@c

CUDDAUX

Auxiliary Library for BDDs library CUDD
Edition 1.0, 14 May 2002

by Bertrand Jeannet

Table of Contents

1 Introduction

This library provides additional functions to the CUDD library, available at http://vlsi.colorado.edu/software.html. Many of them allow to replace 0-1 ADDs by BDDs in if-then-else and generalized cofactors operations, some offer functions for ADDs which are counterparts of functions already implemented for BDDs, others offer a different interface, and last really new functions are implemented.

As the library uses *internal* operations of CUDD, it is mandatory to compile it with the same option than the one applied to CUDD. Among others, check data alignement options !

This library is necessary to the OCAML interface to CUDD that I developped and which is available at http://pop-art.inrialpes.fr/people/bjeannet/mlcuddidl/index.html.

To compile it, set the flags in '`Makefile.config.model`', copy it to '`Makefile.config`', and type *`make all`*. To use it, you should include the header file '`cuddaux.h`' and link your program with '`-lcuddaux`'.

2 Functions

2.1 ADD ITE Functions

`DdNode* Cuddaux_addIte` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `g`, *DdNode* `h`) [Function]

`DdNode* Cuddaux_addIteConstant` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `g`, *DdNode* `h`) [Function]

`DdNode* Cuddaux_addEvalConst` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `g`) [Function]

Same as `Cudd_addIte`, `Cudd_addIteConstant` and `Cudd_addEvalConst`, but here f is a BDD node instead of a 0-1 ADD node. g and h are ADDs.

`DdNode* Cuddaux_addBddAnd` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `g`) [Function]

Same as `Cuddaux_addIte(dd,f,g,Cudd_ReadBackground(dd))`: selects in g the valuations that satisfies f and makes the other valuations lead to the background node.

2.2 Bridge Function for ADDs

`DdNode* Cuddaux_addTransfer` (*DdManager* `ddS`, *DdManager* `ddD`, *DdNode* `f`) [Function]

`Cudd_bddTransfer`-like function for ADDs.

2.3 Generelized Cofactor functions

`DdNode* Cuddaux_bddRestrict` (*DdManager * dd, DdNode * f, DdNode * c*) [Function]

Same as `Cudd_bddRestrict`, but the *real* result is returned instead of the smallest (in term of BDD nodes) among the result and the argument.

`DdNode* Cuddaux_addRestrict` (*DdManager * dd, DdNode * f, DdNode * c*) [Function]

`DdNode* Cuddaux_addConstrain` (*DdManager * dd, DdNode * f, DdNode * c*) [Function]

Same as `Cudd_addRestrict` and `Cudd_addConstrain`, but here c is a BDD node instead of a 0-1 ADD node.

`DdNode* Cuddaux_bddTDRestrict` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `c`) [Function]

`DdNode* Cuddaux_bddTDConstrain` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `c`) [Function]

`DdNode* Cuddaux_addTDRestrict` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `c`) [Function]

`DdNode* Cuddaux_addTDConstrain` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `c`) [Function]

Top-Down Restrict and Constrain operations from P. Raymond. Good but expensive. For BDD versions, f and c are BDD nodes, for ADD versions, f is a ADD node and c a BDD node.

`DdNode* Cuddaux_bddTDSimplify` (*DdManager* `dd`, *DdNode* *inf*, *DdNode* *sup*) [Function]

Given two BDDs *inf* and *sup* such that *inf*`=>`*sup*, compute the smallest BDD in the interval. Core of the BDD version of the previous generalized cofactor operations.

`DdNode* Cuddaux_addTDSimplify` (*DdManager* `dd`, *DdNode* `f`*) [Function]
Given an ADD f with background value, return a small ADD r without background value that coincides with f outside the background value. In other words, for any valuation v, either $f(v) = background$ and $r(v)$ is equal to a non background leaf of f, either $f(v) = r(v)$. Core of the ADD version of the previous generalized cofactor operations.

2.4 Composition functions for ADDs

`DdNode* Cuddaux_addCompose` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `g`, *int* `v`) [Function]
`DdNode* Cuddaux_addVectorCompose` (*DdManager* `dd`, *DdNode* `f`, *DdNode** `vector`) [Function]
Same as `Cudd_addCompose` and `Cudd_addVectorCompose`, but the substitution function (resp. the vector of substitution functions) is a BDD (resp. an array of BDDs) instead of a 0-1 ADD.

`DdNode* Cuddaux_addVarMap` (*DdManager* `dd`, *DdNode* `f`) [Function]
The equivalent for ADDs of the function `Cudd_bddVarMap`.

`int Cuddaux_SetVarMap` (*DdManager* `dd`, *int** `array`) [Function]
Offers the same functionality than `Cudd_SetVarMap`, but with a different interface, which match the interface for the permutation functions. The array *array* gives for each variable of index i present in the manager *dd* the index of the variable to be substituted to i.

2.5 Miscellaneous functions

These functions offers functionality not directly present in CUDD.

`int Cuddaux_IsVarIn` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `var`) [Function]
f is a BDD or an ADD and *var* is a BDD or ADD projection function. Returns `1` whenever *var* occurs in *f*, 0 otherwise. Using this function is more efficient than computing the support and test inclusion of the variable in it. No new node is created.

`DdNode* Cuddaux_bddCubeUnion` (*DdManager* `dd`, *DdNode* `f`, *DdNode* `g`) [Function]
f and *g* are BDD cubes. The function returns the smallest cube which is implied both by *f* and by *g*. It is functionally equivalent to `Cudd_FindEssential(dd,Cudd_bddOr(dd,f,g))`.

`list_t` [Datatype]

```
typedef struct list_t {
  struct list_t* next;
  DdNode* node;
} list_t;
```

`list_t* Cuddaux_NodesBelowLevel` (*DdManager* `dd`, *DdNode* `f`, *int* `level`) [Function]
f is a ADD or a BDD and *level* a variable level. The functions collects in the result all the (regular) nodes pointed by *f*, indexed by a variable of level greater or equal than *level*, and encountered first in the top-down exploration (i.e., whenever a node is collected, its sons are not collected). This function allows for instance to collect efficently all the terminal nodes of an ADD *f*. The result of type `list_t*` is allocated by the function. The nodes in the list are not referenced. No new node is created. Returns `NULL` if not successfull.

`void list_free` (*list_t** `l`) [Function]

Frees the memory occupied by the list *l*.

`DdNode* Cuddaux_addGuardOfNode` (*DdManager** `dd`, *DdNode** `f`, *DdNode** `h`) [Function]

f and *h* are ADDs. Returns a BDD equal to the sum of the paths that leads from the root node *f* to the node *h* in the ADD *f*. If *h* is not in the graph of *f*, the logical false node is returned. Can be used for instance to compute the guard of a terminal node *h* in an ADD *f*.

Appendix A Index

A.1 Datatypes index

A.2 Functions Index

C

L