18-760 Fall'01
About CUDD: the U. Colorado BDD Package

Using BDDs in CUDD

The BDD package we will be using is from the University of Colorado (CU), and is appropriately named **CUDD**. It has an overabundance of BDD functionality, and is generally considered the cleanest and best-maintained BDD package that is freely available. In addition, it supports Multi-Terminal-BDDs, called **MTBDDs** (though they call them *Algebraic Decision Diagrams* or ADDs), and has both C and C++ interfaces.

In CUDD, the C++ interface (like most object-oriented systems) is less efficient, but much *much* easier to deal with. In particular, the C++ interface constructors and destructors handle BDD-node reference counting and garbage collection, and there are overloaded operators that allow you to do things like the following:

```
BDD x = BDDManager.bddVar();
BDD y = BDDManager.bddVar();
BDD z = x + y; // compute x OR y
```

The same code in the C interface requires us to handle all memory allocation ourselves, and would look like the following:

```
DdNode * x,y,z;
x = Cudd_bddVar(BDDManager);
Cudd_Ref( x ); /* increment node x's reference count */
y = Cudd_bddVar(BDDManager);
Cudd_Ref( y );
z = Cudd_bddOr(BDDManager,x,y);
Cudd_Ref( z );
Cudd_RecursiveDeref( x ); /* free up x */
Cudd_RecursiveDeref( y ); /* free up y */
```

The directory /afs/ece/class/ee760/proj2/cudd_example contains an example C++ program that does some very basic manipulation of BDDs and MTBDDs (a.k.a. ADDs). The first thing you should do is copy this entire directory into your working directory, type "make" and make sure it compiles. If so, try running it as "cudd_example". If everything looks right (no coredumps or anything), then you should be fine editing the Makefile to suit your needs. If you have trouble, send mail to the TAs.

Using the C++ interface to CUDD is really simple. The first thing that has to happen before any BDD or MTBDD operations is to declare and initialize the BDD manager. This will create the Unique Table and other structures that are required for all other operations. The manager is of type "Cudd" and is declared as :

```
Cudd mgr(0,0);
```

Once this has been done, you can start creating BDD variables and referring to the constant nodes. The result of the third call below will be a BDD with one non-terminal node having the next-lowest unused index, having it's hison pointing to the terminal "1" and it's loson pointing to the terminal "0":

```
BDD one = mgr.bddOne(); // constant '1' BDD
BDD zero = mgr.bddZero(); // constant '0' BDD
BDD x = mgr.bddVar(); // new variable
```

If you want to explicitly create a variable with a certain index (for example, "5"), or return an already existing version, you can do the following:

```
BDD x = mgr.bddVar(5);
```

Since C++ allows overloading of operators, most of the common operations are extremely simple:

```
BDD foo = x + y; // OR
BDD rob = x ^ y; // XOR
BDD rut = ! rob; // NOT
BDD en = foo * rob; // AND
BDD bar = x.Ite(rob,rut); // bar = ITE(x,rob,rut)
```

Quantification is a little strange, but it works. The strange part is that you have to create a single "auxilliary" BDD that is the AND of all of the variables you want to quantify with respect to. So to compute:

$$\forall(x, y, z)F = G$$

```
BDD c = x * y * z; // build the cube xyz
BDD G = F.UnivAbstract(c);//do G = forall(xyz)F
```

Similarly for existential quantification:

```
BDD H = F.ExistAbstract(c);//do H = there_exists(xyz)F
```

Using ADDs in CUDD

The MTBDD operations are also pretty straightforward. The only confusing thing is that the CUDD authors for some reason didn't like the name MTBDD, so they refer to them as ADDs. To create new constants, we can do the following:

```
ADD x = mgr.constant(5.4);
```

Again, the operators are overloaded as you might expect:

```
ADD x = foo + bar; // addition
ADD y = foo * bar; // multiplication
```

Converting between BDDs and MTBDDs (ADDs) is also straightforward. The "Add" method converts a BDD into an MTBDD. The resulting MTBDD will be essentially the same as the original BDD except that it won't have any negated arcs. It is mostly useful when you need to to an MTBDD ITE operation, since all three arguments have to be MTBDDs. Suppose you wanted to create an MTBDD "R" that had the value of "5" for all cases where BDD x is true, and the value "20" where BDD x is false:

```
ADD xAdd = x.Add(); // build an MTBDD version of x
ADD R = xAdd.Ite( mgr.constant(5), mgr.constant(20));
```

Other useful MTBDD operations for this project are the following:

```
ADD x, y, z;
       // assume we initialize x,y,z here
ADD foo = x.Minimum(y); //foo = min(x,y)
ADD bar = z.Maximum(y); //bar = max(z,y)
```

To get the maximum or minimum valued leaf node in an MTBDD :

```
ADD foo;
ADD maxAdd = foo.FindMax(); // get a pointer to the leaf
double maxval = maxAdd.Value(); // get the leaf value

ADD minAdd = foo.FindMin();
double minval = minAdd.Value();
```

To create a BDD by replacing all MTBDD terminals >= 10 with ‘1’, and the rest with ‘0’:

```
ADD foo;
BDD makesX = foo.BddThreshold( 10 );
```

To ask CUDD for an assignment of variables that makes a BDD true, you use the function First-Cube() to get a pointer to an array of int’s. The array will be filled with 0,1,2 for each variable index, where 0 and 1 mean to set the variable to 0 and 1 respectively, and 2 means the variable at that index is a don’t-care (the code below just sets don’t-cares to 1).

```
BDD M; // some BDD, lets assume it's initialized
int * cube; // a pointer to an array of ints
CUDD_VALUE_TYPE value; // useless required variable
M.FirstCube(&cube,&value); // get an assignment
for(int i = 0; i < NumInputs; i++ ){
        cout << "  Input " << i << " = " <<
          ( cube[i] == 0 ? "0":"1" ) << "\n";
        }
```