# CS 267: Automated Verification

Notes on CUDD Package

Instructor: Tevfik Bultan

# CUDD Package

- It is a BDD package that is implemented in C/C++
- Useful functions for implementing symbolic model checking:
  - `Cudd_Init`
  - `Cudd_Ref`
  - `Cudd_RecursiveDeref`
  - `Cudd_ReadOne`
  - `Cudd_ReadLogicZero`
  - `Cudd_bddIthVar`
  - `Cudd_bddAnd`
  - `Cudd_bddOr`
  - `Cudd_Not`
  - `Cudd_bddPermute`
  - `Cudd_bddAndAbstract`

# Initializing a BDD manager

```
bddMgr = Cudd_Init(numVar,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0);
```

numVar is the number of Boolean variables in the BDDs we will create using this BDD manager

# Creating constant BDDs

- `Cudd_ReadOne`
- `Cudd_ReadLogicZero`

- These functions return the BDDs that correspond to the boolean logic formulas “true” and “false” respectively

```
tmp1 = Cudd_ReadOne(bddMgr);
tmp2 = Cudd_ReadLogicZero(bddMgr);
```

# Variables

- `Cudd_bddIthVar`

- This function can be used for creating boolean logic formulas that correspond to a single variable

```
tmp = Cudd_bddIthVar(bddMgr, i);
```

After this assignment, tmp is a BDD that corresponds to the Boolean logic formula $x_i$

# Memory management

- `Cudd_Ref`
- `Cudd_RecursiveDeref`

- CUDD uses reference counts for memory management
- Calling Ref function increases the reference count
- Calling RecursiveDeref decreases the reference count
- CUDD library periodically does garbage collection to free BDD nodes that have a reference count that is zero

# Constructing formulas

- `Cudd_bddAnd`
- `Cudd_bddOr`
- `Cudd_Not`

- These functions can be used to construct new BDDs from existing BDDs
- Using these functions (and the earlier ones) one can iteratively construct a BDD for a Boolean logic formula

# Constructing Formulas

```
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;
        }
```

# Variable Renaming

- `Cudd bddPermute`

- This function can be used for renaming variables in a given bdd

```
outbdd = Cudd_bddPermute(bddMgr, inbdd, permutation);
Cudd_Ref(outbdd);
Cudd_RecursiveDeref(manager,inbdd);
```

- permutation is an integer array that identifies which variable index should be mapped to which variable index
- For example, to rename variable $x_0$ as $x_2$ and $x_1$ as $x_3$ and visa versa, you need to create the permutation array:

  [2, 3, 0, 1]
- If the inbdd corresponds to the formula $x_0 \wedge x_1$ then after the call the outbdd will correspond to $x_2 \wedge x_3$

# Image Computation

- `Cudd_bddAndAbstract`
- This function can be used for image computation (EX is called backward image)

```
out = Cudd_bddAndAbstract(bddMgr, bdd1, bdd2, cube);
```

- The out bdd corresponds to conjunction of bdd1 and bdd2 followed by existential elimination of variables in the cube
- Cube is a conjunction of variables that will be existentially quantified (i.e. abstracted)
- For example to existentially quantify $x_0$, $x_1$, $x_2$, the cube we construct must be the BDD for the formula $x_0 \wedge x_1 \wedge x_2$
- Then the result of the above call will be equivalent to:

$$f_{out} = \exists x_0 , \exists x_1 , \exists x_2 , f_{bbd1} \wedge f_{bdd2}$$