An Introduction to Zero-Suppressed Binary Decision Diagrams

Alan Mishchenko

Berkeley Verification and Synthesis Research Center
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
alanmi@berkeley.edu

February 5, 2014

Abstract

Zero-suppressed binary Decision Diagrams (ZDDs) have emerged as an efficient way of solving problems in set theory. This tutorial presents ZDDs and assumes that the reader is familiar with Boolean algebra and Binary Decision Diagrams, without prior knowledge of ZDDs. The case studies include the computation of the union of two sets, the generation of all primes of a Boolean function, and the computation of the Irredundant Sum-of-Products of an incompletely specified Boolean function, the latter being perhaps the most practical among the ZDD-based procedures. This tutorial contains annotated source code in C of a ZDD-based procedure implemented in the CUDD decision diagram package.

The appendix contains a list of 30+ ZDD procedures included in the decision diagram package CUDD [36] and 50+ additional ZDD procedures included in the EXTRA library [30] available as a public-domain extension of CUDD.

Table of Contents

1 Introduction

Binary Decision Diagrams (BDDs) [4] and their variations are a known representation of Boolean functions used in various practical applications, in particular, those related to computer-aided design (CAD). For example, in formal verification, the state space of a discrete system can be explored by representing and manipulating the transition relation and the set of reached states using BDDs [7]. In logic synthesis, BDDs has been used to represent cube covers in Sum of Product (SOP) minimization [37][10].

The use of decision diagrams has led to the development of *implicit* algorithms operating on discrete objects, such as state sets or cube covers, by encoding them in a graph-based data-structure. Such graphs are manipulated by considering their nodes without enumerating individual objects. This often results in a faster computation because the size of the graphs can be much smaller than the number of encoded objects.

Decision diagrams are also useful because they provide a *canonical* representation of Boolean functions. Canonical means that, under certain conditions, an object has only one representation of this kind. This property is important in formal verification because in order to prove the identity of two objects, it is sufficient to build their canonical representations and show that these representations are identical. In logic synthesis, canonical representations can be used as signatures for storing objects in hash tables.

The experience of using BDDs in numerous applications shows that they are not a panacea for all types of problems. In some cases, due to the specific properties of discrete data arising in a particular setting, BDDs grow large making processing inefficient or impossible. In particular, this is true when an application deals with sparse sets represented by their characteristic functions [5].

A set is *sparse* if the number of elements in it is much smaller than the total number of elements that *can* appear in the set. Cube covers are an example of sparse sets, because a typical cube contains only a few literals, out of all possible literals that can appear in the cube. The maximum number of literals is reached when a cube is a minterm. In this case, each variable appears as either negative or positive literal. The sparseness of a minterm is ½, because it contains exactly one half of all possible literals.

The problem of the prohibitively large size of a sparse set representation can be remedied by introducing a different type of decision diagrams, called Zero-suppressed binary Decision Diagrams (ZDDs) [21]. These diagrams are similar to BDDs with one difference, which explains the improved efficiency of ZDDs when handling sparse sets.

While BDDs are better for representing functions, ZDDs are better for representing covers. Additionally, there are efficient procedures to perform conversions between them. Taken together, BDDs and ZDDs provide a powerful framework to solve problems in logic synthesis, such as two-level sum-of-product (SOP) minimization [9], three-level minimization, factorization [27][35], and decomposition [18].

The use of ZDDs is not limited to logic synthesis. They have been used, independently of BDDs, in a number of applications, ranging from the graph-theory problems to handling polynomials and regular expressions. (See Section 8 for what is intended to be a complete list of ZDD applications published to date.)

This tutorial paper is an introduction to ZDDs for the reader familiar with Boolean algebra and basic principles of BDDs. The goal is to present ZDDs in the context of three types of applications:

- ZDDs for sets,
- ZDDs for cube covers,
- mixed BDDs/ZDDs for functions and cube covers.

To this end, we first discuss the basic principles and uses of ZDDs. In particular, Section 3 focuses on the main differences between BDDs and ZDDs when it comes to representing Boolean functions, sets, and cube covers.

In Section 4, we classify and discuss the elementary ZDD operators provided by a DD package. Next, we explore the generic structure of a DD-based recursive procedure, which is important for understanding the following sections.

Section 5 shows how ZDDs can be used to manipulate sets. The set-union operator is considered an illustrative example. The complete source code of this operator in the CUDD package is included and explained assuming the reader's familiarity with C programming language.

Section 6 introduces the basics of ZDDs for manipulation of cube covers using the cover-product operator as an illustrative example.

Section 7 contains two practically important examples of mixed ZDD/BDD applications: generation of a ZDD representing all primes for a completely specified Boolean function given by a BDD, and computation of a ZDD representing an Irredundant Sum-of-Products of the incompletely specified Boolean function.

Section 8 provides a complete list of ZDD applications published to date, followed by conclusions in Section 9.

Two appendices contain annotated lists of ZDD-based procedures implemented in the CUDD package [36] and the EXTRA library [30], respectively.

Ideally, after completing the tutorial, the reader should be able to write his or her own ZDD-based procedures using the CUDD package.

2 Definitions

A *literal* is a Boolean variable or its negation, e.g. $a, \bar{b}$. A *product,* or cube, is a Boolean product of literals, e.g.

$\bar{b}c$. A *cover* is a set of products. The *cardinality* of the cover is the number of cubes in the cover. A *complement*, or a negative phase, of cover S is a cover T such that the union of S and T is a *tautology*, that is, the Boolean function constant-1.

Let f: $B^n \rightarrow B$, $B \in \{0,1\}$, be a completely specified Boolean function (CSF). Let F: $B^n \rightarrow \{0,1,-\}$ be an incompletely specified Boolean function (ISF) represented by two CSFs: the *on-set*, $f^1 = \{x \mid F(x) = 1\}$ and the *don't-care-set*, $f^{dc} = \{x \mid F(x) = -\}$.

A CSF can be represented by a set of cubes. This is known as a two-level *sum-of-product representation* (SOP). An SOP, or cover, is *irredundant,* if no cubes can be removed without changing the value of the represented function and no two cubes can be combined into one cube.

A function *essentially* depends on a variable if the variable appears in an irredundant SOP or in a reduced ordered BDD of the Boolean function. The variable set X, on which f essentially depends, is called *support* of f.

A *minterm* is the smallest cube, in which every variable is represented by either a negative or a positive literal. A variable that is not represented by a negative or a positive literal in the cube is said to have the *don't-care literal*. Each don't-care literal can be seen as a sum of the positive literal and the negative literal. This leads to splitting the cube into two smaller cubes. In general, if a cube has k don't-care literals, it is equal to the sum of 2^k minterms, created by splitting each don't-care literal.

The *area* of the Boolean space covered by the cube consists of all minterms created by splitting don't-care literals of the cube. Two areas of the Boolean space *overlap* if they have common minterms.

Two cubes are *disjoint* if the areas of the Boolean space covered by the cubes do not overlap. Two covers are *disjoint* if the areas covered by their cubes do not overlap.

Given a Boolean function f, the *negative cofactor* of f with respect to (w.r.t.) variable x is the Boolean function $f_{x=0} = f(x = 0)$. Similarly, the *positive cofactor* is the Boolean function $f_{x=1} = f(x = 1)$.

The following terminology is accepted in BDD research [4]. The BDD represents the function as a rooted directed acyclic graph. Each non-constant node N is labeled by a variable v and has edges directed towards two successor (children) nodes, else(N) and then(N), representing the cofactors of N with respect to (w.r.t.) v. Each constant node is labeled with 0 or 1. For a given assignment of the variables, the value of the function is found by tracing a path from the root to a constant vertex following the branches indicated by the values assigned to the variables. The function value is given by the constant vertex label. For example, Fig. 1 (left) shows the BDD of the Boolean function F = ab + cd. The edges are directed downwards. The dashed edges (solid) edges correspond to v = 0 (v = 1).

In this paper, a *set* is a collection of elements from a finite domain. In other words, only the presence of elements matters, not their order. Sets {a,b} and {b,a} are identical, while sets {a,b,c} and {a,b} are different. Sometimes it is convenient to assume that the elements are initially ordered and appear in the sets only in that order. Assuming that a precedes b in the order, both {a,b} and {b,a} are represented as {a,b}.

In the following, we use the terms "procedure", "functions", "routine" and "operator" interchangeably to denote a fragment of functionality implemented with decision diagrams.

3 Comparing BDDs and ZDDs

Both BDDs and ZDDs can be seen as decision trees, simplified using two reduction rules that guarantee the canonicity of the representation. The second reduction rule (merging of isomorphic subgraphs) holds for both BDDs and ZDDs; however, they differ in the first reduction rule (node elimination).

For BDDs, the node is removed from the decision tree if both its edges point to the same node. For ZDDs, the node is removed if its positive edge (then-edge) points to the constant node 0. This variation in the rule, as mentioned before, explains the improved efficiency of ZDDs when handling sparse sets and the semantic differences between the two types of diagrams.

One way of understanding the principles of ZDDs is to compare them with BDDs for simple illustrative functions while keeping in mind their main difference.

3.1 Boolean functions

It can be shown that, in a BDD, all paths from the root to constant node 1 can be seen as cubes constituting a disjoint cover of the function. A variable is present in the *positive (negative) polarity* in the corresponding cube if the path contains the 1-edge (0-edge) of a node labeled by this variable; the variable is *absent* in the cube if the path does not go through a node labeled by this variable.

In a ZDD for the same function, all paths from the root to constant 1 also represent a disjoint cover of the function. (This cover is the same if the variable ordering is the same in both diagrams.) A variable is present in the *positive polarity* in the corresponding cube if the path goes through the 1-edge of a node labeled with this variable. A variable is present in the *negative polarity* in the cube if the path goes through the 0-edge *or* if the path does not go through a node labeled by this variable. A variable is *absent* in a cube, if the path goes through a node labeled by this variable and both edges of the node point to the same node.

Consider a BDD and a ZDD of the function F = ab + cd shown in Fig. 1. Both diagrams can be used to trace the disjoint cover of the function: $\{ab, \bar{a}cd, a\bar{b}cd\}$. As can be

seen from Fig. 1, the size of the ZDD, expressed as the number of nodes in the diagram, is almost two times larger than that of the BDD. This is because ZDDs are not as efficient as BDDs when it comes to representing typical Boolean functions.

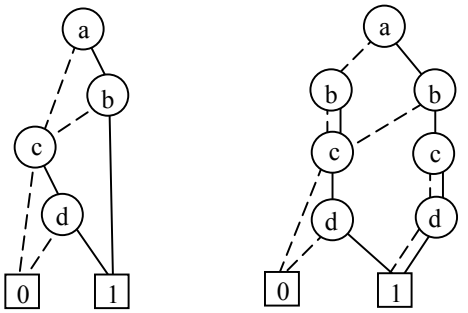

Figure 1. BDD and ZDD for F = ab + cd.

3.2 Sets of subsets

Classical BDDs represent completely specified Boolean functions. In order to represent sets and sets of subsets, a set is put in one-to-one correspondence with its characteristic function [5].

Informally, the characteristic function of a set of subsets is a CSF that depends on as many input variables as there are elements that can potentially appear in a subset. For each subset in the set, one minterm is added to the on-set of the characteristic function. In this minterm, variables appear in positive (negative) polarities if they are present (absent) in the set. In the extreme case of one subset (that is, if a set of subsets contains only one subset), the characteristic function has only one minterm in its on-set.

For example, given three elements (a,b,c), consider the set of subsets { {a,b}, {a,c}, {c} }. If we associate each element with a binary variable having the same name, the characteristic function of the set of subsets is $F = ab\bar{c} + a\bar{b}c + \bar{a}\bar{b}c$. The first minterm corresponds to the subset {a,b}, the second to {a,c}, the third to {c}.

Important for our discussion are the following observations. The empty subset is represented by the minterm $F = \bar{a}\bar{b}\bar{c}$, while the subset containing all elements is represented by F = abc. The empty set is represented by the characteristic function F = 0, while the set of subsets composed of all possible subsets is represented by the characteristic function F = 1. The latter is obvious if we observe that the constant-0 function has no on-set minterms, while the constant-1 function has 2^n on-set minterms, corresponding to the complete Boolean space.

Note that there is a difference between the empty set of subsets and the set of subsets containing the empty set. The former has the characteristic function equal to constant 0, while the latter has the characteristic function $F = \bar{a}\bar{b}\bar{c}$.

A characteristic function can be represented using a BDD or a ZDD. The two representations of the set of subsets {{a,b}, {a,c}, {c}} are given in Fig. 2.

In both diagrams, there are three paths from the root node (on top) to the constant node 1 (at the bottom) corresponding to the subsets {a,b}, {a,c}, and {c}. The encoding of the variables is discussed in section 3.1.

Note that the size of the ZDD in Fig. 2 is smaller than that of the BDD. It can be proved that an upper bound on the size of the ZDD is the total number of elements appearing in all subsets of the set of subsets. Meanwhile, an upper bound on the size of the BDD is given by the number of subsets multiplied by the number of all elements that can appear in them. This observation shows that ZDDs are more compact when representing sets of subsets. The above theoretical upper bound on the ZDD size is rarely reached; in practice, ZDDs tend to be even more compact.

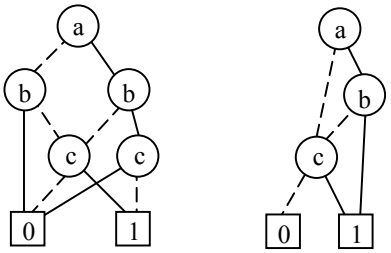

Figure 2. The BDD and the ZDD for the set of subsets {{a,b}, {a,c}, {c}}.

3.3 Cube covers

Let us now consider the ZDD representation of a cube cover. First, it is necessary to introduce additional variables, because a BDD and a ZDD depending on the primary input variables represents a Boolean function and one *disjoint* cover of this function. To represent an *arbitrary* cover, each primary inputs is mapped into two variables: one of them stands for the positive literal and another for the negative literal. These variables are often kept adjacent in the variable order. Similarly to the set of subsets, a cube cover is represented by its characteristic function introduced as follows:

- The characteristic function of the cube cover depends on 2*n variables representing literals, where n is the number of variables in the original Boolean function.
- For each cube of the cover, one minterm is added to the on-set of the characteristic function.
- The minterm has those variables in the positive polarity that correspond to literals present in the cube and those variables in the negative polarity that correspond to literals missing in the cube.

For example, to represent arbitrary covers of the four-variable function F = ab + cd, eight variables are used:

(a_1, a_0, b_1, b_0, c_1, c_0, d_1, d_0). These variables correspond to the positive and negative literals of each input variable.

Consider the cover {ab, cd}. The characteristic functions of this cover is: $\chi = a_1\bar{a}_0b_1\bar{b}_0\bar{c}_1\bar{c}_0\bar{d}_1\bar{d}_0 + \bar{a}_1\bar{a}_0\bar{b}_1\bar{b}_0c_1\bar{c}_0d_1\bar{d}_0$. ZDD for the characteristic function is shown in Fig. 3.

Unlike the BDD for function χ, which depends on all eight variables, the ZDD depends on four variables only. These are variables that appear in the characteristic function in the positive or negative polarity and correspond to literals actually present in the cover. All other variables are missing in the ZDD, because according to the ZDD reduction rules a variable missing on a path is interpreted as a variable taking value 0 in the minterm of the characteristic function. This property makes ZDDs ideal for representing and manipulating large cube covers.

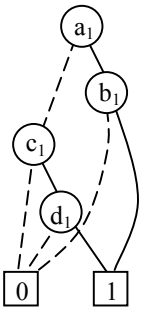

Figure 3. ZDD for the cover {ab, cd}.

The constant node 0 as a ZDD represents the empty cover. In this case, there are no assignments for which the characteristic function evaluates to 1, and therefore there are no cubes in the cover.

The constant node 1 stands for the cover containing only the *tautology cube*, that is, the cube in which all the literals are missing. Indeed, there is only one path for which the characteristic function evaluates to 1, and this path does not go through any nodes. According to the ZDD reduction rules, it means that all the variables on the path are equal to zero, which in turn means that the cube contains no literals.

The decomposition of a cover with respect to (w.r.t.) a primary input variable is the triple of covers containing: (1) cubes with the variable as the positive literal, (2) cubes with the variable as the negative literal, (3) cubes without the variable (or with the variable as a don't-care literal).

The inverse operation is the composition of the three covers into one. The composition is performed using a variable that is not currently used in the covers.

For example, consider the cover C = $ab\bar{c} + a\bar{b}d + ac + d$. Decomposing C w.r.t. the primary input variable b yields: C0 = ad, C1 = $a\bar{c}$, C2 = ac + d. The reverse operation, the composition of the covers C0, C1, and C2 w.r.t. b, which does not appear in them, produces the initial cover C.

If the cover is represented by a ZDD, the decomposition and composition operations are performed by functions of the ZDD package. In the traversal procedures presented in this paper, the functions are denoted DecomposeCover() and ComposeCover().

DecomposeCover() takes the cover and the primary input variable and returns three subcovers. ComposeCover() takes three subcovers and the primary input variable and returns the composed cover.

For further details on using ZDDs in the representation of cube covers, the reader is referred to [23][26][27] where some basic ZDD-based recursive operators are introduced and explained.

4 Basic ZDD procedures

The basic ZDD operators can be classified as follows:

4.1 Procedures dealing with functions

These procedures are similar to those developed to manipulate Boolean functions using BDDs.

- *Procedures returning elementary functions:*
  - Constant-0 ZDD (constant zero function, F = 0)
  - Universal ZDD (constant one function, F = 1)
  - Single-variable ZDD (the function equal to the elementary variable, F = v)
- *Procedures performing Boolean operations:*
  - If-Then-Else (ITE) operator. This function returns the result of applying ITE to A, B, and C:

$$ITE(A, B, C) = AB + \bar{A}C .$$

Note that the complement of a Boolean function represented as a ZDD cannot be computed by complementing a pointer to the topmost ZDD node, as in the case of a BDD with complement edges. Instead, the complement is computed as follows: $\bar{F} = ITE(F, 0, 1)$.

4.2 Procedures dealing with sets

- *Procedures returning elementary sets:*
  - Constant-0 ZDD (the empty set, {})
  - Constant-1 ZDD (the set of subsets consisting of the empty set, {{}})
  - Single-variable ZDD (the set of subsets with a subset containing element v, {{v}})
- *Procedures performing operations on the set of subsets w.r.t. to a single element (variable):*
  - Subset0(S, v) returns the set of subsets of S not containing element v.
  - Subset1(S, v) returns the set of subsets of S containing element v.
  - Change(S, v) returns the set of subsets derived from S by adding element v to those subsets that did not contain it and removing element v from those subsets that contain it.
- *Procedures performing standard set operations for two sets of subsets:*

- Union(X, Y) returns the set of subsets belonging to X or Y.
- Intersection(X, Y) returns the set of subsets belonging to both X and Y.
- Difference(X, Y) returns the set of subsets of X not belonging to Y.

It is important to distinguish single-variable ZDDs as used in the manipulation of Boolean functions and those used in the manipulation of sets of subsets. Fig. 4 shows the ZDDs for F = b, assuming that there are three variables (a,b,c). In the case of functions, this ZDD represents function F = b. In the case of sets of subsets, the ZDD represents the set of subsets containing one set composed of a single element b. In this case, the characteristic function is $F = \bar{a}b\bar{c}$.

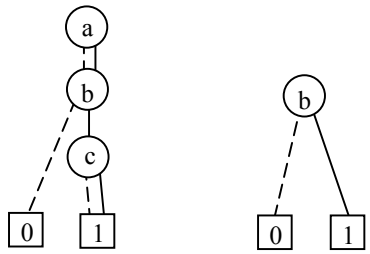

Figure 4. ZDDs for elementary variable b used in functions manipulation (left) and in set manipulation (right).

In addition to the above nine elementary operators defined for sets, the two pairs of set product and weak-division operators have been implemented using ZDDs. These two pairs of operators correspond to unate and binate algebras [23][27]. The remainder is another operator in each of these algebras. In practice, however, the remainder (%) is computed using product (*), weak-division (/), and set-difference (–) as follows: X%Y = X – X*(X/Y).

Speaking informally, in unate algebra, every literal of a cube is either present or absent, while in binate algebra every literal may be complemented, non-complemented, or missing.

Correspondingly, to manipulate sets in unate algebra every literal is encoded using one ZDD variable, while to manipulate sets in binate algebra, two literals are used, one represents the variable in positive polarity, the other presents it in negative polarity.[*]

In terms of the definitions introduced above, ZDDs used to manipulate sets of subsets implement the unate algebra, while ZDD used to manipulate the cube covers implement the binate algebra.

4.3 Structure of a recursive ZDD procedure

In this subsection, we discuss the generic structure of a recursive ZDD procedure. The presentation is also true for recursive procedures written using other types of decision diagrams, in particular, BDDs and ADDs [2]. Therefore, in this subsection we use the term "DD" instead of "ZDD".

Procedures written with DDs can be roughly divided into two classes:

- Recursive procedures that rely on the DD structure to perform computation.
- Non-recursive procedures that do not use the DD structure but may call the recursive procedures.

The former type is also known as *traversal procedures*, because, in the process of recursion, nodes of the DD are visited in the depth-first manner starting from the root. The efficiency of traversal procedures comes from the fact that, due to the caching of the intermediate results of computation, each node in the tree of recursive calls is visited at most once.

If the traversal procedure takes only one DD as an argument, the number of nodes in the tree is equal to the number of nodes in the DD. If there are more arguments, the number of nodes in the tree is bounded by the product of the number of nodes in the argument DDs. This upper bound is rarely reached in practice.

```
dd TraversalProcedure( dd A, dd B, ... )
{
   // (1) consider trivial cases
   if ( A = 0 )  return ...;
   if ( A = 1 )  return ...;
   ...

   // (2) perform a cache lookup
   R = CacheLookup( A, B, ... );
   if ( R exists )  return R;

   // (3) find the topmost variable in A, B, ...
   Var = TopMostVariable( A, B, ... );

   // (4) cofactor arguments w.r.t. Var
   A0 = Cofactor( A, Var' );
   A1 = Cofactor( A, Var  );
   ...

   // (5) recursively solve subproblems
   R0 = TraversalProcedure( A0, B0, ... );
   R1 = TraversalProcedure( A1, B1, ... );
   ...

   // (6) derive the solution of the problem
   //     from those of the subproblems
   R  = GetResult( R0, R1, ... );

   // (7) cache the result
   CacheInsert( A, B, ..., R );

   // (8) return the result
   return R;
}
```

Figure 5. Structure of a recursive DD-based procedure.

Fig. 5 shows the structure of a recursive traversal procedure. Steps (2)-(5) and (7)-(8) are similar for the majority of the recursive procedures. Steps (1) and (6) are application-specific.

In particular, step (1) solves the problem in the extreme case when the argument DDs are such that no further

[*] In the public distribution of the CUDD package, unate product and division are implemented as Cudd_zddUnateProduct() and Cudd_zddDivide(), while binate product and division are implemented as Cudd_zddProduct() and Cudd_zddWeakDiv().

recursive calls are necessary. Step (6) answers the question: How to solve the problem if partial solutions of subproblems are known? In some cases, this step requires a lot of creativity to implement. Some algorithms are not implemented as a single recursive procedure because solutions for Step (6) are not known, for example:

- Complementation of the cover represented by a ZDD. The simplest known solution requires two traversals: create the BDD of the function represented by the cover, complement the BDD (this is done in constant time for BDDs with complement edges), and compute the ZDD of the complemented cover.
- Computation of the set of dominated columns and rows in the unate covering problem represented by BDDs. So far, this problem has been solved either explicitly, without DDs, or by applying formulas with quantifiers to the BDDs of dominance relations [19].

5 Manipulation of sets

In this section, we apply the principles of traversal procedures discussed above to the computation of the union of two sets of subsets.

For example, given the two sets of subsets: A = {{a,b}, {c}} and B = {{a,b}, {a,c}}, the union of A and B is {{a,b}, {a,c}, {c}}.

```
set Union( set A, set B )
{
   // (1) consider trivial cases
   if ( A = {} ) return B;
   if ( B = {} ) return A;
   if ( A = B  ) return A;

   // (3) find the topmost variable in A and B
   var x  = TopVariable( A, B );

  // (4) cofactor arguments w.r.t. x
   set A0 = Subset0( A, x );
   set A1 = Subset1( A, x );
   set B0 = Subset0( B, x );
   set B1 = Subset1( B, x );

   // (5) recursively solve subproblems
   set R0 = Union( A0, B0 );
   set R1 = Union( A1, B1 );

   // (6) derive the solution of the problem
   //     from those of the subproblems
   set R  = CreateZdd( x, R1, R0 );

   // (8) return the result
   return R;
}
```

Figure 6. Pseudo-code of the union of two sets of subsets.

The pseudo-code of the set-union operator is shown in Fig. 6, where steps (2) and (7) (the cache lookup and insert) have been omitted for clarity. Note how the steps (1) and (6) are solved in the pseudo-code.

The trivial cases take place when at least one of the arguments is an empty set (in this case, the union is equal to the other argument) or when the arguments are equal (in this case, the union is any of the sets).

Procedures Subset0() and Subset1() compute the cofactors of the initial sets, that is, the sets that do not contain the topmost element and the sets that contain the topmost element.

The solution R of the problem can be derived from the solution of the subproblems. To get the subsets with (without) the topmost element, R0 and R1, we compute the union of the argument subsets with (without) the topmost element. This is done using two recursive calls to Union().

Next, we create the ZDD with the topmost element x and cofactors R0 and R1. This way we include into the resulting set of subsets all the subsets with (without) the topmost element if they appear with (without) the topmost element in one of the argument sets of subsets, A or B.

5.1 A case study of the CUDD source code

Now, consider the source code of the procedure cuddZddUnion(), which implements the recursive step of the set-union operator in the CUDD package (Fig. 7). The code is taken from file "cuddZddSetop.c" of the CUDD Release 2.3.1. Here it is reproduced with minor changes to improve its readability.

The procedure cuddZddUnion() is called with three arguments: the pointer to the decision diagram manager (zdd), and the ZDDs, P and Q. The local variables defined in the function store the levels of the topmost nodes in P and Q (p_top and q_top), the partial results (t and e), and the final result (res). Variables t, e, and res stand for R0, R1 and R in Fig. 6.

Function statLine(), implemented as a macro, is called with the pointer to the manager. It collects statistics about the number of recursive calls. It does not influence the functionality of cuddZddUnion().

The next three lines of the code implement the trivial cases. The macro DD_ZERO(zdd) returns constant-0 node of the DD manager representing the ZDD of the empty set.

Function cuddCacheLookup2Zdd() performs the cache lookup for a recursive procedure with two DD arguments. This function takes four arguments: the pointer to the manager, the pointer to the calling function (used as a signature to distinguish this cache entry from entries created by other functions possibly called with the same argument DDs), and two arguments, P and Q. This function returns NULL if there is no matching entry in the cache; otherwise, it returns the pointer to the ZDD of the result.

Each DD node F in the CUDD package is annotated with the variable number (F->index). The constant nodes have the variable number equal to CUDD_CONST_INDEX. Because dynamic variable reordering may be periodically performed in the manager, the variable number alone is not enough to find the position of the given DD node in the

variable order. To find the position, the levels of nodes should be determined.

The next two if-statements in the source code determine levels, p_top and q_top, of the topmost nodes in the argument DDs, P and Q. The level is the same as the variable index (CUDD_CONST_INDEX), if the DD node represents the constant function; otherwise it is determined using the mapping of variables into the corresponding levels, which is stored in the array zdd->permZ.

Next, three cases are considered:

- P is higher in the variable order than Q
- Q is higher in the variable order than P
- P is on the same level as Q

We discuss only the last one, the other two being similar.

Macros cuddT() and cuddE() return the "then" and "else" children (cofactors) of the given DD node. In ZDDs, these cofactors correspond to the sets of subsets, in which the topmost element is present (the "then" cofactor) or absent (the "else" cofactor). The cofactors are used in the recursive calls to cuddZddUnion(), which determine the two components of the result: the set of subsets with the topmost element (t) and without it (e).

All the DDs returned by the function calls are checked to determine if they are NULL. If the returned pointer to the node is not NULL, the node is referenced. If the returned pointer is NULL, it means that during the call either (1) the operating system has run out of memory when the CUDD package attempted to extend the node table, or (2) the dynamic variable reordering has been triggered. In both cases, the recursive traversal is interrupted and NULL is returned to the caller. Note also that the intermediate results referenced at some point during computation, are dereferenced by calling Cudd_RecursiveDerefZdd().

A few remarks should be made regarding the reference counting conventions employed in the CUDD package. A detailed treatment is given in CUDD User Manual [36].

Decision diagrams are stored in the DD manager as a shared directed acyclic graph of nodes. To mark the nodes that are in use, they are reference-counted. The reference counter of a DD node tells how many times this node participates in the DDs currently present in the manager. Therefore, each time a new DD node is created, its reference counter is incremented by the call to cuddRef().

```
DdNode *
cuddZddUnion(
  DdManager * zdd,
  DdNode *    P,
  DdNode *    Q)
{
 int p_top, q_top;
 DdNode *t, *e, *res;

 statLine(zdd);

 if ( P == DD_ZERO(zdd) )    return ( Q );
 if ( Q == DD_ZERO(zdd) )    return ( P );
 if ( P == Q )               return ( P );

 /* check cache */
 res = cuddCacheLookup2Zdd(zdd,cuddZddUnion,P,Q );
 if ( res != NULL )  return ( res );

 if ( cuddIsConstant( P ) )
   p_top = P->index;
 else
   p_top = zdd->permZ[P->index];

 if ( cuddIsConstant( Q ) )
   q_top = Q->index;
 else
   q_top = zdd->permZ[Q->index];

 if ( p_top < q_top )
 {
   e = cuddZddUnion( zdd, cuddE( P ), Q );
   if ( e == NULL )  return ( NULL );
   cuddRef( e );
   res = cuddZddGetNode(zdd,P->index,cuddT(P),e);
   if ( res == NULL )
   {
     Cudd_RecursiveDerefZdd( zdd, e );
     return ( NULL );
   }
   cuddDeref( e );
 }
 else if ( p_top > q_top )
 {
   e = cuddZddUnion( zdd, P, cuddE( Q ) );
   if ( e == NULL ) return ( NULL );
   cuddRef( e );
   res = cuddZddGetNode(zdd,Q->index,cuddT(Q),e);
   if ( res == NULL )
   {
     Cudd_RecursiveDerefZdd( zdd, e );
     return ( NULL );
   }
   cuddDeref( e );
 }
 else
 {
    t = cuddZddUnion( zdd, cuddT(P), cuddT(Q) );
    if ( t == NULL )   return ( NULL );
    cuddRef( t );
    e = cuddZddUnion( zdd, cuddE(P), cuddE(Q) );
    if ( e == NULL )
    {
      Cudd_RecursiveDerefZdd( zdd, t );
      return ( NULL );
    }
    cuddRef( e );
    res = cuddZddGetNode( zdd, P->index, t, e );
    if ( res == NULL )
    {
      Cudd_RecursiveDerefZdd( zdd, t );
      Cudd_RecursiveDerefZdd( zdd, e );
      return ( NULL );
    }
    cuddDeref( t );
    cuddDeref( e );
 }
 cuddCacheInsert2( zdd,cuddZddUnion,P,Q,res );
 return ( res );
} /* end of cuddZddUnion */
```

Figure 7. CUDD source code implementing the union of two sets represented as ZDD.

Similarly, each time an old DD node is deleted, its reference counter is decremented. If after decrementing, the reference counter becomes zero, the node is considered "dead" and the reference counters of the successor nodes are decremented in turn. This recursive decrementing of the

reference counters is performed by the call to Cudd_RecursiveDerefZdd().

Functions in the CUDD package (with very few exceptions mentioned in the user manual) return *non-referenced* DD nodes. It is the responsibility of a caller function to reference the returned node after checking that it is not NULL.

This principle is used in the source code in Fig. 7 several times. First of all, the returned nodes are referenced after each function call, except for the last call to cuddZddGetNode(). In this case, there is no need to reference the node to be returned (res) because the function should return the non-referenced node. Of course, it would not be an error to reference it by cuddRef(res) and then dereference it right before returning by cuddDeref(res). In the CUDD source code, these two steps are skipped for the sake of efficiency.

The last comment is about function cuddZddGetNode(). It returns the new ZDD node of the manager (zdd) with the given variable (P->index) and cofactors (t and e). Note the order of cofactors in the argument list: first the "then" cofactor, next the "else" cofactor. After the call to cuddZddGetNode(), the cofactor DDs (t and e) should be dereferenced because cuddZddGetNode() references them when it creates the new node (res). Because cofactor ZDDs (t and e) are now part of the result ZDD (res), there is no need for recursive dereferencing by the call to the function Cudd_RecursiveDerefZdd(). The cofactor ZDDs can be efficiently dereferenced using cuddDeref().

The commutativity of set-union allows us to improve the implementation by ordering the argument ZDDs (P and Q), which tends to increase hit-rate of the cache.

There are several ways to implement this improvement. One of them is based on the assumption that the ordering of arguments is given by the ordering of the pointers to the argument DdNode-structures. In this case, it is enough to replace each call to cuddZddUnion(zdd, A, B) by the lines

```
if ( (unsigned)A < (unsigned)B )
  cuddZddUnion( zdd, A, B );
else
  cuddZddUnion( zdd, B, A );
```

Our experiments have shown that this improvement leads to a 5% speedup in applications performing many calls to Cudd_zddUnion(), which is an exported procedure of the package internally calling procedure cuddZddUnion().

6 Manipulation of cube covers

This section gives an illustrative example of a traversal procedure dealing with cube covers represented by ZDDs. Fig. 8 shows the pseudo-code of the function implementing the product of two cube covers. As in the case of Union(), the cache lookups are omitted.

Consider the trivial cases. If any of the covers contains no cubes, the function represented by the cover is the constant-0 function and the product is 0, so the empty cover is returned. If any of the covers is the tautology cube, the product is equal to the other cover. Finally, if the covers are the same, the product is equal to any of them.

```
cover Product( cover A, cover B )
{
   if ( A = {}  || B = {} ) return {};
   if ( A = {{}} ) return B;
   if ( B = {{}} ) return A;
   if ( A = B    ) return A;

   var x  = TopVariable( A, B );

   cover A0, A1, A2, B0, B1, B2;
   ( A0, A1, A2 ) = DecomposeCover( A, x );
   ( B0, B1, B2 ) = DecomposeCover( B, x );

   cover TA0B0 = Product( A0, B0 );
   cover TA0B2 = Product( A0, B2 );
   cover TA1B1 = Product( A1, B1 );
   cover TA1B2 = Product( A1, B2 );
   cover TA2B0 = Product( A2, B0 );
   cover TA2B1 = Product( A2, B1 );
   cover TA2B2 = Product( A2, B2 );

   cover R0 = Union( Union(TA0B0, TA0B2), TA2B0 );
   cover R1 = Union( Union(TA1B1, TA1B2), TA2B1 );
   cover R2 = TA2B2;
   cover R  = ComposeCover( x, R0, R1, R2 );
   return R;
}
```

Figure 8. Pseudo-code of the product of two covers.

When the topmost variable x is determined, note that this is a primary input variable, not a ZDD variable. Each primary input variable is represented by two ZDD variables. This is exploited by the function TopVariable(), which is more complex than the function with the same name used in the procedure Union(). Next, both argument covers are cofactored into three subcovers containing cubes with the given variable in the negative polarity (A0 and B0), in the positive polarity (A1 and B1), and without the given variable (A2 and B2).

The main part of computation of cover-product (step (6)) is based on the following equality:

$$A * B = (\bar{x}\,A0 + xA1 + A2) * (\bar{x}\,B0 + xB1 + B2) =$$
$$= \bar{x}\,(A0*B2 + A2*B0 + A0*B0) +$$
$$+ x(A1*B2 + A2*B1 + A1*B1) + A2*B2.$$

This equality reduces the computation of the cover-product to seven recursive calls to cover-product of the cofactor DDs. Out of nine possible combinations (each of the three subcovers of A with each of the three subcovers of B), there is no need to consider only two combinations, A0*B1 and A1*B0, because the product of the negative and the positive literals reduces these sets to zero.

Finally, after five two-argument set-union operations which compute three subcovers, R0, R1, and R2, the resulting cover R is composed using the topmost variable.

Another implementation is possible that replaces two (out of seven) internal calls to Product() by two extra calls to

Union(). Because the implementation of Union() is simpler, this implementation is more efficient. The alternative implementation is based on the following equality:

$$A * B = (\bar{x} A0 + xA1 + A2) * (\bar{x} B0 + xB1 + B2) =$$
$$= \bar{x}(A0*B2 + B0*(A0+A2)) +$$
$$+ x(A1*B2 + B1*(A1+A2)) + A2*B2.$$

7 Mixed ZDD/BDD applications

In this section, we discuss two procedures, which play an important role in the SOP minimization. These are (1) the computation of all primes of the CSF and (2) the computation of an irredundant SOP of the ISF. In both cases, the arguments are represented as BDDs while the return values are represented by ZDDs. In general, decision-diagram-based procedures may take ZDDs and return BDDs or have other argument assignments.

7.1 Computation of the set of all primes

The recursive approach to the prime computation has been proposed in [30] and implemented using BDDs/ZDDs in [9]. The pseudo-code is shown in Fig. 9.

The trivial cases are simple. If the input function is constant-0, the set of primes is empty. If the input function is constant-1, the prime set composed of the tautology cube is returned. Note that the set of subsets that includes only the empty subset, {{}}, represents the cube with no literals, that is, the tautology cube.

If it is not a trivial case, the topmost variable in the BDD of F is determined and the function is decomposed w.r.t. this variable. Next, the problem is solved in three steps.

First, the set of primes (P2) belonging to the intersection of cofactors is computed. These primes do not have the topmost variable as the positive or the negative literal.

Second, the set of all primes of the negative cofactor of the function (P0) is computed. These primes will have the topmost variable in the negative polarity. Before the topmost literal is added (when composing the result at the end of the procedure), some of them may be identical to the primes in P2. After adding the literal corresponding to the topmost variable, some of the cubes in P2 will contain the corresponding cubes in P0, because cubes in P0 get the negative literal while the cubes in P2 do not get a literal associated with this variable. Because the contained cubes are, by definition, not primes, they should be removed. This is done by the set-difference operator applied to P0 and P2.

Similarly, in the third step, we compute the set of all primes with the positive literal (P1).

Finally, the resulting set of primes is composed from the three subsets, P0, P1, and P2 and returned.

Again, the prime computation can be improved by detecting situations when the given function is unate in its topmost variable. In this case, there is no need to make one out of the three recursive calls to Primes() because the primes of the unate function do not have the given variable in any polarity or have it in only one polarity, either negative or positive, depending on the type of unateness.

```
cover Primes( func F )
{
   if ( F = 0 ) return {};
   if ( F = 1 ) return {{}};

   var x = TopVariable( F );

   func F0, F1;
   (F0, F1) = DecomposeBdd( F, x );

   cover P2 = Primes( F0 & F1 );
   cover P0 = Primes( F0 );
   cover P1 = Primes( F1 );
   P0 = P0 - P2;
   P1 = P1 - P2;

   cover P = ComposeCover( x, P0, P1, P2 );
   return P;
}
```

Figure 9. Pseudo-code of the prime set computation.

The mixed BDD/ZDD implementation of Primes() is very efficient because it takes a fractions of a second to compute the primes of any function from the Espresso PLA benchmark set, including the so-called hard benchmarks, on a modern computer. To this end, multi-output functions are converted into single-output ones, as shown in [5].

7.2 Computation of an irredundant SOP

The algorithm for recursive computation of an ISOP has been proposed in [31]. It has been implemented using decision diagrams in [20][8]. The pseudo-code is given in Fig. 10. Symbols "+", "&", and "-" in the pseudo-code stand for the Boolean operations OR, AND, and SHARP where SHARP(a, b) = a & $\bar{b}$.

The procedure IrrSOP() to compute the irredundant sum-of-products is called with two arguments representing an ISF. The first argument F is the on-set, while the second argument FD is the sum of the on-set and the don't-care-set. If the ISF is represented by the on-set and off-set, FD is computed by complementing the off-set.

The following are the trivial cases. If the on-set F is empty, the function is a constant-0 function, and the empty cover is returned. If the union of the on-set and dc-set, FD, covers the whole Boolean space, the tautology cube (the set of subsets composed of the empty subset) is returned.

Next, the topmost variable x and the cofactors of F and FD w.r.t. the variable x are derived. To find the decomposition of BDDs, the ordinary BDD cofactoring w.r.t. the topmost variable is used. Note that if one of the functions, F or FD, does not depend on the topmost variable, its cofactors w.r.t. the topmost variable are equal to the function itself.

The ISOP is computed in three steps. The first step finds the ISOP cover of that part of the on-set F, which cannot be covered by the cubes without the given variable. To achieve

this, the ISOP of the part of F0 that is outside FD1 is computed. This part can be covered only by the cubes that contain the topmost variable in the negative polarity. These cubes are assigned to R0.

Similarly, in the second step, the ISOP of the part of the on-set covered only by the cubes with the topmost variable in the positive polarity is computed. These cubes are assigned to R1.

In the third and final step, the area in the intersection of FD0 and FD1, which (1) belongs to F0 or F1 and (2) is not covered by the cubes in R0 and R1 is computed. The pseudo-code employs the function Bdd(), which takes the cover as a ZDD and returns the BDD of the area of the Boolean space corresponding to the cover.

The CUDD package provides the implementation of the ISOP procedure, in which two values are returned in each call: the ZDD of the ISOP cover and the BDD of the area covered by the cover. In this way, there is no need for the call to Bdd(). The penalty for this solution is the necessity to cache two values, the ZDD and the BDD, and the repeated recursive call if at least one of the values is lost in the cache (the loss of values in the cache happens when two different computed results hash into the same cache entry resulting in the loss of the earlier result).

The EXTRA library gives an alternative implementation of this procedure, in which only one value is cached and returned. This implementation is based on a specialized operator, which takes two arguments, the BDD of the area and the ZDD of the cover and returns the BDD of the area that is not covered by the cover. This operator can implement expressions of the type A – Bdd(B) in one traversal, while in the pseudo-code of Fig. 10, expressions of this kind are implemented by two traversals: Bdd() and Boolean SHARP. Experimentation shows that this implementation of an ISOP cover is more efficient than the one proposed in the CUDD package.

```
cover IrrSOP( func F, func FD )
{
   if ( F  = 0 )  return {};
   if ( FD = 1 )  return {{}};

   var x = TopVariable( F, FD );

   func F0, F1, FD0, FD1;
   (F0, F1) = DecomposeBdd( F, x );
   (FD0,FD1)= DecomposeBdd( FD, x );

   func  G0 = F0 - FD1;
   cover R0 = IrrSOP( G0, FD0 );
   func  G1 = F1 - FD0;
   cover R1 = IrrSOP( G1, FD1 );

   func  H  = (F0 - Bdd(R0)) + (F1 - Bdd(R1));
   func  HD = FD0 & FD1;
   cover R2 = IrrSOP( H, HD );

   cover R  = ComposeCover( x, R0, R1, R2 );
   return R;
}
```

Figure 10. Pseudo-code of Irredundant SOP computation.

The ISOP computed by this algorithm has remarkable properties [29]. The actual cover depends on the ordering of variables in the DD manager. In many cases, the number of cubes in the resulting ISOP is close to that in the exact minimum cover [20][8].

One main advantage of the ZDD-based ISOP is the speed of computing it. For large benchmarks, good-quality covers containing thousands of cubes can be derived in a fraction of a second. Using ISOP instead of other heuristic algorithms for the SOP computation may lead to a substantial speedup in some applications [18].

Our experiments have also shown that the resulting ISOPs often lead to factored forms with fewer literals than those computed from SOPs minimized using other methods. This may have to do with the fact that the ZDD-based computation enforces a fixed variable order while processing different cofactors of the original function.

8 A list of published ZDD applications

ZDDs have been introduced in 1993 by S. Minato [21] and studied in [26][28]. Several ZDD packages have been implemented [15][16][24][36]. Here is an incomplete list of problems arising in computer science and engineering that have been successfully solved using ZDDs:

- To represent sets in various applications [23][34].
- To represent cubes and essential primes in two-level SOP minimization [9] and factorization [22][27][35].
- To solve the unate covering problem arising in multi-layer planar routing [11].
- To find dichotomy-based constraint encoding [12][14].
- To solve graph optimization problems [13].
- To represent and manipulate regular expressions under length constraint [17].
- To represent and manipulate polynomials with integer coefficients [25].
- In exclusive SOP minimization [32][33].
- In symbolic traversal of FSMs and Petri nets [38][40].
- In Davis-Putman resolution procedure [6].
- In pass-transistor logic synthesis [3].
- In finding all disjoint-support decompositions of completely specified logic functions [29].
- In unate decomposition of boolean functions [18].

9 Conclusions

This tutorial introduces the reader to the fascinating world of Zero-Suppressed Binary Decision Diagrams and shows by way of example how to use them for solving a number of computationally hard problems.

10 Acknowledgements

The author thanks Gianpiero Cabodi, Olivier Coudert, Timothy Kam, Jørn Lind-Nielsen, Shin-ichi Minato, In-Ho Moon, Thomas Shiple, and Fabio Somenzi whose books, papers, source code, and personal advice helped him learn implementation and applications of decision diagrams.

11 Appendix A:

ZDD procedures in CUDD

This section lists the ZDD procedures implemented in the package, as described in the CUDD user manual.

ZDD construction

Cudd_ReadZero: Returns a contant-0 ZDD node representing the empty set.

Cudd_ReadOne: Returns a contant-1 ZDD node representing the set composed of the empty subset only.

Cudd_ReadZddOne: Returns the ZDD of the constant-1 function assuming that this function depends on the given number of variables.

Cudd_zddIthVar: Returns the ZDD of the function equal to the elementary variable if this variable exists in the DD manager, or creates a new ZDD variable.

Cudd_zddIte: Computes the ITE of three functions represented by ZDDs.

Cudd_zddChange: Substitutes a variable by its complement in a ZDD.

Porting

Cudd_zddPortFromBdd: Converts a BDD into a ZDD.

Cudd_zddPortToBdd: Converts a ZDD into a BDD.

Cudd_zddVarsFromBddVars: Creates one or more ZDD variables for each BDD variable.

Cofactoring

Cudd_zddSubset0 (*Cudd_zddSubset1*): Computes the negative (positive) cofactor of a ZDD w.r.t. a variable.

Set operators

Cudd_zddUnion: The union of two sets.

Cudd_zddIntersect: The intersection of two sets.

Cudd_zddDiff: The difference of two sets.

Cudd_zddDiffConst: Inclusion test for sets (P implies Q).

Cudd_zddUnateProduct: The product of two unate covers. Unate covers use one ZDD variable for each BDD variable.

Cudd_zddDivide: The quotient of two unate covers.

Cover manipulation

Cudd_zddProduct: The product of two binate covers. The binate covers use two ZDD variables for each BDD variable.

Cudd_zddWeakDiv: The quotient of two binate covers.

Cudd_zddComplement: The complement of a cover.

Cudd_zddIsop: An irredundant sum of products (ISOP) in ZDD form BDDs for the on-set and the on+dc-set.

Cudd_BddFromZddCover: Returns the BDD of the function representated by a cover.

Counting functions

Cudd_zddDagSize: Counts nodes in a ZDD.

Cudd_zddCount: Returns the number of paths in a ZDD.

Cudd_zddCountMinterm (*Cudd_zddCountDouble*): Count the number of minterms of a ZDD.

Reordering

Cudd_zddReduceHeap: The main dynamic reordering routine for ZDDs.

Cudd_zddShuffleHeap: Reorders ZDD variables according to given permutation.

Cudd_zddSymmProfile: Prints statistics on symmetric ZDD variables.

Realignment of Variables

Cudd_zddRealignEnable: Enables realignment of the ZDD variable order to the BDD variable order after the BDDs and ADDs have been reordered.

Cudd_zddRealignDisable: Disables realignment of ZDD order to BDD order.

Cudd_zddRealignmentEnabled: Returns 1 if the realignment of ZDD order to BDD order is enabled.

Printing and visualization

Cudd_zddDumpDot: Writes a file representing the argument ZDDs in a format suitable for the graph drawing program DOT [1].

Cudd_zddPrintCover: Prints an SOP representation of a ZDD.

Cudd_zddPrintDebug: Prints a DD and its statistics to the standard output.

Cudd_zddPrintSubtable: Prints the ZDD table for debugging purposes.

12 Appendix B:

ZDD Procedures in the EXTRA library

The ZDD operators in the library are grouped according to their purpose. Unless stated otherwise, all the operators are implemented in the bottom-up reorder-independent fashion, meaning that an operator restarts when the dynamic variable reordering has taken place.

ZDD construction

Extra_zddCombination: Creates a ZDD of one combination.

Extra_zddUniverse: Builds a ZDD for all possible combinations of variables from the given set.

Extra_zddTuples: Builds a ZDD representing the set of all tuples of the given cardinality composed of variables from the given set.

Extra_zddRandomSet: Builds the random set of k combinations, each of which may contain up to n elements with the average density d.

Extra_zddConvertBddCubeIntoZddCube: Takes a BDD of the variable product and returns a ZDD cube composed of the same variables.

Set operators

Extra_zddMaximal: Computes the maximal of the set of subsets defined as follows:

$$\max(S) = \{ s \in S \mid \forall s' \in S, s \subseteq s' \Rightarrow s = s' \}.$$

*Extra_zddMinimal***:** Computes the minimal of the set of subsets defined as follows:

$$\min(S) = \{ s \in S \mid \forall s' \in S, s \supseteq s' \Rightarrow s = s' \}.$$

Extra_zddMaxUnion (*Extra_zddMinUnion*): Computes the maximal (minimal) of the union of sets X and Y in one bottom-up traversal.

Extra_zddDotProduct: Computes the set of subsets created by taking pair-wise unions of subsets from X and Y:

$$\text{DotProduct}(X, Y) = \{ x \cup y \mid x \in X, y \in Y \}.$$

Extra_zddMaxDotProduct: Computes the maximal of the set of subsets creates by taking pair-wise unions of subsets from X and Y:

$$\text{MaxDotProduct}(X, Y) = \max(\{ x \cup y \mid x \in X, y \in Y \}).$$

Extra_zddSubSet: Computes the set of subsets in X that are contained in at least one subset of Y:

$$\text{SubSet}(X, Y) = \{ x \in X \mid \exists y \in Y, x \subseteq y \}.$$

Extra_zddSupSet: Computes the set of subsets in X that contain at least one subset of Y:

$$\text{SupSet}(X, Y) = \{ x \in X \mid \exists y \in Y, x \supseteq y \}.$$

Extra_zddNotSubSet: Computes the set of subsets in X that are contained in at least one subset of Y:

$$\text{NotSubSet}(X, Y) = \{ x \in X \mid \forall y \in Y, x \not\subseteq y \}.$$

Extra_zddNotSupSet: Computes the set of subsets in X that are contained in at least one subset of Y:

$$\text{NotSupSet}(X, Y) = \{ x \in X \mid \forall y \in Y, x \not\supseteq y \}.$$

Extra_zddMaxNotSupSet: Computes the maximal of the set of subsets in X that do not contain any subset of Y in one bottom-up traversal.

Extra_zddEmptyBelongs: Returns 1 if the given ZDD contains the empty combination, and 0 otherwise.

Extra_zddExistAbstract: Removes from a ZDD the occurrences of variables belonging to the given set.

Extra_zddChangeVars: Changes the values of the variables belonging to the given set in all combination of the ZDD.

Extra_zddCommonCube: Computes all the variables that appear in all combinations of a ZDD.

Extra_zddCofactor0 (*Extra_zddCofactor1*): Computes all combinations that contain (do not contain) the variables belonging to the given set.

Extra_zddMaximum (*Extra_zddMinimum*): Returns a ZDD representing all combinations of the set S containing the maximum (minimum) number of elements.

Extra_zddSinglesToComb: Takes a ZDD of singleton combinations (combinations including exactly one element) and returns a ZDD containing one combination composed of all elements.

Cover manipulation

Extra_zddPrimes: Given the BDD of the function F, computes a ZDD representing the set of all prime implicants of F.

Extra_zddProductAlt: An alternative implementation of the product of two covers.

Extra_zddPrimeProduct: Computes the product of two covers from which the contained cubes are removed "on the fly".

Extra_zddResolve: Computes all resolvents of the set of clauses S w.r.t. the set of variables Vars.

Extra_zddCompatible: Computes all the cubes from the given set that overlap with the given cube.

Extra_zddDisjointCover: Computes the ZDD of the cover represented by disjoint variable paths in the given BDD.

Extra_zddSelectOneCube: Returns a randomly selected cube from the given cover.

Extra_zddCheckUnateness: Returns 1 if the given cover is (positive or negative) unate in all its variables.

Extra_zddUnionExor: Computes the union of two covers, while removing the cubes contained in both covers.

Extra_zddSupercubes: Given two sets of cubes, computes the set of their pair-wise supercubes.

Extra_zddSelectDist1Cubes: Selects cubes from the given set that have at least one distance-1 cube in another set.

Extra_zddEssential: Computes the essential cubes.

extraDecomposeCover: Find the cofactors of the cover w.r.t. the top variable, without creating new DD nodes.

extraComposeCover: Composes the cover from the three subcovers using the given variable.

Cover/area manipulation

Extra_zddCoveredByArea: Returns the cubes from the given cover, completely contained in the given area of the Boolean space.

Extra_zddOverlappingWithArea: Returns the cubes from the given cover, overlapping with the given area do the Boolean space.

Extra_zddNotCoveredByCover: Returns the cubes belonging to the given cover, that are not completely covered by another cover.

Extra_zddNotContainedCubesOverArea: Computes the cubes that do not overlap with the cubes from the other set over the given area.

Extra_zddConvertToBdd: An alternative implementation of the procedure *Cudd_MakeBddFromZddCover* from the CUDD package.

Extra_zddConvertToBddAndAdd: Computes the Boolean OR of the given area and the area covered by the cover in one traversal.

Extra_zddConvertEsopToBdd: Computes the BDD of the exclusive sum-of-products represented by a ZDD.

Extra_zddSingleCoveredArea: Computes the area of the Boolean space covered by only one cube from the cover.

Extra_zddGetMostCoveredArea: Computes the area covered by the maximum number of cubes in a ZDD.

Irredundant SOP computation

Extra_zddIsopCover: A wrapper around *Extra_zddIsop* from the CUDD package. This function returns only the ZDD of the cover and does not return its BDD.

Extra_zddIsopCoverAlt: An alternative implementation of the ISOP computation. This function may be more efficient than *Extra_zddIsop* and *Extra_zddIsopCover.*

Extra_zddIsopCoverRandom: Computes an ISOP cover assuming the random permutation of variables.

Extra_zddIsopCoverAllVars: Tries all possible permutations of variables in every subcover and returns the ISOP with the smallest number of cubes (potentially very slow for more than 10 variables).

Extra_zddIsopCoverUnateVars: Detects unate variables and performs decomposition w.r.t. these variables first (this function is slower but gives smaller covers compared to *Extra_zddIsop* and *Extra_zddIsopCover.*)

Graph input/output

Extra_zddGraphRead: Reads the file with the non-directed graph in DIMACS formats and creates its representation as a ZDD.

Extra_zddGraphWrite: Writes the non-directed graph represented as a ZDD into a file in DIMACS ASCII format.

Extra_zddGraphDumpDot: Writes a file representing the graph in a format suitable for the graph-drawing program DOT [1].

Graph operators

Extra_zddCliques: Finds the set of all cliques of the graph represented as a ZDD.

Extra_zddMaxCliques: Finds the set of all maximal cliques of the graph represented as a ZDD in one bottom-up traversal.

Extra_zddIncremCliques: Given a ZDD of the graph and a ZDD of all cliques of size k, computes the set of all cliques of size k+1. (Note that the graph representation *is* the set of all cliques of the size two.)

Extra_zddGraphColoring: Given a ZDD of the graph, finds a heuristic coloring of the graph (not finished)

Extra_zddRandomGraph: Generates a ZDD representing a random graph with n nodes and density d (not finished)

Set covering

Extra_zddSolveUCP: Solves the set-covering problem specified as follows: Each element is encoded using a separate ZDD variable. The only argument S is the set of subsets that covers all elements. The set of elements is determined as the support of S (not finished).

Extra_zddSolveCC: Solves the cyclic core specified by the pair of ZDDs representing the set of rows and the set of columns, using a fast greedy heuristic method.

Reordering

Extra_zddPermute: Given a ZDD and the permutation of variables, creates a ZDD with permuted variables.

Counting functions

Extra_zddLitCount: Counts how many times each element occurs in the combinations of the set.

Other functions

Extra_zddSupport: Returns a ZDD representing a set of variables, on which the given DD depends.

Extra_zddVerifyCover: Takes the cover and the function interval represented by two BDDs. Returns 1 if the cover belongs to the interval.

13 References

[1] AT&T Labs Research. Graphviz. http://www.research.att.com/sw/tools/graphviz/.

[2] R. I. Bahar, E. A. Frohm, Ch. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, F. Somenzi. Algebraic Decision Diagrams and their Applications. *Proc. of ICCAD '93*. pp. 188-191.

[3] V. Bertacco, S. Minato, R. Verplaetse, L. Benini, G. De Micheli. Decision Diagrams and Pass Transistor Logic Synthesis. *Technical Report CSL-TR-97-748.* Computer System Laboratory. Stanford University. December 1997.

[4] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comp.*, Vol. C-35, No. 8 (August, 1986), pp. 677-691.

[5] E. Cerny, M.A. Marin. An Approach to Unified Methodology of Combinational Switching Circuits. *IEEE Trans. on Computers*, C-26, 8 (Aug.), pp. 745-756.

[6] P. Chatalic, L. Simon. ZRes: The Old Davis-Putnam Procedure Meets ZBDDs. *Proc. of CADE-17: Conference on Automated Deduction 2000*, pp. 449-454.

[7] O. Coudert, C. Berthet, J. C. Madre, Verification of Sequential Machines using Boolean Functional Vectors, in *Formal VLSI Correctness Verification*, L. J. M. Claesen Editor, North-Holland, pp. 179-196, Nov. 1989.

[8] O. Coudert, J. C. Madre, H. Fraisse, H. Touati. Implicit Prime Cover Computation: An Overview. *Proc. of SASIMI '93*, Nara, Japan.

[9] O. Coudert. Two-Level Logic Minimization: An Overview. *Integration*. Vol. 17, No. 2, pp. 97-140, October 1994.

[10] O. Coudert. Doing Two-Level Logic Minimization 100 Times Faster. *Proc. of Symposium on Discrete Algorithms (SODA)*, San Francisco CA, January 1995.

[11] O. Coudert, C.-J. R. Shi. Exact Multi-Layer Topological Planar Routing. *Proc. of IEEE Custom Integrated Circuit Conference '96*, pp. 179-182.

[12] O. Coudert, C.-J. R. Shi. Exact Dichotomy-based Constraint Encoding. *Proc. of ICCD '96*, pp. 426-431.

[13] O. Coudert. Solving Graph Optimization Problems with ZBDDs. *Proc. of ED&T '97*, pp. 224-228.

[14] O. Coudert. A New Paradigm of Dichotomy-based Constraint Encoding. *Proc. of DATE '98*, pp. 830-834.

[15] O. Coudert. TiGeR Package.

[16] S. Höreth. TUD Package.

[17] S. Ishihara, S. Minato. Manipulation of Regular Expressions Under Length Constraints Using Zero-Suppressed BDDs. *Proc. of ASP-DAC '95*, pp. 391-396.

[18] J. Jacob, A. Mishchenko. Unate Decomposition of Boolean Functions. *Proc. of IWLS '01*, Lake Tahoe, California.

[19] T. Kam, T. Villa, R. Brayton, A. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization.* Kluwer Academic Publishers, 1997.

[20] S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. *Proc. of SASIMI'92 (Synthesis and Simulation Meeting and International Interchange)*, Kobe, Japan, pp. 64-73.

[21] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. *Proc. of DAC '93*, pp. 272-277.

[22] S. Minato. Fast Weak-Division Method for Implicit Cube Representation, *Proc. SASIMI '93*, pp. 423-432, Oct.1993.

[23] S. Minato. Calculation of Unate Cube Set Algebra Using Zero-Suppressed BDDs. *Proc. of DAC '94*, p. 420-424.

[24] S. Minato. ZDD package referenced in [23].

[25] S. Minato. Implicit Manipulation of Polynomials Using Zero-Suppressed BDDs. *Proc. of ED&TC '95*, pp. 449-454.

[26] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD.* Kluwer 1995.

[27] S. Minato. Fast Factorization Method for Implicit Cube Cover Representation. *IEEE Trans. CAD*, Vol. 15, No 4, April 1996, pp. 377-384.

[28] S. Minato. Graph-Based Representations of Discrete Functions", In T. Sasao, ed., *Representation of Discrete Functions*, Ch. 1, pp. 1-27, Kluwer 1996.

[29] S. Minato, G. DeMicheli. Finding All Simple Disjunctive Decompositions Using Irredundant Sum-of-Products Forms. *Proc. of ICCAD '98*, pp. 111-117.

[30] A. Mishchenko. EXTRA Library of DD procedures. http://www.eecs.berkeley.edu/~alanmi/research/extra

[31] E. Morreale. Recursive Operators for Prime Implicant and Irredundant Normal Form Determination. *IEEE Trans. Comp.*, C-19(6), 1970, pp. 504-509.

[32] H. Ochi. An Exact Minimization of AND-EXOR Expressions Using Encoded MRCF. *IEICE Trans. Fundamentals*, Vol. E79-A, No. 12, Dec. 1996, pp. 2131-2133.

[33] H. Ochi. A Zero-Suppressed BDD package with Pruning and Its Applications to GRM Minimization. *IEICE Trans. Fundamentals*, Vol. E79-A, No. 12, Dec. 1996, pp. 2134-2139.

[34] H. G. Okuno, S. Minato, H. Isozaki. On the Properties of Combination Set Operations. *Information Processing Letters*, 66 (1998), pp. 195-199.

[35] H. Sawada, Sh. Yamashita, A. Nagoya. An Efficient Method for Generating Kernels on Implicit Cube Set Representations. *Proc. of IWLS' 99*, pp. 260-263.

[36] F. Somenzi. *CUDD Package.* http://vlsi.Colorado.EDU/~fabio/CUDD/cuddIntro.html

[37] G. Swamy, R. Brayton, and P. McGeer. A Fully Implicit Quine-McCluskey Procedure Using BDD's. *Tech. Report No. UCB/ERL M92/127*, 1992.

[38] M. Tomisaka, T. Yoneda. Partial Order Reduction in Symbolic State Space Traversal Using ZBDDs. *IEICE Trans. Fundamentals*, Vol. E00-D, No. 1, Jan.1999, pp. 1-8.

[39] T.Villa, T.Kam, R.Brayton, A.Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Logic Optimization*. Kluwer Academic Publishers, 1997. Chapter 10. "Implicit Formulation of Unate Covering". pp. 301-321.

[40] T. Yoneda, H. Hatori, A. Takahara, S. Minato. BDDs vs. Zero-Suppressed BDDs: for CTL Symbolic Model Checking of Petri Nets. *Proc. FMCAD '96*. LNCS #1166. pp.435-449.