manager, an index index, and two subBDDs as arguments. New MTBDD terminals can be created with **bdd_find_terminal**. The result cache is manipulated using the **bdd_lookup_in_cache** and **bdd_insert_in_cache** routines. There are different versions of these routines depending on exactly what is being cached. The basic ones are **bdd_lookup_in_cache31** and **bdd_insert_in_cache31**. The first of these takes a cache entry type (CACHE_TYPE_ITE, CACHE_TYPE_TWO, etc.), three arguments of unspecified type (passed as longs), and a pointer to an unspecified type of result (a pointer to a long). It returns a nonzero result if the lookup succeeds. The corresponding insert routine is similar except that the result is passed in as a long, and nothing is returned. There are similar functions that are for routines that take two arguments and return two results (or a single double-word result), or for routines that take one argument and return three results. There are also macros such as **bdd_lookup_in_cache2** that are wrappers for things like two-argument functions, etc. In general, some action must be taken when results are returned from the cache, when entries are purged from the cache, when entries are garbage collected, and when a variable association ID is reclaimed. For the built-in cache entry types, these actions are done automatically. For example, when a BDD is returned from an entry with CACHE_TYPE_TWO, the temporary reference count of the BDD is incremented. Some of the entry types are available for customization. The actions to take for these entry types are specified by calling **bdd_cache_functions**. This function takes a BDD manager, an integer between 1 and 3 specifying the number of arguments you want to cache on, and four function pointers. When returning a result, purging an entry, garbage collecting, or reclaiming an association ID, these functions are called. The first three functions are passed the BDD manager and the entry. (The tag bits will have already been masked off the entry pointer.) The last receives these plus the association ID being freed (cast to a pointer). The garbage collection function should return a nonzero result if the entry should be garbage collected. If the entry contains some BDD nodes, they should be tested with **BDD_IS_USED**. The function called when an association ID is reclaimed should return a nonzero result if the entry should be flushed from the cache. This function and the purge function and return functions may be null, specifying that no action need be taken. **bdd_cache_functions** returns an integer that represents a tag to use with the cache insertion and lookup routines, or -1 if there are no more free tags available. The routine **bdd_free_cache_tag** makes a tag available again.

Routines that build new BDD nodes must take into account the possibility of running into the node limit. The package is set up to make this easy if you use the following strategy. Organize your routine as a top-level (user-callable) procedure and an internal procedure for performing the actual computation. The top-level procedure should check its arguments before calling the internal routine. The **bdd_check_arguments** function can be used to test for null arguments (indicating a prior overflow) or arguments with a zero reference count (indicating a bug). It should also use the **FIREWALL** macro to set up an overflow trap. The internal routine should use temporary reference counts to keep track of the nodes it is using. When a node is returned from the internal routine, increment its temporary reference count once. (You don't have to do this for the constants or for variables, since they can't be garbage collected.) When you pass a node to **bdd_find**, its temporary reference count is decremented once automatically, and its reference count is incremented. Also, the result of **bdd_find** has its temporary reference count incremented once

automatically. Hence, if you your routine has the standard organization (Shannon's expansion followed by **bdd_find** on the subresults), you usually don't have to worry about incrementing or decrementing the reference counts yourself. If you don't use a subresult, or if you want a subresult to stick around after calling **bdd_find**, you'll have to do the appropriate twiddling. When the internal routine finally returns, you should have a BDD with a single temporary reference count. Use **RETURN_BDD** to convert this temporary reference count to an external one and return the result to the user. If you follow this strategy, you won't have to deal with overflow; when the node limit is reached, **bdd_find** will try garbage collecting, and if that doesn't work, will call the overflow trap set up by **FIREWALL**. The overflow trap handler will automatically zero all temporary reference counts and return a null pointer to the user. Note: if you want to call other routines, such as the IF-THEN-ELSE routine, within your internal procedure, you should call the internal procedure for the routine. That way, the overflow handler will give control back to the user if the routine you are calling causes an overflow.

A typical routine looks like:

```
bdd
foo_step(bddm, f, g)
    bdd_manager bddm;
    bdd f, g;
{
 bdd_indexindex_type top_indexindex;
 bdd f1, f2;
 bdd g1, g2;
 bdd temp1, temp2;
 bdd result;

 BDD_SETUP(f);
 BDD_SETUP(g);
 if (<terminal case>)
   {
    BDD_TEMP_INCREFS(f);
    return (f);
   }
 if (bdd_lookup_in_cache2(bddm, <op>, f, g, &result))
  return (result);
 BDD_TOP_VAR2(top_indexindex, bddm, f, g);
 BDD_COFACTOR(top_indexindex, f, f1, f2);
 BDD_COFACTOR(top_indexindex, g, g1, g2);
 temp1=foo_step(bddm, f1, g1);
 temp2=foo_step(bddm, f2, g2);
 result=bdd_find(bddm, top_indexindex, temp1, temp2);
 bdd_insert_in_cache2(bddm, <op>, f, g, result);
 return (result);
}

bdd
foo(bddm, f, g)
    bdd_manager bddm;
    bdd f, g;
{
 if (bdd_check_arguments(2, f, g))
   {
    FIREWALL(bddm);
    RETURN_BDD(foo_step(bddm, f, g));
```

```
    }
  return ((bdd)0);
}
```

In the case of dynamic variable reordering, the same abort mechanism is used. After reordering, all reference counts are reset to their original values and the operation is retried. This is handled automatically by the FIREWALL macro. (The operation is aborted since after reordering, the implicit ordering represented in the C subroutine call stack may be different from the new variable order. Reordering occurs before freeing the temporaries, since we want to minimize the aggregate size of the operands plus the result that is being constructed.)

Storage can be allocated through a number of mechanisms. The routines **mem_get_block**, **mem_free_block**, and **mem_resize_block** are generally used for large single items. For smaller uniformly sized items, you probably should use a record manager. **mem_new_rec_mgr** will return a record manager that handles blocks of a given size. Use **mem_new_rec** and **mem_free_rec** to obtain and free individual records. Finally, **mem_free_rec_mgr** will dispose of the record manager and all of its associated records. These routines are documented in more detail in the storage management library man page. If your structures are at most 64 bytes in size, you can use the macros **BDD_NEW_REC** and **BDD_FREE_REC**. These obtain records from the internal BDD record managers.

PORTABILITY NOTES

Since pointer tagging is heavily used, you'll have major problems if you can't cast back and forth between pointers and longs without losing something. The low-level storage management routines are fairly UNIX specific; they call **sbrk** directly. If you don't have something similar, you may have to rewrite them. The storage management routines also need to be able to move and clear blocks of memory whose size is given by a long. You may have to fiddle with these, especially if you have a machine where int and long are different. If you encounter portability problems, let me know; maybe the next release will be able to accommodate your machine.

SEE ALSO

mem(3)

BUGS

Surely you're joking.

REFERENCES

[1] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677-691, August 1986.

[2] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proceedings of the 1990 IEEE International Conference on Computer-Aided Design*, November, 1990.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, June, 1990.

[4] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. C.-Y. Yang. Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, June, 1993.

AUTHOR

David E. Long
long@research.att.com