CUDD: CU Decision Diagram Package
Release 2.5.0

Fabio Somenzi
Department of Electrical, Computer, and Energy Engineering
University of Colorado at Boulder
<Fabio@Colorado.EDU>

February 4, 2012

Contents

1 Introduction

The CUDD package provides functions to manipulate Binary Decision Diagrams (BDDs) [5, 3], Algebraic Decision Diagrams (ADDs) [1], and Zero-suppressed Binary Decision Diagrams (ZDDs) [12]. BDDs are used to represent switching functions; ADDs are used to represent function from $\{0, 1\}^n$ to an arbitrary set. ZDDs represent switching functions like BDDs; however, they are much more efficient than BDDs when the functions to be represented are characteristic functions of cube sets, or in general, when the ON-set of the function to be represented is very sparse. They are inferior to BDDs in other cases.

The package provides a large set of operations on BDDs, ADDs, and ZDDs, functions to convert BDDs into ADDs or ZDDs and vice versa, and a large assortment of variable reordering methods.

The CUDD package can be used in three ways:

- As a black box. In this case, the application program that needs to manipulate decision diagrams only uses the exported functions of the package. The rich set of functions included in the CUDD package allows many applications to be written in this way. Section 3 describes how to use the exported functions of the package. An application written in terms of the exported functions of the package needs not concern itself with the details of variable reordering, which may take place behind the scenes.

- As a clear box. When writing a sophisticated application based on decision diagrams, efficiency often dictates that some functions be implemented as direct recursive manipulation of the diagrams, instead of being written in terms of existing primitive functions. Section 4 explains how to add new functions to the CUDD package. It also details how to write a recursive function that can be interrupted by dynamic variable reordering.

- Through an interface. Object-oriented languages like C++ and Perl5 can free the programmer from the burden of memory management. A C++ interface is included in the distribution of CUDD. It automatically frees decision diagrams that are no longer used by the application and overloads operators. Almost all the functionality provided by the CUDD exported functions is available through the C++ interface, which is especially recommended for fast prototyping. Section 5 explains how to use the interface. A Perl5 interface also exists and is ditributed separately. (See Section 2.2.) Some applications define their own interfaces. See for example Section 3.18.

In the following, the reader is supposed to be familiar with the basic ideas about decision diagrams, as found, for instance, in [3].

2 How to Get CUDD

2.1 The CUDD Package

The CUDD package is available via anonymous FTP from vlsi.Colorado.EDU. A compressed tar file named `cudd-2.5.0.tar.gz` can be found in directory `pub`. Once you have this file,

```
gzip -dc cudd-2.5.0.tar.gz | tar xvf -
```

will create directory `cudd-2.5.0` and its subdirectories. These directories contain the decision diagram package, a few support libraries, and a toy application based on the decision diagram package. There is a README file with instructions on configuration and installation in `cudd-2.5.0`. You can use a compiler for either ANSI C or C++.

Once you have made the libraries and program, you can type:

```
cd nanotrav
nanotrav -p 1 -autodyn -reordering sifting -trav
mult32a.blif
```

This will run a simple-minded FSM traversal program. (On a 2.4 GHz Pentium 4 (TM), it takes about 0.5 s.) The output produced by the program can be checked against `cudd-2.5.0/nanotrav/mult32a.out`. More information on the `nanotrav` program can be found in `cudd-2.5.0/nanotrav/README`.

If you want to be notified of new releases of the CUDD package, send a message to `Fabio@Colorado.EDU`.

2.2 CUDD Friends

Two CUDD extensions are available via anonymous FTP from vlsi.Colorado.EDU.

- *PerlDD* is an object-oriented Perl5 interface to CUDD. It is organized as a standard Perl extension module. The Perl interface is at a somewhat higher level than the C++ interface, but it is not as complete.

- *DDcal* is a graphic BDD calculator based on CUDD, Perl-Tk, and dot. (See Section 3.19 for information on *dot*.)

3 User's Manual

This section describes the use of the CUDD package as a black box.

3.1 Compiling and Linking

To build an application that uses the CUDD package, you should add

```
#include "util.h"
#include "cudd.h"
```

to your source files, and should link `libcudd.a`, `libmtr.a`, `libst.a`, and `libutil.a` to your executable. (All these libraries are part of the distribution.) Some platforms require specific compiler and linker flags. Refer to the `Makefile` in the top level directory of the distribution.

Keep in mind that whatever flags affect the size of data structures—for instance the flags used to use 64-bit pointers where available—must be specified when compiling both CUDD and the files that include its header files.

3.2 Basic Data Structures

3.2.1 Nodes

BDDs, ADDs, and ZDDs are made of DdNode's. A DdNode (node for short) is a structure with several fields. Those that are of interest to the application that uses the CUDD package as a black box are the variable index, the reference count, and the value. The remaining fields are pointers that connect nodes among themselves and that are used to implement the unique table. (See Section 3.2.2.)

The *index* field holds the name of the variable that labels the node. The index of a variable is a permanent attribute that reflects the order of creation. Index 0 corresponds to the variable created first. On a machine with 32-bit pointers, the maximum number of variables is the largest value that can be stored in an unsigned short integer minus 1. The largest index is reserved for the constant nodes. When 64-bit pointers are used, the maximum number of variables is the largest value that can be stored in an unsigned integer minus 1.

When variables are reordered to reduce the size of the decision diagrams, the variables may shift in the order, but they retain their indices. The package keeps track of the variable permutation (and its inverse). The application is not affected by variable reordering, except in the following cases.

- If the application uses generators (*Cudd_ForeachCube* and *Cudd_ForeachNode*) and reordering is enabled, then it must take care not to call any operation that may create new nodes (and hence possibly trigger reordering). This is because the cubes (i.e., paths) and nodes of a diagram change as a result of reordering.

- If the application uses *Cudd_bddConstrain* and reordering takes place, then the property of *Cudd_bddConstrain* of being an image restrictor is lost.

The CUDD package relies on garbage collection to reclaim the memory used by diagrams that are no longer in use. The scheme employed for garbage collection is based on keeping a reference count for each node. The references that are counted are both the internal references (references from other nodes) and external references (typically references from the calling environment). When an application creates a new BDD, ADD, or ZDD, it must increase its reference count explicitly, through a call to *Cudd_Ref*. Similarly, when a diagram is no longer needed, the application must call *Cudd_RecursiveDeref* (for BDDs and ADDs) or *Cudd_RecursiveDerefZdd* (for ZDDs) to "recycle" the nodes of the diagram.

Terminal nodes carry a value. This is especially important for ADDs. By default, the value is a double. To change to something different (e.g., an integer), the package must be modified and recompiled. Support for this process is currently very rudimentary.

3.2.2 The Manager

All nodes used in BDDs, ADDs, and ZDDs are kept in special hash tables called the *unique tables*. Specifically, BDDs and ADDs share the same unique table, whereas ZDDs have their own table. As the name implies, the main purpose of the unique table is to guarantee that each node is unique; that is, there is no other node labeled by the same variable and with the same children. This uniqueness property makes decision diagrams canonical. The unique tables and some auxiliary data structures make up the DdManager (manager for short). Though the application that uses only the exported functions needs not be concerned with most details of the manager, it has to deal with the manager in the following sense. The application must initialize the manager by calling an appropriate function. (See Section 3.3.) Subsequently, it must pass a pointer to the manager to all the functions that operate on decision diagrams.

With the exception of a few statistical counters, there are no global variables in the CUDD package. Therefore, it is quite possible to have multiple managers simultaneously active in the same application.[1] It is the pointers to the managers that tell the functions on what data they should operate.

[1]The global statistical counters are used locally; hence they are compatible with the use of multiple managers.

3.2.3 Cache

Efficient recursive manipulation of decision diagrams requires the use of a table to store computed results. This table is called here the *cache* because it is effectively handled like a cache of variable but limited capacity. The CUDD package starts by default with a small cache, and increases its size until either no further benefit is achieved, or a limit size is reached. The user can influence this policy by choosing initial and limit values for the cache size.

Too small a cache will cause frequent overwriting of useful results. Too large a cache will cause overhead, because the whole cache is scanned every time garbage collection takes place. The optimal parameters depend on the specific application. The default parameters work reasonably well for a large spectrum of applications.

The cache of the CUDD package is used by most recursive functions of the package, and can be used by user-supplied functions as well. (See Section 4.4.)

3.3 Initializing and Shutting Down a DdManager

To use the functions in the CUDD package, one has first to initialize the package itself by calling *Cudd_Init*. This function takes four parameters:

- numVars: It is the initial number of variables for BDDs and ADDs. If the total number of variables needed by the application is known, then it is slightly more efficient to create a manager with that number of variables. If the number is unknown, it can be set to 0, or to any other lower bound on the number of variables. Requesting more variables than are actually needed is not incorrect, but is not efficient.

- numVarsZ: It is the initial number of variables for ZDDs. See Sections 3.9 and 3.11 for a discussion of the value of this argument.

- numSlots: Determines the initial size of each subtable of the unique table. There is a subtable for each variable. The size of each subtable is dynamically adjusted to reflect the number of nodes. It is normally O.K. to use the default value for this parameter, which is CUDD_UNIQUE_SLOTS.

- cacheSize: It is the initial size (number of entries) of the cache. Its default value is CUDD_CACHE_SLOTS.

- maxMemory: It is the target value for the maximum memory occupation (in bytes). The package uses this value to decide two parameters.

  - the maximum size to which the cache will grow, regardless of the hit rate or the size of the unique table.

- the maximum size to which growth of the unique table will be preferred to garbage collection.

If maxMemory is set to 0, CUDD tries to guess a good value based on the available memory.

A typical call to *Cudd_Init* may look like this:

```
manager = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0);
```

To reclaim all the memory associated with a manager, an application must call *Cudd_Quit*. This is normally done before exiting.

3.4 Setting Parameters

The package provides several functions to set the parameters that control various functions. For instance, the package has an automatic way of determining whether a larger unique table would make the application run faster. In that case, the package enters a "fast growth" mode in which resizing of the unique subtables is favored over garbage collection. When the unique table reaches a given size, however, the package returns to the normal "slow growth" mode, even though the conditions that caused the transition to fast growth still prevail. The limit size for fast growth can be read by *Cudd_ReadLooseUpTo* and changed by *Cudd_SetLooseUpTo*. Similar pairs of functions exist for several other parameters. See also Section 4.8.

3.5 Constant Functions

The CUDD Package defines several constant functions. These functions are created when the manager is initialized, and are accessible through the manager itself.

3.5.1 One, Logic Zero, and Arithmetic Zero

The constant 1 (returned by *Cudd_ReadOne*) is common to BDDs, ADDs, and ZDDs. However, its meaning is different for ADDs and BDDs, on the one hand, and ZDDs, on the other hand. The diagram consisting of the constant 1 node only represents the constant 1 function for ADDs and BDDs. For ZDDs, its meaning depends on the number of variables: It is the conjunction of the complements of all variables. Conversely, the representation of the constant 1 function depends on the number of variables. The constant 1 function of n variables is returned by *Cudd_ReadZddOne*.

The constant 0 is common to ADDs and ZDDs, but not to BDDs. The BDD logic 0 is **not** associated with the constant 0 function: It is obtained by complementation (*Cudd_Not*) of the constant 1. (It is also returned by *Cudd_ReadLogicZero*.) All other constants are specific to ADDs.

3.5.2 Predefined Constants

Besides 0 (returned by *Cudd_ReadZero*) and 1, the following constant functions are created at initialization time.

1. PlusInfinity and MinusInfinity: On computers implementing the IEEE standard 754 for floating-point arithmetic, these two constants are set to the signed infinities. On the DEC Alphas, the option `-ieee_with_no_inexact` or `-ieee_with_inexact` must be passed to the DEC compiler to get support of the IEEE standard. (The compiler still produces a warning, but it can be ignored.) Compiling with those options may cause substantial performance degradation on the Evolution IV CPUs. (Especially if the application does use the infinities.) The problem is reportedly solved in the Evolution V CPUs. If `gcc` is used to compile CUDD on the Alphas, the symbol `HAVE_IEEE_754` must be undefined. (See the Makefile for the details.) The values of these constants are returned by *Cudd_ReadPlusInfinity* and *Cudd_ReadMinusInfinity*.

2. Epsilon: This constant, initially set to 10^{-12}, is used in comparing floating point values for equality. Its value is returned by *Cudd_ReadEpsilon*, and it can be modified by calling *Cudd_SetEpsilon*. Unlike the other constants, it does not correspond to a node.

3.5.3 Background

The background value is a constant typically used to represent non-existing arcs in graphs. Consider a shortest path problem. Two nodes that are not connected by an arc can be regarded as being joined by an arc of infinite length. In shortest path problems, it is therefore convenient to set the background value to PlusInfinity. In network flow problems, on the other hand, two nodes not connected by an arc can be regarded as joined by an arc of 0 capacity. For these problems, therefore, it is more convenient to set the background value to 0. In general, when representing sparse matrices, the background value is the value that is assumed implicitly.

At initialization, the background value is set to 0. It can be read with *Cudd_ReadBackground*, and modified with *Cudd_SetBackground*. The background value affects procedures that read sparse matrices/graphs (*Cudd_addRead* and *Cudd_addHarwell*), procedures that print out sum-of-product expressions for ADDs (*Cudd_PrintMinterm*), generators of cubes (*Cudd_ForeachCube*), and procedures that count minterms (*Cudd_CountMinterm*).

3.5.4 New Constants

New constant can be created by calling *Cudd_addConst*. This function will retrieve the ADD for the desired constant, if it already exist, or it will create a new one. Obviously, new constants should only be used when manipulating ADDs.

3.6 Creating Variables

Decision diagrams are typically created by combining simpler decision diagrams. The simplest decision diagrams, of course, cannot be created in that way. Constant functions have been discussed in Section 3.5. In this section we discuss the simple variable functions, also known as *projection functions*.

3.6.1 New BDD and ADD Variables

The projection functions are distinct for BDDs and ADDs. A projection function for BDDs consists of an internal node with both outgoing arcs pointing to the constant 1. The *else* arc is complemented.

An ADD projection function, on the other hand, has the *else* pointer directed to the arithmetic zero function. One should never mix the two types of variables. BDD variables should be used when manipulating BDDs, and ADD variables should be used when manipulating ADDs. Three functions are provided to create BDD variables:

- *Cudd_bddIthVar*: Returns the projection function with index i. If the function does not exist, it is created.
- *Cudd_bddNewVar*: Returns a new projection function, whose index is the largest index in use at the time of the call, plus 1.
- *Cudd_bddNewVarAtLevel*: Similar to *Cudd_bddNewVar*. In addition it allows to specify the position in the variable order at which the new variable should be inserted. In contrast, *Cudd_bddNewVar* adds the new variable at the end of the order.

The analogous functions for ADDs are *Cudd_addIthVar*, *Cudd_addNewVar*, and *Cudd_addNewVarAtLevel*.

3.6.2 New ZDD Variables

Unlike the projection functions of BDDs and ADDs, the projection functions of ZDDs have diagrams with $n+1$ nodes, where n is the number of variables. Therefore the ZDDs of the projection functions change when new variables are added. This will be discussed in Section 3.9. Here we assume that the number of variables is fixed. The ZDD of the i-th projection function is returned by *Cudd_zddIthVar*.

3.7 Basic BDD Manipulation

Common manipulations of BDDs can be accomplished by calling *Cudd_bddIte*. This function takes three BDDs, f, g, and h, as arguments and computes $f \cdot g + f' \cdot h$. Like all the functions that create new BDDs or ADDs, *Cudd_bddIte* returns a result that must be explicitly referenced by the caller. *Cudd_bddIte* can be used to implement all two-argument boolean functions. However, the package also provides *Cudd_bddAnd* as well as the other two-operand boolean functions, which are slightly more efficient when a two-operand function is called for. The following fragment of code illustrates how to build the BDD for the function $f = x_0' x_1' x_2' x_3'$.

```
DdManager *manager;
DdNode *f, *var, *tmp;
int i;

...

f = Cudd_ReadOne(manager);
Cudd_Ref(f);
for (i = 3; i >= 0; i--) {
    var = Cudd_bddIthVar(manager,i);
    tmp = Cudd_bddAnd(manager,Cudd_Not(var),f);
    Cudd_Ref(tmp);
    Cudd_RecursiveDeref(manager,f);
    f = tmp;
}
```

This example illustrates the following points:

- Intermediate results must be "referenced" and "dereferenced." However, `var` is a projection function, and its reference count is always greater than 0. Therefore, there is no call to *Cudd_Ref*.

- The new `f` must be assigned to a temporary variable (`tmp` in this example). If the result of *Cudd_bddAnd* were assigned directly to `f`, the old `f` would be lost, and there would be no way to free its nodes.

- The statement `f = tmp` has the same effect as:

  ```
  f = tmp;
  Cudd_Ref(f);
  Cudd_RecursiveDeref(manager,tmp);
  ```

  but is more efficient. The reference is "passed" from `tmp` to `f`, and `tmp` is now ready to be reutilized.

- It is normally more efficient to build BDDs "bottom-up." This is why the loop goes from 3 to 0. Notice, however, that after variable reordering, higher index does not necessarily mean "closer to the bottom." Of course, in this simple example, efficiency is not a concern.

- Had we wanted to conjoin the variables in a bottom-up fashion even after reordering, we should have used *Cudd_ReadInvPerm*. One has to be careful, though, to fix the order of conjunction before entering the loop. Otherwise, if reordering takes place, it is possible to use one variable twice and skip another variable.

3.8 Basic ADD Manipulation

The most common way to manipulate ADDs is via *Cudd_addApply*. This function can apply a wide variety of operators to a pair of ADDs. Among the available operators are addition, multiplication, division, minimum, maximum, and boolean operators that work on ADDs whose leaves are restricted to 0 and 1 (0-1 ADDs).

The following fragment of code illustrates how to build the ADD for the function $f = 5x_0x_1x_2x_3$.

```
DdManager *manager;
DdNode *f, *var, *tmp;
int i;

...

f = Cudd_addConst(manager,5);
Cudd_Ref(f);
for (i = 3; i >= 0; i--) {
```

```
    var = Cudd_addIthVar(manager,i);
    Cudd_Ref(var);
    tmp = Cudd_addApply(manager,Cudd_addTimes,var,f);
    Cudd_Ref(tmp);
    Cudd_RecursiveDeref(manager,f);
    Cudd_RecursiveDeref(manager,var);
    f = tmp;
}
```

This example, contrasted to the example of BDD manipulation, illustrates the following points:

- The ADD projection function are not maintained by the manager. It is therefore necessary to reference and dereference them.
- The product of two ADDs is computed by calling *Cudd_addApply* with *Cudd_addTimes* as parameter. There is no "apply" function for BDDs, because *Cudd_bddAnd* and *Cudd_bddXor* plus complementation are sufficient to implement all two-argument boolean functions.

3.9 Basic ZDD Manipulation

ZDDs are often generated by converting existing BDDs. (See Section 3.11.) However, it is also possible to build ZDDs by applying boolean operators to other ZDDs, starting from constants and projection functions. The following fragment of code illustrates how to build the ZDD for the function $f = x_0' + x_1' + x_2' + x_3'$. We assume that the four variables already exist in the manager when the ZDD for f is built. Note the use of De Morgan's law.

```
DdManager *manager;
DdNode *f, *var, *tmp;
int i;

manager = Cudd_Init(0,4,CUDD_UNIQUE_SLOTS,
    CUDD_CACHE_SLOTS,0);
...

tmp = Cudd_ReadZddOne(manager,0);
Cudd_Ref(tmp);
for (i = 3; i >= 0; i--) {
    var = Cudd_zddIthVar(manager,i);
    Cudd_Ref(var);
```

```
    f = Cudd_zddIntersect(manager,var,tmp);
    Cudd_Ref(f);
    Cudd_RecursiveDerefZdd(manager,tmp);
    Cudd_RecursiveDerefZdd(manager,var);
    tmp = f;
}
f = Cudd_zddDiff(manager,Cudd_ReadZddOne(manager,0),tmp);
Cudd_Ref(f);
Cudd_RecursiveDerefZdd(manager,tmp);
```

This example illustrates the following points:

- The projection functions are referenced, because they are not maintained by the manager.
- Complementation is obtained by subtracting from the constant 1 function.
- The result of *Cudd_ReadZddOne* does not require referencing.

CUDD provides functions for the manipulation of covers represented by ZDDs. For instance, *Cudd_zddIsop* builds a ZDD representing an irredundant sum of products for the incompletely specified function defined by the two BDDs L and U. *Cudd_zddWeakDiv* performs the weak division of two covers given as ZDDs. These functions expect the two ZDD variables corresponding to the two literals of the function variable to be adjacent. One has to create variable groups (see Section 3.14) for reordering of the ZDD variables to work. BDD automatic reordering is safe even without groups: If realignment of ZDD and ADD/BDD variables is requested (see Section 3.15) groups will be kept adjacent.

3.10 Converting ADDs to BDDs and Vice Versa

Several procedures are provided to convert ADDs to BDDs, according to different criteria. (*Cudd_addBddPattern*, *Cudd_addBddInterval*, and *Cudd_addBddThreshold*.) The conversion from BDDs to ADDs (*Cudd_BddToAdd*) is based on the simple principle of mapping the logical 0 and 1 on the arithmetic 0 and 1. It is also possible to convert an ADD with integer values (more precisely, floating point numbers with 0 fractional part) to an array of BDDs by repeatedly calling *Cudd_addIthBit*.

3.11 Converting BDDs to ZDDs and Vice Versa

Many applications first build a set of BDDs and then derive ZDDs from the BDDs. These applications should create the manager with 0 ZDD variables and create the

BDDs. Then they should call *Cudd_zddVarsFromBddVars* to create the necessary ZDD variables—whose number is likely to be known once the BDDs are available. This approach eliminates the difficulties that arise when the number of ZDD variables changes while ZDDs are being built.

The simplest conversion from BDDs to ZDDs is a simple change of representation, which preserves the functions. Simply put, given a BDD for f, a ZDD for f is requested. In this case the correspondence between the BDD variables and ZDD variables is one-to-one. Hence, *Cudd_zddVarsFromBddVars* should be called with the *multiplicity* parameter equal to 1. The conversion proper can then be performed by calling *Cudd_zddPortFromBdd*. The inverse transformation is performed by *Cudd_zddPortToBdd*.

ZDDs are quite often used for the representation of *covers*. This is normally done by associating two ZDD variables to each variable of the function. (And hence, typically, to each BDD variable.) One ZDD variable is associated with the positive literal of the BDD variable, while the other ZDD variable is associated with the negative literal. A call to *Cudd_zddVarsFromBddVars* with *multiplicity* equal to 2 will associate to BDD variable i the two ZDD variables $2i$ and $2i + 1$.

If a BDD variable group tree exists when *Cudd_zddVarsFromBddVars* is called (see Section 3.13) the function generates a ZDD variable group tree consistent to it. In any case, all the ZDD variables derived from the same BDD variable are clustered into a group.

If the ZDD for f is created and later a new ZDD variable is added to the manager, the function represented by the existing ZDD changes. Suppose, for instance, that two variables are initially created, and that the ZDD for $f = x_0 + x_1$ is built. If a third variable is added, say x_2, then the ZDD represents $g = (x_0 + x_1)x_2'$ instead. This change in function obviously applies regardless of what use is made of the ZDD. However, if the ZDD is used to represent a cover, the cover itself is not changed by the addition of new variable. (What changes is the characteristic function of the cover.)

3.12 Variable Reordering for BDDs and ADDs

The CUDD package provides a rich set of dynamic reordering algorithms. Some of them are slight variations of existing techniques [16, 6, 2, 10, 15, 11]; some others have been developed specifically for this package [14, 13].

Reordering affects a unique table. This means that BDDs and ADDs, which share the same unique table are simultaneously reordered. ZDDs, on the other hand, are reordered separately. In the following we discuss the reordering of BDDs and ADDs. Reordering for ZDDs is the subject of Section 3.14.

Reordering of the variables can be invoked directly by the application by calling *Cudd_ReduceHeap*. Or it can be automatically triggered by the package when the number of nodes has reached a given threshold. (The threshold is initialized and automatically adjusted after each reordering by the package.) To enable automatic dynamic reordering (also called *asynchronous* dynamic reordering in this document) the application must call *Cudd_AutodynEnable*. Automatic dynamic reordering can subsequently be disabled by calling *Cudd_AutodynDisable*.

All reordering methods are available in both the case of direct call to *Cudd_ReduceHeap* and the case of automatic invocation. For many methods, the reordering procedure is iterated until no further improvement is obtained. We call these methods the *converging* methods. When constraints are imposed on the relative position of variables (see Section 3.13) the reordering methods apply inside the groups. The groups themselves are reordered by sifting. Each method is identified by a constant of the enumerated type *Cudd_ReorderingType* defined in *cudd.h* (the external header file of the CUDD package):

CUDD_REORDER_NONE: This method causes no reordering.

CUDD_REORDER_SAME: If passed to *Cudd_AutodynEnable*, this method leaves the current method for automatic reordering unchanged. If passed to *Cudd_ReduceHeap*, this method causes the current method for automatic reordering to be used.

CUDD_REORDER_RANDOM: Pairs of variables are randomly chosen, and swapped in the order. The swap is performed by a series of swaps of adjacent variables. The best order among those obtained by the series of swaps is retained. The number of pairs chosen for swapping equals the number of variables in the diagram.

CUDD_REORDER_RANDOM_PIVOT: Same as CUDD_REORDER_RANDOM, but the two variables are chosen so that the first is above the variable with the largest number of nodes, and the second is below that variable. In case there are several variables tied for the maximum number of nodes, the one closest to the root is used.

CUDD_REORDER_SIFT: This method is an implementation of Rudell's sifting algorithm [16]. A simplified description of sifting is as follows: Each variable is considered in turn. A variable is moved up and down in the order so that it takes all possible positions. The best position is identified and the variable is returned to that position.

In reality, things are a bit more complicated. For instance, there is a limit on the number of variables that will be sifted. This limit can be read with

Cudd_ReadSiftMaxVar and set with *Cudd_SetSiftMaxVar*. In addition, if the diagram grows too much while moving a variable up or down, that movement is terminated before the variable has reached one end of the order. The maximum ratio by which the diagram is allowed to grow while a variable is being sifted can be read with *Cudd_ReadMaxGrowth* and set with *Cudd_SetMaxGrowth*.

CUDD_REORDER_SIFT_CONVERGE: This is the converging variant of CUDD_REORDER_SIFT.

CUDD_REORDER_SYMM_SIFT: This method is an implementation of symmetric sifting [14]. It is similar to sifting, with one addition: Variables that become adjacent during sifting are tested for symmetry. If they are symmetric, they are linked in a group. Sifting then continues with a group being moved, instead of a single variable. After symmetric sifting has been run, *Cudd_SymmProfile* can be called to report on the symmetry groups found. (Both positive and negative symmetries are reported.)

CUDD_REORDER_SYMM_SIFT_CONV: This is the converging variant of CUDD_REORDER_SYMM_SIFT.

CUDD_REORDER_GROUP_SIFT: This method is an implementation of group sifting [13]. It is similar to symmetric sifting, but aggregation is not restricted to symmetric variables.

CUDD_REORDER_GROUP_SIFT_CONV: This method repeats until convergence the combination of CUDD_REORDER_GROUP_SIFT and CUDD_REORDER_WINDOW4.

CUDD_REORDER_WINDOW2: This method implements the window permutation approach of Fujita [8] and Ishiura [10]. The size of the window is 2.

CUDD_REORDER_WINDOW3: Similar to CUDD_REORDER_WINDOW2, but with a window of size 3.

CUDD_REORDER_WINDOW4: Similar to CUDD_REORDER_WINDOW2, but with a window of size 4.

CUDD_REORDER_WINDOW2_CONV: This is the converging variant of CUDD_REORDER_WINDOW2.

CUDD_REORDER_WINDOW3_CONV: This is the converging variant of CUDD_REORDER_WINDOW3.

CUDD_REORDER_WINDOW4_CONV: This is the converging variant of CUDD_REORDER_WINDOW4.

CUDD_REORDER_ANNEALING: This method is an implementation of simulated annealing for variable ordering, vaguely resemblant of the algorithm of [2]. This method is potentially very slow.

CUDD_REORDER_GENETIC: This method is an implementation of a genetic algorithm for variable ordering, inspired by the work of Drechsler [6]. This method is potentially very slow.

CUDD_REORDER_EXACT: This method implements a dynamic programming approach to exact reordering [9, 7, 10], with improvements described in [11]. It only stores one BDD at the time. Therefore, it is relatively efficient in terms of memory. Compared to other reordering strategies, it is very slow, and is not recommended for more than 16 variables.

So far we have described methods whereby the package selects an order automatically. A given order of the variables can also be imposed by calling *Cudd_ShuffleHeap*.

3.13 Grouping Variables

CUDD allows the application to specify constraints on the positions of group of variables. It is possible to request that a group of contiguous variables be kept contiguous by the reordering procedures. It is also possible to request that the relative order of some groups of variables be left unchanged. The constraints on the order are specified by means of a tree, which is created in one of two ways:

- By calling *Cudd_MakeTreeNode*.
- By calling the functions of the MTR library (part of the distribution), and by registering the result with the manager using *Cudd_SetTree*. The current tree registered with the manager can be read with *Cudd_ReadTree*.

Each node in the tree represents a range of variables. The lower bound of the range is given by the *low* field of the node, and the size of the group is given by the *size* field of the node.[2] The variables in each range are kept contiguous. Furthermore, if a node is marked with the MTR_FIXED flag, then the relative order of the variable ranges associated to its children is not changed. As an example, suppose the initial variable order is:

[2] When the variables in a group are reordered, the association between the *low* field and the index of the first variable in the group is lost. The package updates the tree to keep track of the changes. However, the application cannot rely on *low* to determine the position of variables.

```
x0, y0, z0, x1, y1, z1, ... , x9, y9, z9.
```

Suppose we want to keep each group of three variables with the same index (e.g., `x3, y3, z3`) contiguous, while allowing the package to change the order of the groups. We can accomplish this with the following code:

```
for (i = 0; i < 10; i++) {
     (void) Cudd_MakeTreeNode(manager,i*3,3,MTR_DEFAULT);
}
```

If we want to keep the order within each group of variables fixed (i.e., `x` before `y` before `z`) we need to change MTR_DEFAULT into MTR_FIXED.

The *low* parameter passed to *Cudd_MakeTreeNode* is the index of a variable (as opposed to its level or position in the order). The group tree can be created at any time. The result obviously depends on the variable order in effect at creation time.

It is possible to create a variable group tree also before the variables themselves are created. The package assumes in this case that the index of the variables not yet in existence will equal their position in the order when they are created. Therefore, applications that rely on *Cudd_bddNewVarAtLevel* or *Cudd_addNewVarAtLevel* to create new variables have to create the variables before they group them.

The reordering procedure will skip all groups whose variables are not yet in existence. For groups that are only partially in existence, the reordering procedure will try to reorder the variables already instantiated, without violating the adjacency constraints.

3.14 Variable Reordering for ZDDs

Reordering of ZDDs is done in much the same way as the reordering of BDDs and ADDs. The functions corresponding to *Cudd_ReduceHeap* and *Cudd_ShuffleHeap* are *Cudd_zddReduceHeap* and *Cudd_zddShuffleHeap*. To enable dynamic reordering, the application must call *Cudd_AutodynEnableZdd*, and to disable dynamic reordering, it must call *Cudd_AutodynDisableZdd*. In the current implementation, however, the choice of reordering methods for ZDDs is more limited. Specifically, these methods are available:

CUDD_REORDER_NONE;

CUDD_REORDER_SAME;

CUDD_REORDER_RANDOM;

CUDD_REORDER_RANDOM_PIVOT;

CUDD_REORDER_SIFT;

CUDD_REORDER_SIFT_CONVERGE;

CUDD_REORDER_SYMM_SIFT;

CUDD_REORDER_SYMM_SIFT_CONV.

To create ZDD variable groups, the application calls *Cudd_MakeZddTreeNode*.

3.15 Keeping Consistent Variable Orders for BDDs and ZDDs

Several applications that manipulate both BDDs and ZDDs benefit from keeping a fixed correspondence between the order of the BDD variables and the order of the ZDD variables. If each BDD variable corresponds to a group of ZDD variables, then it is often desirable that the groups of ZDD variables be in the same order as the corresponding BDD variables. CUDD allows the ZDD order to track the BDD order and vice versa. To have the ZDD order track the BDD order, the application calls *Cudd_zddRealignEnable*. The effect of this call can be reversed by calling *Cudd_zddRealignDisable*. When ZDD realignment is in effect, automatic reordering of ZDDs should be disabled.

3.16 Hooks

Hooks in CUDD are lists of application-specified functions to be run on certain occasions. Each hook is identified by a constant of the enumerated type *Cudd_HookType*. In Version 2.5.0 hooks are defined for these occasions:

- before garbage collection (CUDD_PRE_GC_HOOK);
- after garbage collection (CUDD_POST_GC_HOOK);
- before variable reordering (CUDD_PRE_REORDERING_HOOK);
- after variable reordering (CUDD_POST_REORDERING_HOOK).

The current implementation of hooks is experimental. A function added to a hook receives a pointer to the manager, a pointer to a constant string, and a pointer to void as arguments; it must return 1 if successful; 0 otherwise. The second argument is one of "DD," "BDD," and "ZDD." This allows the hook functions to tell the type of diagram for which reordering or garbage collection takes place. The third argument varies depending on the hook. The hook functions called before or after garbage collection do not use it. The hook functions called before reordering are passed, in addition to the pointer to the manager, also the method used for reordering. The hook functions called after reordering are passed the start time. To add a function to a hook, one uses *Cudd_AddHook*. The function of a given hook are called in the order in which they were added to the hook. For sample hook functions, one may look at *Cudd_StdPreReordHook* and *Cudd_StdPostReordHook*.

3.17 Timeouts and Limits

It is possible to set a time limit for a manger with *Cudd_SetTimeLimit*. Once set, the time available to the manager can be modified through other API functions. CUDD checks for expiration periodically. When time has expired, it returns NULL from the call in progress, but it leaves the manager in a consistent state. The invoking application must be designed to handle the NULL values returned.

When reordering, if a timout is approaching, CUDD will quit reordering to give the application a chance to finish some computation.

It is also possible to invoke some functions that return NULL if they cannot complete without creating more than a set number of nodes. See, for instance, *Cudd_bddAndLimit*.

3.18 The SIS/VIS Interface

The CUDD package contains interface functions that emulate the behavior of the original BDD package used in SIS [17] and in the newer VIS[3] [4]. How to build VIS with CUDD is described in the installation documents of VIS. (Version 1.1 and later.)

3.18.1 Using the CUDD Package in SIS

This section describes how to build SIS with the CUDD package. Let `SISDIR` designate the root of the directory hierarchy where the sources for SIS reside. Let `CUDDDIR` be the root of the directory hierarchy where the distribution of the CUDD package resides. To build SIS with the CUDD package, follow these steps.

1. Create directories `SISDIR/sis/cudd` and `SISDIR/sis/mtr`.
2. Copy all files from `CUDDDIR/cudd` and `CUDDDIR/sis` to `SISDIR/sis/cudd` and all files from `CUDDDIR/mtr` to `SISDIR/sis/mtr`.
3. Copy `CUDDDIR/cudd/doc/cudd.doc` to `SISDIR/sis/cudd`; also copy `CUDDDIR/mtr/doc/mtr.doc` to `SISDIR/sis/mtr`.

[3] http://vlsi.Colorado.EDU/ vis/

4. In `SISDIR/sis/cudd` make `bdd.h` a symbolic link to `cuddBdd.h`. (That is: `ln -s cuddBdd.h bdd.h`.)

5. In `SISDIR/sis/cudd` delete `Makefile` and rename `Makefile.sis` as `Makefile`. Do the same in `SISDIR/sis/mtr`.

6. Copy `CUDDDIR/sis/st.[ch]` and `CUDDDIR/st/doc/st.doc` to `SISDIR/sis/st`. (This will overwrite the original files: You may want to save them beforehand.)

7. From `CUDDDIR/util` copy `datalimit.c` to `SISDIR/sis/util`. Update `util.h` and `Makefile` in `SISDIR/sis/util`. Specifically, add the declaration `EXTERN long getSoftDataLimit();` to `util.h` and add `datalimit.c` to the list of source files (PSRC) in `Makefile`.

8. In `SISDIR/sis` remove the link from `bdd` to `bdd_cmu` or `bdd_ucb` (that is, `rm bdd`) and make `bdd` a symbolic link to `cudd`. (That is: `ln -s cudd bdd`.)

9. Still in `SISDIR/sis`, edit `Makefile`, `Makefile.oct`, and `Makefile.nooct`. In all three files add mtr to the list of directories to be made (DIRS).

10. In `SISDIR/sis/include` make `mtr.h` a symbolic link to `../mtr/mtr.h`.

11. In `SISDIR/sis/doc` make `cudd.doc` a symbolic link to `../cudd/cudd.doc` and `mtr.doc` a symbolic link to `../mtr/mtr.doc`. (That is: `ln -s ../cudd/cudd.doc .; ln -s ../mtr/mtr.doc .`.)

12. From `SISDIR` do `make clean` followed by `make -i`. This should create a working copy of SIS that uses the CUDD package.

The replacement for the `st` library is because the version shipped with the CUDD package tests for out-of-memory conditions. Notice that the version of the `st` library to be used for replacement is not the one used for the normal build, because the latter has been modified for C++ compatibility. The above installation procedure has been tested on SIS 1.3. SIS can be obtained via anonymous FTP from `ic.eecs.berkeley.edu`. To build SIS 1.3, you need `sis-1.2.tar.Z` and `sis-1.2.patch1.Z`. When compiling on a DEC Alpha, you should add the `-ieee_with_no_inexact` flag. (See Section 3.5.2.) Refer to the `Makefile` in the top level directory of the distribution for how to compile with 32-bit pointers.

3.19 Writing Decision Diagrams to a File

The CUDD package provides several functions to write decision diagrams to a file. *Cudd_DumpBlif* writes a file in *blif* format. It is restricted to BDDs. The diagrams are written as a network of multiplexers, one multiplexer for each internal node of the BDD.

Cudd_DumpDot produces input suitable to the graph-drawing program *dot*[4] written by Eleftherios Koutsofios and Stephen C. North. An example of drawing produced by dot from the output of *Cudd_DumpDot* is shown in Figure 1. It is restricted to BDDs and ADDs. *Cudd_zddDumpDot* is the analog of *Cudd_DumpDot* for ZDDs.

Cudd_DumpDaVinci produces input suitable to the graph-drawing program *daVinci*[5] developed at the University of Bremen. It is restricted to BDDs and ADDs.

Functions are also available to produce the input format of *DDcal* (see Section 2.2) and factored forms.

3.20 Saving and Restoring BDDs

The *dddmp*[6] library by Gianpiero Cabodi and Stefano Quer allows a CUDD application to save BDDs to disk in compact form for later retrieval. See the library's own documentation for the details.

4 Programmer's Manual

This section provides additional detail on the working of the CUDD package and on the programming conventions followed in its writing. The additional detail should help those who want to write procedures that directly manipulate the CUDD data structures.

4.1 Compiling and Linking

If you plan to use the CUDD package as a clear box (for instance, you want to write a procedure that traverses a decision diagram) you need to add

```
#include "cuddInt.h"
```

[4]http://www.research.att.com/sw/tools/graphviz

[5]ftp://ftp.uni-bremen.de/pub/graphics/daVinci

[6]ftp://ftp.polito.it/pub/research/dddmp/

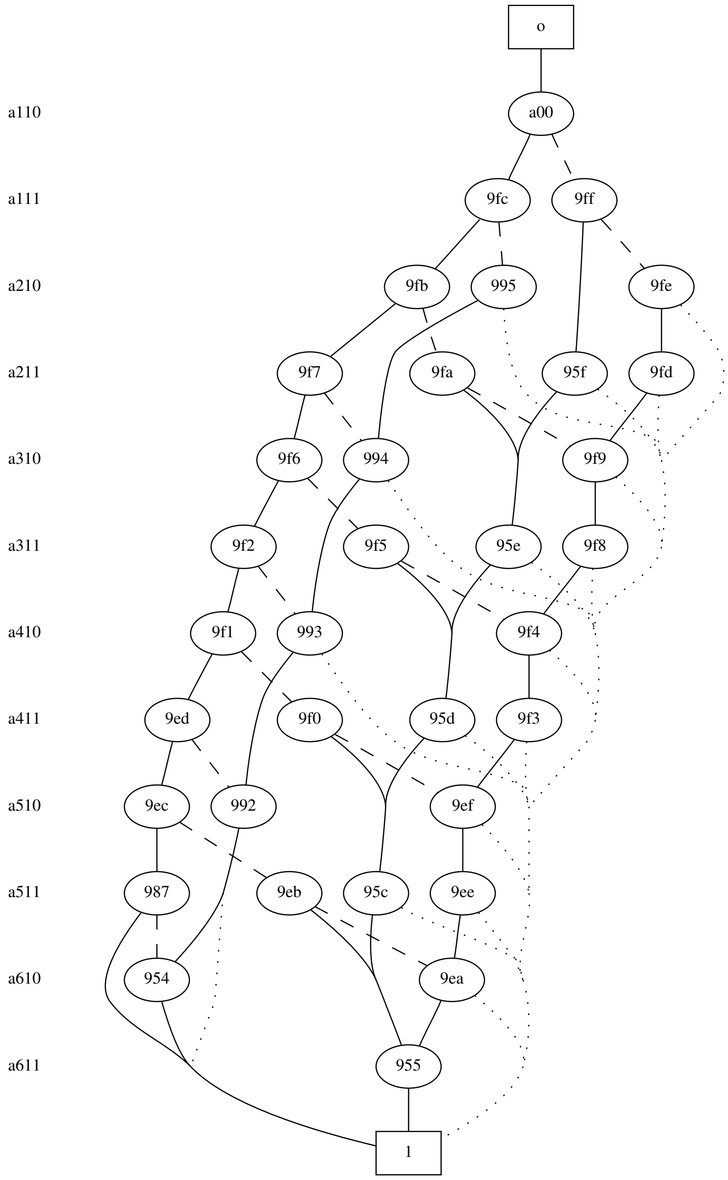

Figure 1: A BDD representing a phase constraint for the optimization of fixed-polarity Reed-Muller forms. The label of each node is the unique part of the node address. All nodes on the same level correspond to the same variable, whose name is shown at the left of the diagram. Dotted lines indicate complement arcs. Dashed lines indicate regular "else" arcs.

to your source files. In addition, you should link `libcudd.a` to your executable. Some platforms require specific compiler and linker flags. Refer to the `Makefile` in the top level directory of the distribution.

4.2 Reference Counts

Garbage collection in the CUDD package is based on reference counts. Each node stores the sum of the external references and internal references. An internal BDD or ADD node is created by a call to *cuddUniqueInter*, an internal ZDD node is created by a call to *cuddUniqueInterZdd*, and a terminal node is created by a call to *cuddUniqueConst*. If the node returned by these functions is new, its reference count is zero. The function that calls *cuddUniqueInter*, *cuddUniqueInterZdd*, or *cuddUniqueConst* is responsible for increasing the reference count of the node. This is accomplished by calling *Cudd_Ref*.

When a function is no longer needed by an application, the memory used by its diagram can be recycled by calling *Cudd_RecursiveDeref* (BDDs and ADDs) or *Cudd_RecursiveDerefZdd* (ZDDs). These functions decrease the reference count of the node passed to them. If the reference count becomes 0, then two things happen:

1. The node is declared "dead;" this entails increasing the counters of the dead nodes. (One counter for the subtable to which the node belongs, and one global counter for the unique table to which the node belongs.) The node itself is not affected.

2. The function is recursively called on the two children of the node.

For instance, if the diagram of a function does not share any nodes with other diagrams, then calling *Cudd_RecursiveDeref* or *Cudd_RecursiveDerefZdd* on its root will cause all the nodes of the diagram to become dead.

When the number of dead nodes reaches a given level (dynamically determined by the package) garbage collection takes place. During garbage collection dead nodes are returned to the node free list.

When a new node is created, it is important to increase its reference count before one of the two following events occurs:

1. A call to *cuddUniqueInter*, to *cuddUniqueInterZdd*, to *cuddUniqueConst*, or to a function that may eventually cause a call to them.

2. A call to *Cudd_RecursiveDeref*, to *Cudd_RecursiveDerefZdd*, or to a function that may eventually cause a call to them.

In practice, it is recommended to increase the reference count as soon as the returned pointer has been tested for not being NULL.

4.2.1 NULL Return Values

The interface to the memory management functions (e.g., malloc) used by CUDD intercepts NULL return values and calls a handler. The default handler exits with an error message. If the application does not install another handler, therefore, a NULL return value from an exported function of CUDD signals an internal error.

If the aplication, however, installs another handler that lets execution continue, a NULL pointer returned by an exported function typically indicates that the process has run out of memory. *Cudd_ReadErrorCode* can be used to ascertain the nature of the problem.

An application that tests for the result being NULL can try some remedial action, if it runs out of memory. For instance, it may free some memory that is not strictly necessary, or try a slower algorithm that takes less space. As an example, CUDD overrides the default handler when trying to enlarge the cache or increase the number of slots of the unique table. If the allocation fails, the package prints out a message and continues without resizing the cache.

4.2.2 *Cudd_RecursiveDeref* vs. *Cudd_Deref*

It is often the case that a recursive procedure has to protect the result it is going to return, while it disposes of intermediate results. (See the previous discussion on when to increase reference counts.) Once the intermediate results have been properly disposed of, the final result must be returned to its pristine state, in which the root node may have a reference count of 0. One cannot use *Cudd_RecursiveDeref* (or *Cudd_RecursiveDerefZdd*) for this purpose, because it may erroneously make some nodes dead. Therefore, the package provides a different function: *Cudd_Deref*. This function is not recursive, and does not change the dead node counts. Its use is almost exclusively the one just described: Decreasing the reference count of the root of the final result before returning from a recursive procedure.

4.2.3 When Increasing the Reference Count is Unnecessary

When a copy of a predefined constant or of a simple BDD variable is needed for comparison purposes, then calling *Cudd_Ref* is not necessary, because these simple functions are guaranteed to have reference counts greater than 0 at all times. If no call to *Cudd_Ref* is made, then no attempt to free the diagram by calling *Cudd_RecursiveDeref* or *Cudd_RecursiveDerefZdd* should be made.

4.2.4 Saturating Increments and Decrements

On 32-bit machines, the CUDD package stores the reference counts in unsigned short int's. For large diagrams, it is possible for some reference counts to exceed the capacity of an unsigned short int. Therefore, increments and decrements of reference counts are *saturating*. This means that once a reference count has reached the maximum possible value, it is no longer changed by calls to *Cudd_Ref*, *Cudd_RecursiveDeref*, *Cudd_RecursiveDerefZdd*, or *Cudd_Deref*. As a consequence, some nodes that have no references may not be declared dead. This may result in a small waste of memory, which is normally more than offset by the reduction in size of the node structure.

When using 64-bit pointers, there is normally no memory advantage from using short int's instead of int's in a DdNode. Therefore, increments and decrements are not saturating in that case. What option is in effect depends on two macros, SIZEOF_VOID_P and SIZEOF_INT, defined in the external header file (*cudd.h*). The increments and decrements of the reference counts are performed using two macros: *cuddSatInc* and *cuddSatDec*, whose definitions depend on SIZEOF_VOID_P and SIZEOF_INT.

4.3 Complement Arcs

If ADDs are restricted to use only the constants 0 and 1, they behave like BDDs without complement arcs. It is normally easier to write code that manipulates 0-1 ADDs, than to write code for BDDs. However, complementation is trivial with complement arcs, and is not trivial without. As a consequence, with complement arcs it is possible to check for more terminal cases and it is possible to apply De Morgan's laws to reduce problems that are essentially identical to a standard form. This in turn increases the utilization of the cache.

The complement attribute is stored in the least significant bit of the "else" pointer of each node. An external pointer to a function can also be complemented. The "then" pointer to a node, on the other hand, is always *regular*. It is a mistake to use a complement pointer as it is to address memory. Instead, it is always necessary to obtain a regular version of it. This is normally done by calling *Cudd_Regular*. It is also a mistake to call *cuddUniqueInter* with a complemented "then" child as argument. The calling procedure must apply De Morgan's laws by complementing both pointers passed to *cuddUniqueInter* and then taking the complement of the result.

4.4 The Cache

Each entry of the cache consists of five fields: The operator, three pointers to operands and a pointer to the result. The operator and the three pointers to the operands are combined to form three words. The combination relies on two facts:

- Most operations have one or two operands. A few bits are sufficient to discriminate all three-operands operations.
- All nodes are aligned to 16-byte boundaries. (32-byte boundaries if 64-bit pointers are used.) Hence, there are a few bits available to distinguish the three-operand operations from te others and to assign unique codes to them.

The cache does not contribute to the reference counts of the nodes. The fact that the cache contains a pointer to a node does not imply that the node is alive. Instead, when garbage collection takes place, all entries of the cache pointing to dead nodes are cleared.

The cache is also cleared (of all entries) when dynamic reordering takes place. In both cases, the entries removed from the cache are about to become invalid.

All operands and results in a cache entry must be pointers to DdNodes. If a function produces more than one result, or uses more than three arguments, there are currently two solutions:

- Build a separate, local, cache. (Using, for instance, the *st* library.)
- Combine multiple results, or multiple operands, into a single diagram, by building a "multiplexing structure" with reserved variables.

Support of the former solution is under development. (See `cuddLCache.c`..) Support for the latter solution may be provided in future versions of the package.

There are three sets of interface functions to the cache. The first set is for functions with three operands: *cuddCacheInsert* and *cuddCacheLookup*. The second set is for functions with two operands: *cuddCacheInsert2* and *cuddCacheLookup2*. The second set is for functions with one operand: *cuddCacheInsert1* and *cuddCacheLookup1*. The second set is slightly faster than the first, and the third set is slightly faster than the second.

4.4.1 Cache Sizing

The size of the cache can increase during the execution of an application. (There is currently no way to decrease the size of the cache, though it would not be difficult to do it.) When a cache miss occurs, the package uses the following criteria to decide whether to resize the cache:

1. If the cache already exceeds the limit given by the `maxCache` field of the manager, no resizing takes place. The limit is the minimum of two values: a value set at initialization time and possibly modified by the application, which constitutes the hard limit beyond which the cache will never grow; and a number that depends on the current total number of slots in the unique table.

2. If the cache is not too large already, resizing is decided based on the hit rate. The policy adopted by the CUDD package is "reward-based." If the cache hit rate is high, then it is worthwhile to increase the size of the cache.

When resizing takes place, the statistical counters used to compute the hit rate are reinitialized so as to prevent immediate resizing. The number of entries is doubled.

The rationale for the "reward-based" policy is as follows. In many BDD/ADD applications the hit rate is not very sensitive to the size of the cache: It is primarily a function of the problem instance at hand. If a large hit rate is observed, chances are that by using a large cache, the results of large problems (those that would take longer to solve) will survive in the cache without being overwritten long enough to cause a valuable cache hit. Notice that when a large problem is solved more than once, so are its recursively generated subproblems. If the hit rate is low, the probability of large problems being solved more than once is low.

The other observation about the cache sizing policy is that there is little point in keeping a cache which is much larger than the unique table. Every time the unique table "fills up," garbage collection is invoked and the cache is cleared of all dead entries. A cache that is much larger than the unique table is therefore less than fully utilized.

4.4.2 Local Caches

Sometimes it may be necessary or convenient to use a local cache. A local cache can be lossless (no results are ever overwritten), or it may store objects for which canonical representations are not available. One important fact to keep in mind when using a local cache is that local caches are not cleared during garbage collection or before reordering. Therefore, it is necessary to increment the reference count of all nodes pointed by a local cache. (Unless their reference counts are guaranteed positive in some other way. One such way is by including all partial results in the global result.) Before disposing of the local cache, all elements stored in it must be passed to *Cudd_RecursiveDeref*. As consequence of the fact that all results in a local cache are referenced, it is generally convenient to store in the local cache also the result of trivial problems, which are not usually stored in the global cache.

Otherwise, after a recursive call, it is difficult to tell whether the result is in the cache, and therefore referenced, or not in the cache, and therefore not referenced.

An alternative approach to referencing the results in the local caches is to install hook functions (see Section 3.16) to be executed before garbage collection.

4.5 The Unique Table

A recursive procedure typically splits the operands by expanding with respect to the topmost variable. Topmost in this context refers to the variable that is closest to the roots in the current variable order. The nodes, on the other hand, hold the index, which is invariant with reordering. Therefore, when splitting, one must use the permutation array maintained by the package to get the right level. Access to the permutation array is provided by the macro *cuddI* for BDDs and ADDs, and by the macro *cuddIZ* for ZDDs.

The unique table consists of as many hash tables as there are variables in use. These has tables are called *unique subtables*. The sizes of the unique subtables are determined by two criteria:

1. The collision lists should be short to keep access time down.
2. There should be enough room for dead nodes, to prevent too frequent garbage collections.

While the first criterion is fairly straightforward to implement, the second leaves more room to creativity. The CUDD package tries to figure out whether more dead node should be allowed to increase performance. (See also Section 3.4.) There are two reasons for not doing garbage collection too often. The obvious one is that it is expensive. The second is that dead nodes may be reclaimed, if they are the result of a successful cache lookup. Hence dead nodes may provide a substantial speed-up if they are kept around long enough. The usefulness of keeping many dead nodes around varies from application to application, and from problem instance to problem instance. As in the sizing of the cache, the CUDD package adopts a "reward-based" policy to decide how much room should be used for the unique table. If the number of dead nodes reclaimed is large compared to the number of nodes directly requested from the memory manager, then the CUDD package assumes that it will be beneficial to allow more room for the subtables, thereby reducing the frequency of garbage collection. The package does so by switching between two modes of operation:

1. Fast growth: In this mode, the ratio of dead nodes to total nodes required for garbage collection is higher than in the slow growth mode to favor resizing of the subtables.

2. Slow growth: In this mode keeping many dead nodes around is not as important as keeping memory requirements low.

Switching from one mode to the other is based on the following criteria:

1. If the unique table is already large, only slow growth is possible.
2. If the table is small and many dead nodes are being reclaimed, then fast growth is selected.

This policy is especially effective when the diagrams being manipulated have lots of recombination. Notice the interplay of the cache sizing and unique sizing: Fast growth normally occurs when the cache hit rate is large. The cache and the unique table then grow in concert, preserving a healthy balance between their sizes.

4.6 Allowing Asynchronous Reordering

Asynchronous reordering is the reordering that is triggered automatically by the increase of the number of nodes. Asynchronous reordering takes place when a new internal node must be created, and the number of nodes has reached a given threshold. (The threshold is adjusted by the package every time reordering takes place.)

Those procedures that do not create new nodes (e.g., procedures that count the number of nodes or minterms) need not worry about asynchronous reordering: No special precaution is necessary in writing them.

Procedures that only manipulate decision diagrams through the exported functions of the CUDD package also need not concern themselves with asynchronous reordering. (See Section 3.2.1 for the exceptions.)

The remaining class of procedures is composed of functions that visit the diagrams and may create new nodes. All such procedures in the CUDD package are written so that they can be interrupted by dynamic reordering. The general approach followed goes under the name of "abort and retry." As the name implies, a computation that is interrupted by dynamic reordering is aborted and tried again.

A recursive procedure that can be interrupted by dynamic reordering (an interruptible procedure from now on) is composed of two functions. One is responsible for the real computation. The other is a simple wrapper, which tests whether reordering occurred and restarts the computation if it did.

Asynchronous reordering of BDDs and ADDs can only be triggered inside *cuddUniqueInter*, when a new node is about to be created. Likewise, asynchronous reordering of ZDDs can only be triggered inside *cuddUniqueInterZdd*. When reordering is triggered, three things happen:

1. *cuddUniqueInter* returns a NULL value;
2. The flag *reordered* of the manager is set to 1. (0 means no reordering, while 2 indicates an error occurred during reordering.)
3. The counter *reorderings* of the manager is incremented. The counter is initialized to 0 when the manager is started and can be accessed by calling *Cudd_ReadReorderings*. By taking two readings of the counter, an application can determine if variable reordering has taken place between the first and the second reading. The package itself, however, does not make use of the counter: It is mentioned here for completeness.

The recursive procedure that receives a NULL value from *cuddUniqueInter* must free all intermediate results that it may have computed before, and return NULL in its turn.

The wrapper function does not decide whether reordering occurred based on the NULL return value, because the NULL value may be the result of lack of memory. Instead, it checks the *reordered* flag.

When a recursive procedure calls another recursive procedure that may cause reordering, it should bypass the wrapper and call the recursive procedure directly. Otherwise, the calling procedure will not know whether reordering occurred, and will not be able to restart. This is the main reason why most recursive procedures are internal, rather than static. (The wrappers, on the other hand, are mostly exported.)

4.7 Debugging

By defining the symbol DD_DEBUG during compilation, numerous checks are added to the code. In addition, the procedures *Cudd_DebugCheck*, *Cudd_CheckKeys*, and *cuddHeapProfile* can be called at any point to verify the consistency of the data structure. (*cuddHeapProfile* is an internal procedure. It is declared in *cuddInt.h*.) Procedures *Cudd_DebugCheck* and *Cudd_CheckKeys* are especially useful when CUDD reports that during garbage collection the number of nodes actually deleted from the unique table is different from the count of dead nodes kept by the manager. The error causing the discrepancy may have occurred much earlier than it is discovered. A few strategicaly placed calls to the debugging procedures can considerably narrow down the search for the source of the problem. (For instance, a call to *Cudd_RecursiveDeref* where one to *Cudd_Deref* was required may be identified in this way.)

One of the most common problems encountered in debugging code based on the CUDD package is a missing call to *Cudd_RecursiveDeref*. To help identify this type of problems, the package provides a function called *Cudd_CheckZeroRef*. This function should be called immediately before shutting down the manager. *Cudd_CheckZeroRef* checks that the only nodes left with non-zero reference counts are the predefined constants, the BDD projection functions, and nodes whose reference counts are saturated.

For this function to be effective the application must explicitly dispose of all diagrams to which it has pointers before calling it.

4.8 Gathering and Interpreting Statistics

Function *Cudd_PrintInfo* can be called to print out the values of parameters and statistics for a manager. The output of *Cudd_PrintInfo* is divided in two sections. The first reports the values of parameters that are under the application control. The second reports the values of statistical counters and other non-modifiable parameters. A quick guide to the interpretation of all these quantities follows. For ease of exposition, we reverse the order and describe the non-modifiable parameters first. We'll use a sample run as example. There is nothing special about this run.

4.8.1 Non Modifiable Parameters

The list of non-modifiable parameters starts with:

```
**** CUDD non-modifiable parameters ****
Memory in use: 32544220
```

This is the memory used by CUDD for three things mainly: Unique table (including all DD nodes in use), node free list, and computed table. This number almost never decreases in the lifetime of a CUDD manager, because CUDD does not release memory when it frees nodes. Rather, it puts the nodes on its own free list. This number is in bytes. It does not represent the peak memory occupation, because it does not include the size of data structures created temporarily by some functions (e.g., local look-up tables).

```
Peak number of nodes: 837018
```

This number is the number of nodes that the manager has allocated. This is not the largest size of the BDDs, because the manager will normally have some dead nodes and some nodes on the free list.

```
Peak number of live nodes: 836894
```

This is the largest number of live nodes that the manager has held since its creation.

```
Number of BDD variables: 198
Number of ZDD variables: 0
```

These numbers tell us this run was not using ZDDs.

```
Number of cache entries: 1048576
```

Current number of slots of the computed table. If one has a performance problem, this is one of the numbers to look at. The cache size is always a power of 2.

```
Number of cache look-ups: 2996536
Number of cache hits: 1187087
```

These numbers give an indication of the hit rate in the computed table. It is not unlikely for model checking runs to get hit rates even higher than this one (39.62%).

```
Number of cache insertions: 1809473
Number of cache collisions: 961208
Number of cache deletions: 0
```

A collision occurs when a cache entry is overwritten. A deletion occurs when a cache entry is invalidated (e.g., during garbage collection). If the number of deletions is high compared to the number of collisions, it means that garbage collection occurs too often. In this case there were no garbage collections; hence, no deletions.

```
Cache used slots = 80.90% (expected 82.19%)
```

Percentage of cache slots that contain a valid entry. If this number is small, it may signal one of three conditions:

1. The cache may have been recently resized and it is still filling up.
2. The cache is too large for the BDDs. This should not happen if the size of the cache is determined by CUDD.
3. The hash function is not working properly. This is accompanied by a degradation in performance. Conversely, a degradation in performance may be due to bad hash function behavior.

The expected value is computed assuming a uniformly random distribution of the accesses. If the difference between the measured value and the expected value is large (unlike this case), the cache is not working properly.

```
Soft limit for cache size: 1318912
```

This number says how large the cache can grow. This limit is based on the size of the unique table. CUDD uses a reward-based policy for growing the cache. (See Section 4.4.1.) The default hit rate for resizing is 30% and the value in effect is reported among the modifiable parameters.

```
Number of buckets in unique table: 329728
```

This number is exactly one quarter of the one above. This is indeed how the soft limit is determined currently, unless the computed table hits the specified hard limit. (See below.)

```
Used buckets in unique table: 87.96% (expected 87.93%)
```

Percentage of unique table buckets that contain at least one node. Remarks analogous to those made about the used cache slots apply.

```
Number of BDD and ADD nodes: 836894
Number of ZDD nodes: 0
```

How many nodes are currently in the unique table, either alive or dead.

```
Number of dead BDD and ADD nodes: 0
Number of dead ZDD nodes: 0
```

Subtract these numbers from those above to get the number of live nodes. In this case there are no dead nodes because the application uses delayed dereferencing *Cudd_DelayedDerefBdd*.

```
Total number of nodes allocated: 836894
```

This is the total number of nodes that were requested and obtained from the free list. It never decreases, and is not an indication of memory occupation after the first garbage collection. Rather, it is a measure of the package activity.

```
Total number of nodes reclaimed: 0
```

These are the nodes that were resuscitated from the dead. If they are many more than the allocated nodes, and the total number of slots is low relative to the number of nodes, then one may want to increase the limit for fast unique table growth. In this case, the number is 0 because of delayed dereferencing.

```
Garbage collections so far: 0
Time for garbage collections: 0.00 sec
Reorderings so far: 0
Time for reordering: 0.00 sec
```

There is a GC for each reordering. Hence the first count will always be at least as large as the second.

```
Node swaps in reordering: 0
```

This is the number of elementary reordering steps. Each step consists of the re-expression of one node while swapping two adjacent variables. This number is a good measure of the amount of work done in reordering.

4.8.2 Modifiable Parameters

Let us now consider the modifiable parameters, that is, those settings on which the application or the user has control.

```
**** CUDD modifiable parameters ****
Hard limit for cache size: 8388608
```

This number counts entries. Each entry is 16 bytes if CUDD is compiled to use 32-bit pointers. Two important observations are in order:

1. If the datasize limit is set, CUDD will use it to determine this number automatically. On a Unix system, one can type "limit" or "ulimit" to verify if this value is set. If the datasize limit is not set, CUDD uses a default which is rather small. If you have enough memory (say 64MB or more) you should seriously consider *not* using the default. So, either set the datasize limit, or override the default with *Cudd_SetMaxCacheHard*.

2. If a process seems to be going nowhere, a small value for this parameter may be the culprit. One cannot overemphasize the importance of the computed table in BDD algorithms.

In this case the limit was automatically set for a target maximum memory occupation of 104 MB.

```
Cache hit threshold for resizing: 15%
```

This number can be changed if one suspects performance is hindered by the small size of the cache, and the cache is not growing towards the soft limit sufficiently fast. In such a case one can change the default 30% to 15% (as in this case) or even 1%.

```
Garbage collection enabled: yes
```

One can disable it, but there are few good reasons for doing so. It is normally preferable to raise the limit for fast unique table growth. (See below.)

```
Limit for fast unique table growth: 1363148
```

See Section 4.5 and the comments above about reclaimed nodes and hard limit for the cache size. This value was chosen automatically by CUDD for a datasize limit of 1 GB.

```
Maximum number of variables sifted per reordering: 1000
Maximum number of variable swaps per reordering: 2000000
Maximum growth while sifting a variable: 1.2
```

Lowering these numbers will cause reordering to be less accurate and faster. Results are somewhat unpredictable, because larger BDDs after one reordering do not necessarily mean the process will go faster or slower.

```
Dynamic reordering of BDDs enabled: yes
Default BDD reordering method: 4
Dynamic reordering of ZDDs enabled: no
Default ZDD reordering method: 4
```

These lines tell whether automatic reordering can take place and what method would be used. The mapping from numbers to methods is in `cudd.h`. One may want to try different BDD reordering methods. If variable groups are used, however, one should not expect to see big differences, because CUDD uses the reported method only to reorder each leaf variable group (typically corresponding present and next state variables). For the relative order of the groups, it always uses the same algorithm, which is effectively sifting.

As for enabling dynamic reordering or not, a sensible recommendation is the following: Unless the circuit is rather small or one has a pretty good idea of what the order should be, reordering should be enabled.

```
Realignment of ZDDs to BDDs enabled: no
Realignment of BDDs to ZDDs enabled: no
Dead nodes counted in triggering reordering: no
Group checking criterion: 7
Recombination threshold: 0
Symmetry violation threshold: 0
Arc violation threshold: 0
GA population size: 0
Number of crossovers for GA: 0
```

Parameters for reordering. See the documentation of the functions used to control these parameters for the details.

```
Next reordering threshold: 100000
```

When the number of nodes crosses this threshold, reordering will be triggered. (If enabled; in this case it is not.) This parameter is updated by the package whenever reordering takes place. The application can change it, for instance at start-up. Another possibility is to use a hook function (see Section 3.16) to override the default updating policy.

4.8.3 Extended Statistics and Reporting

The following symbols can be defined during compilation to increase the amount of statistics gathered and the number of messages produced by the package:

- DD_STATS;
- DD_CACHE_PROFILE;
- DD_UNIQUE_PROFILE.
- DD_VERBOSE;

Defining DD_CACHE_PROFILE causes each entry of the cache to include an access counter, which is used to compute simple statistics on the distribution of the keys.

4.9 Guidelines for Documentation

The documentation of the CUDD functions is extracted automatically from the sources by Stephen Edwards's extdoc. (The Ext system is available via anonymous FTP from `ic.eecs.berkeley.edu`.) The following guidelines are followed in CUDD to insure consistent and effective use of automatic extraction. It is recommended that extensions to CUDD follow the same documentation guidelines.

- The documentation of an exported procedure should be sufficient to allow one to use it without reading the code. It is not necessary to explain how the procedure works; only what it does.
- The *SeeAlso* fields should be space-separated lists of function names. The *SeeAlso* field of an exported procedure should only reference other exported procedures. The *SeeAlso* field of an internal procedure may reference other internal procedures as well as exported procedures, but no static procedures.
- The return values are detailed in the *Description* field, not in the *Synopsis* field.
- The parameters are documented alongside their declarations. Further comments may appear in the *Description* field.
- If the *Description* field is non-empty—which is the normal case for an exported procedure—then the synopsis is repeated—possibly slightly changed—at the beginning of the *Description* field. This is so because extdoc will not put the synopsis in the same HTML file as the description.
- The *Synopsis* field should be about one line long.

5 The C++ Interface

5.1 Compiling and Linking

To build an application that uses the CUDD C++ interface, you should add

```
#include "cuddObj.hh"
```

to your source files. In addition to the normal CUDD libraries (see Section 3.1) you should link `libobj.a` to your executable. Refer to the `Makefile` in the top level directory of the distribution for further details.

5.2 Basic Manipulation

The following fragment of code illustrates some simple operations on BDDs using the C++ interface.

```
Cudd mgr(0,0);
BDD x = mgr.bddVar();
BDD y = mgr.bddVar();
BDD f = x * y;
BDD g = y + !x;
cout << "f is" << (f <= g ? "" : " not")
     << " less than or equal to g\n";
```

This code creates a manager called `mgr` and two variables in it. It then defines two functions `f` and `g` in terms of the variables. Finally, it prints a message based on the comparison of the two functions. No explicit referencing or dereferencing is required. The operators are overloaded in the intuitive way. BDDs are freed when execution leaves the scope in which they are defined or when the variables referring to them are overwritten.

6 Acknowledgments

The contributors: Iris Bahar, Hyunwoo Cho, Erica Frohm, Charlie Gaona, Cheng Hua, Jae-Young Jang, Seh-Woong Jeong, Balakrishna Kumthekar, Enrico Macii, Bobbie Manne, In-Ho Moon, Curt Musfeldt, Shipra Panda, Abelardo Pardo, Bernard Plessier, Kavita Ravi, Hyongkyoon Shin, Alan Shuler, Arun Sivakumaran, Jorgen Sivesind.
The early adopters: Gianpiero Cabodi, Jordi Cortadella, Mario Escobar, Gayani Gamage, Gary Hachtel, Mariano Hermida, Woohyuk Lee, Enric Pastor, Massimo Poncino, Ellen Sentovich, the students of ECEN5139.

I am also particularly indebted to the following people for in-depth discussions on BDDs: Armin Biere, Olivier Coudert, Arie Gurfinkel, Geert Janssen, Don Knuth, David Long, Jean Christophe Madre, Ken McMillan, Shin-Ichi Minato, Jaehong Park, Rajeev Ranjan, Rick Rudell, Ellen Sentovich, Tom Shiple, Christian Stangier, and Bwolen Yang.

Special thanks to Norris Ip for guiding my faltering steps in the design of the C++ interface. Gianpiero Cabodi and Stefano Quer have graciously agreed to let me distribute their dddmp library with CUDD.

Masahiro Fujita, Gary Hachtel, and Carl Pixley have provided encouragement and advice.

The National Science Foundation and the Semiconductor Research Corporation have supported in part the development of this package.

References

[1] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of the International Conference on Computer-Aided Design*, pages 188–191, Santa Clara, CA, November 1993.

[2] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. Presented at the International Workshop on Logic Synthesis, Granlibakken, CA, May 1995.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, pages 40–45, Orlando, FL, June 1990.

[4] R. K. Brayton et al. VIS: A system for verification and synthesis. Technical Report UCB/ERL M95/104, Electronics Research Lab, Univ. of California, December 1995.

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[6] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for variable ordering of OBDDs. Presented at the International Workshop on Logic Synthesis, Granlibakken, CA, May 1995.

[7] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, 39(5):710–713, May 1990.

[8] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation*, pages 50–54, Amsterdam, February 1991.

[9] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *J. SIAM*, 10(1):196–210, 1962.

[10] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Proceedings of the International Conference on Computer-Aided Design*, pages 472–475, Santa Clara, CA, November 1991.

[11] S.-W. Jeong, T.-S. Kim, and F. Somenzi. An efficient method for optimal BDD ordering computation. In *International Conference on VLSI and CAD (ICVC'93)*, Taejon, Korea, November 1993.

[12] S.-I. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the Design Automation Conference*, pages 272–277, Dallas, TX, June 1993.

[13] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proceedings of the International Conference on Computer-Aided Design*, pages 74–77, San Jose, CA, November 1995.

[14] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 628–631, San Jose, CA, November 1994.

[15] B. F. Plessier. *A General Framework for Verification of Sequential Circuits*. PhD thesis, University of Colorado at Boulder, Dept. of Electrical and Computer Engineering, 1993.

[16] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, Santa Clara, CA, November 1993.

[17] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, pages 328–333, Cambridge, MA, October 1992.

Index