Implementation and Visualization of a BDD Package in JAVA

Rolf Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen
drechsle@informatik.uni-bremen.de

Jochen Römmler

Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg im Breisgau
roemmler@informatik.uni-freiburg.de

Abstract

Ordered Binary Decision Diagrams (BDDs) are a data structure for representation and manipulation of Boolean functions often applied in VLSI CAD. Algorithms have to be developed to allow efficient BDD manipulation and minimization.

In this paper we describe the implementation of a BDD package in JAVA. An interactive interface allows to display the construction and minimization process and by this gives further insight into the data structure. New minimization algorithms can easily be evaluated and their performance can be analyzed.

1 Introduction

Decision Diagrams (DDs) are often used in VLSI CAD systems for efficient representation and manipulation of Boolean functions. The most popular data structure are *ordered Binary Decision Diagrams* (BDDs) [3, 4]. But, as well known BDDs are very sensitive to the variable ordering, i.e. the size of a BDD (measured in the number of nodes) may vary from linear to exponential. Finding the optimal variable ordering is an NP-hard problem [1] and the best known algorithm has runtime $O(n^2 \cdot 3^n)$ [5], where n denotes the number of variables.

This is the reason why many authors presented heuristics for finding good variable orderings from circuit descriptions in the last few years (see e.g. [6]). The most promising methods for BDD minimization are based on *Dynamic Variable Ordering* (DVO) [7], i.e. improving graph size using exchanges of neighboring variables. The best results measured in the number of nodes of the resulting BDD were obtained using *sifting* [12, 11]. For getting a better insight into the minimization process, it is very helpful to have some visualization of the minimization process. This is the reason, why several authors of BDD packages provide (graphic) outputs of the BDDs [2, 10, 13, 8]. But none of these packages allow any user interaction during visualization.

In this paper we describe the implementation of a BDD package in JAVA. JAVA has become very popular: it offers many predefined graphics routines and it has an internal garbage collector, which simplifies programming significantly, because former allocated memory does not need to be freed explicitly. Our BDD package has an interactive interface that allows to display the construction and minimization process and by this gives further insight in the data structure. Formulas can be entered and immediately translated into a corresponding BDD. Users can easily change the global variable ordering in a given BDD simply by selecting a certain variable with

```
ite(F,G,H) {
    if (terminal case) return result;
    if (computed-table entry (F,G,H) exists) return result;

    let x_i be the top variable of {F,G,H};

    THEN= ite(F_{x_i}, G_{x_i}, H_{x_i}) ;
    ELSE = ite(F_{x̄_i}, G_{x̄_i}, H_{x̄_i}) ;

    if (THEN == ELSE) return THEN;

    // Find or create a new node with variable v and sons THEN and ELSE
    R = Find_or_add_unique_table(x_i,THEN,ELSE);

    // Store computation and result in computed table
    Insert_computed_table({F,G,H},R);

    return R;
}
```

Figure 1: ITE-algorithm

the mouse and exchange its position with adjacent variables. The effects on the BDD can be seen immediately. This allows the user to manually optimize the BDD if the automatic minimization algorithm returns unsatisfying results. The viewable area can also be changed using the mouse in order to zoom into larger structures. New minimization algorithms can easily be evaluated and their performance can be analyzed.

The next section will shortly describe the main aspects of BDDs and the impact of the variable ordering followed by a chapter on implementation of the package in JAVA. Finally the key features of the *Graphical User Interface* (GUI) are briefly discussed.

2 Preliminaries

2.1 Binary Decision Diagrams

As well-known, each Boolean function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ can be represented by a *Binary Decision Diagram* (BDD) [3], i.e. a directed acyclic graph where a Shannon decomposition is carried out in each node. In the following, only reduced, ordered BDDs are considered and for briefness these graphs are called BDDs. (For more details see [4].)

We briefly consider the typical synthesis operation on BDDs: The synthesis of two BDDs is carried out by performing a recursive call on subgraphs. A sketch of the recursive *If-Then-Else* (ITE) algorithm from [2] is given in Figure 1. Based on ITE all binary synthesis operations, like AND and OR, can be carried out (see Table 1).

BDDs are a canonical form, i.e. two different BDDs represent two different Boolean functions and vice versa. To maintain this form, two things are important: isomorphism and Shannon-reduction.

In a canonical representation we don't want isomorphic subgraphs. To avoid this problem, a hash-based table called `unique-table` is introduced in ITE as part of a canonical representation of the resulting function. Every time ITE is called a lookup in this hash-table must be made.

Table 1: Binary synthesis operations performed by ITE

Table	Name	Expression	Equivalent form
0000	0	0	0
0001	$AND(F,G)$	$F \cdot G$	$ITE(F,G,0)$
0010	$F > G$	$F \cdot \bar{G}$	$ITE(F,\bar{G},0)$
0011	F	F	F
0100	$F < G$	$\bar{F} \cdot G$	$ITE(F,0,G)$
0101	G	G	G
0110	$XOR(F,G)$	$F \oplus G$	$ITE(F,\bar{G},G)$
0111	$OR(F,G)$	$F + G$	$ITE(F,1,G)$
1000	$NOR(F,G)$	$\overline{F+G}$	$ITE(F,0,\bar{G})$
1001	$XNOR(F,G)$	$\overline{F \oplus G}$	$ITE(F,G,\bar{G})$
1010	$NOT(F,G)$	$\bar{G}$	$ITE(G,0,1)$
1011	$F \geq G$	$F + \bar{G}$	$ITE(F,1,\bar{G})$
1100	$NOT(F)$	$\bar{F}$	$ITE(F,0,1)$
1101	$F \leq G$	$\bar{F} + G$	$ITE(F,G,1)$
1110	$NAND(F,G)$	$\overline{F \cdot G}$	$ITE(F,\bar{G},1)$
1111	1	1	1

If a corresponding node could be found, the hash-table entry is used rather than creating a new node. Otherwise the new node is added to the `unique-table`.

The second issue is Shannon-reduction: We omit a node, if its THEN and ELSE branches are referring to the same node. This reduction is done automatically when building a BDD using ITE algorithm (it's a simple `if-then-else` statement). For more details see [2].

These two aspects guarantee a strong canonical form of the represented function at all times.

2.2 Dynamic Minimization

The basic operation of dynamic variable ordering is the exchange of adjacent variables [7, 12]. The exchange is performed very quickly since only edges have to be redirected within these levels. Thus, the size is optimized without a complete reconstruction of the BDD. Only local transformations for the two levels are performed. This is due to the observation that BDDs are a canonical form. The exchange of two variables do not change the sub-BDDs of other levels.

The sifting algorithm [12] successively considers all variables of a given BDD. When a variable is chosen, the goal is to find the best position of the variable, assuming that the relative order of all other variables remains the same. In a first step, the order in which the variables are considered is determined. This is done by sorting the levels according to their size with largest level first. To find the best position, the variable is moved across the whole BDD. In [12], this is done in three steps:

1. The variable is exchanged with its successor variable until it is the last variable in the ordering.

2. The variable is exchanged with its predecessor until it is the topmost variable.

3. The variable is moved back to the closest position which has led to the minimal size of the BDD.

3 BDD Implementation in JAVA

In this section we describe the main components of our implementation. The core of the BDD package is the underlying graph structure.

3.1 Storing Nodes

For each node the two outgoing edges and the label at the node have to be stored. Furthermore, we support additional information to allow simple handling of the graphical output, like e.g. the x- and y-coordinates when drawing the graph. For the constructor of the node class we get:

```
public node(int vi) {
 G = null;
 H = null;
 v = vi;
 last_x = 0;
 last_y = 0;
 version = 0;
 selected=false;
 counted=false;
}
```

In the same way a node can be created by assigning the successors `g` and `h` already during the call, i.e. `public node(int vi, node g, node h)`. The class also provides a set of operators for efficiently traversing the BDD and determining informations, like the number of nodes. Based on this simple setup the recursive algorithm ITE described in the previous section can be directly applied.

3.2 Synthesis of Nodes

The `ITE` algorithm operates on three nodes F, G and H given as parameters (see Figure 1) and merges them together returning a result node R. The function represented by R has to be in a strong canonical form. Thus, additional checks have to be added to maintain this form. The fact that `ITE` can be used to perform all binary operations simplifies the implementation of all synthesis operations; the following example shows the synthesis of nodes using the basic Boolean functions `AND`, `OR` and `NOT`:

```
public node AND(node a, node b) {
  return ITE(a, b, ZERO);
}

public node OR(node a, node b) {
  return ITE(a, ONE, b);
}

public node NOT(node a) {
  return ITE(a, ZERO, ONE);
}
```

These class methods are automatically invoked during parsing of the given user formula.

```
public static void structure_analysis(node p, int height, int x, int y) {
      int deltax = scaleX*height; // spread-factor for subgraphs (sons)
      boolean sonG_finished = false; // recursively operate on sons?
      boolean sonH_finished = false;
      if (NOT assigned_yet(p)) { assign(p,x,y); }
      if (NOT p.is_leaf()) {
            if assigned(p.G) { // testing G son
               if (need_to_adjust_level(p.G,y,scaleY)) {
                  adjust_level(p.G); // re-analyse subgraph
                  structure_analysis(p.G,height-1,p.G.last_x,p.G.last_y);
               }
               sonG_finished = true;
            }
            if assigned(p.H) { // testing H son
               if (need_to_adjust_level(p.H,y,scaleY)) {
                  adjust_level(p.H); // re-analyse subgraph
                  structure_analysis(p.H,height-1,p.H.last_x,p.H.last_y);
               }
               sonH_finished = true;
            }
            if NOT sonG_finished
               structure_analysis(p.G,height-1,x-deltax,y+scaleY);
            if NOT sonH_finished
               structure_analysis(p.H,height-1,x+deltax,y+scaleY);
      } // finish if node is leaf
}
```

Figure 2: Analysis of the BDD

3.3 Parsing the Formula

Parsing of the user formula is implemented using the parser generator `CUP` ("Constructor of Useful Parsers") by Hudson [9]. Using a simple parser-specification file `CUP` generates JAVA source-code which can be adapted to the given context. In general, these specification files contain the grammar for the parser.

3.4 Drawing the Graph

3.4.1 Structure Analysis

Before we can draw the BDD data structure in a panel for user interaction, a complete analysis must be performed. Starting from the graph's root we descent recursively in the BDD to determine the correct x- and y-coordinates for each node. Figure 2 shows a brief simplified example how analysis takes place.

This analysis is only possible because each node is unique. Structure analysis must be performed every time the BDD is redrawn and the corresponding method `structure_analysis` is invoked automatically. A difficult issue is that nodes can be referenced multiple times by different parents. This makes it difficult to keep all nodes of one kind on the same level. An example of the structure analysis is given in Figure 3.

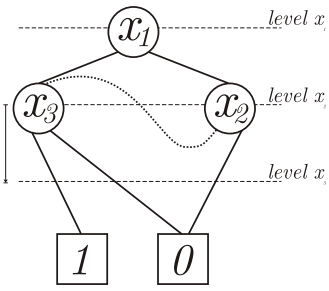


Figure 3: Example of structure analysis

3.4.2 Displaying the BDD

The actual displaying (in the following called 'painting' - derived from JAVA's `paint()`-methods) of the BDD is done by a special method in class `node`. The simplified code in Figure 4 gives an idea on how graphics are successively added to a `Graphics` object g.

Seen in a larger context, the `paint()`-method of the BDD's panel is responsible for a correct call sequence of the `structure_analysis-` and `paint_node`-method. The pseudo-code in Figure 5 gives an example.

By modifying `orig_x` and `orig_y` the Graph can be moved easily, e.g. by dragging it with the mouse.

4 GUI - Graphical User Interface

The visualization of BDDs is an important topic, since this gives some feedback how well the minimization techniques, like sifting, are doing. For this, most BDD packages presented so far [2, 10, 13, 8] have an output option to display the graph already built. But none of them allow any user interaction or visualization of the BDD manipulation, i.e. minimization and graph construction.

The key advantage of this BDD package implementation is the ability to change the global variable ordering. This can be achieved in different ways. Another important improvement is the visualization during automatic minimization using sifting heuristic and during BDD construction.

In the following we describe the *Graphical User Interface* (GUI) of our JAVA based BDD package (see also Figure 6 and 7 for GUI screenshots). The key features are:

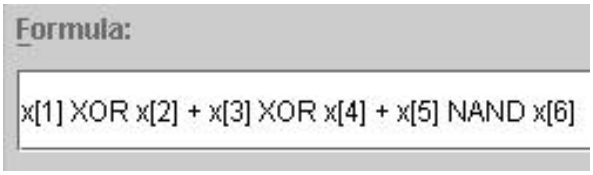
- **Trace mode:** During BDD construction using the `ite` operator each operand F, G and H is displayed in a window. By this, the construction can be observed step by step. This is very helpful when getting familiar with BDDs and also to debug programs.
- **Automatic minimization:** Sifting heuristic can be applied to the current BDD to minimize the number of nodes used. A *slideshow*-mode is available to watch the BDD after each sifting-step. Animation delay can be adjusted.
- **Setting the variable ordering:** Using the variable ordering window a (partial) ordering can be set and the BDD is directly modified accordingly. Algorithms can be tested easily this way (see also Figure 8).
- **Interactively change variable ordering:** Use the mouse to select a specific variable and transpose it up and down the current ordering while watching immediate reconstruction of the BDD accordingly.
- **Verbose mode:** In each step a protocol is reporting the current action. This simplifies the understanding of the algorithms.

```
public void paint_node(Graphics g) {
    int x = last_x;
    int y = last_y;
    if (is_leaf()) drawLeaf(g);
    else {
        g.setColor(edge_color);
        if (show_edgename) { // label edges with 0/1
            g.drawLine(x,y,G.last_x,G.last_y); // G son
            g.drawLine(x,y,H.last_x,H.last_y); // H son
            if (selected) {
                high_light_edge0();
                high_light_edge1();
            }
            derive_label_coord(l0_x, l0_y, l1_x, l1_y);
            g.drawString("0",l0_x,l0_y); // draw labels
            g.drawString("1",l1_x,l1_y);
        } else { // color edges instead
            g.setColor(edge0_color);
            g.drawLine(x,y,G.last_x,G.last_y);
            if (selected) high_light_edge0();
            g.setColor(edge1_color);
            g.drawLine(x,y,H.last_x,H.last_y);
            if (selected) high_light_edge1();
        }
        G.paint_node(g); // draw low-son
        H.paint_node(g); // draw high-son

        // draw node and label it
        g.setColor(node_fill_color);
        g.drawImage(nodeImage);
        if (selected) mark_node();
        g.setColor(node_text_color);
        g.drawString(variable_name,x,y);
    }
}
```

Figure 4: Recursive BDD-drawing

- **Simple input format:** A simple format allows to input formulas in a readable form, like `x[1]*x[2]+x[3]` stands for $x_1 AND x_2 OR x_3$. An example formula in the GUI looks as follows:



All operators listed in Table 1 are supported.

```
public void paint(Graphics g) {
        super.paint(g);
        my_root.reset_xy(); // reset coordinates
        structure_analysis(my_root, depth, orig_x, orig_y);
        if (show_Grid) drawGrid(g);
        if (show_statistics) drawStatistics(g);
        my_root.paint_node(g);
}
```

Figure 5: `paint()`-method of BDD-panel

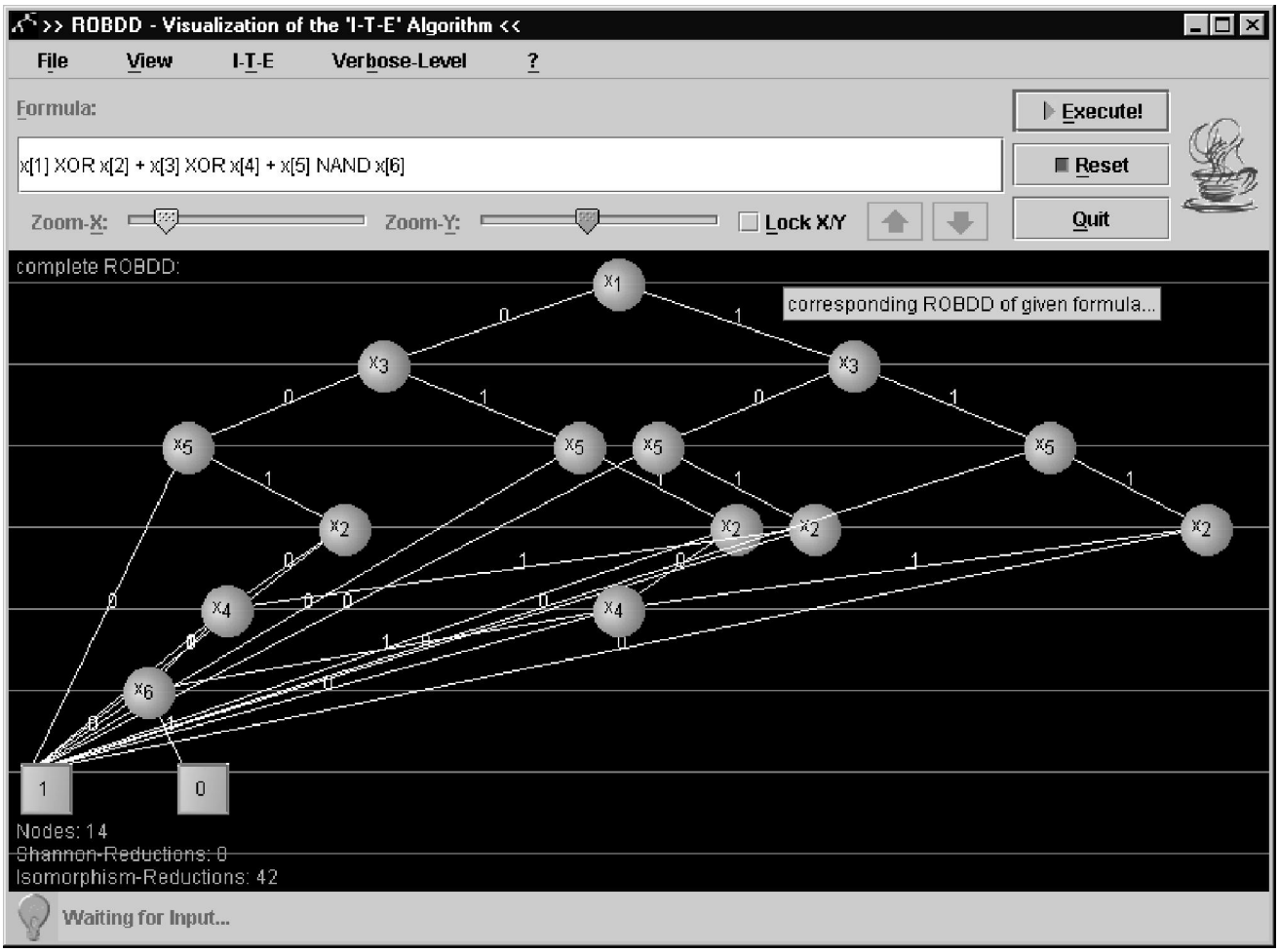


Figure 6: Screenshot before sifting

- **Resizing the BDD:** Use the mouse to easily move (click and drag) and resize the BDD to fit your current window size or to zoom into the data structure.
- **Swap high/low edges:** Low and high sons (THEN and ELSE branches) can be swapped, i.e. left son is *low* son and right son is *high* son or vice versa.

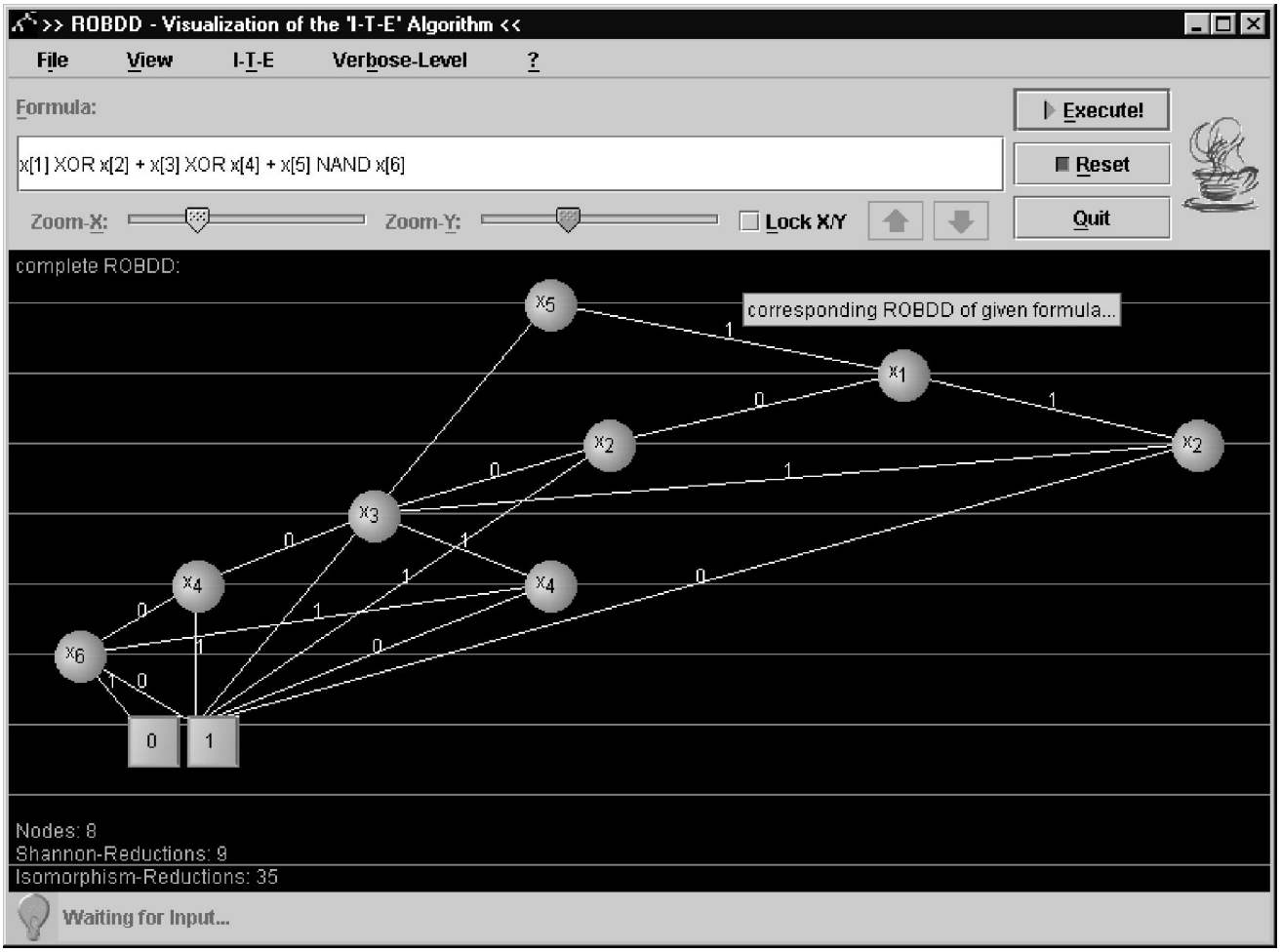


Figure 7: Screenshot after sifting

- **Keep configuration:** When finishing the program the current configuration can be stored in a file. Next time the program starts it may restore the saved values. The program supports several configuration files.

5 Conclusions

In this paper we presented a visualization of a JAVA based implementation of a BDD package introduced in [2]. We enhanced the recursive `ITE`-algorithm to accept different variable orderings which could be changed interactively. Additionally the sifting algorithm was implemented to allow automatic minimization of the BDD by finding a good variable ordering. An optional step-mode allowed a user to watch the BDD synthesis algorithm.

References

[1] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. on Comp.*, 45(9):993–1002, 1996.

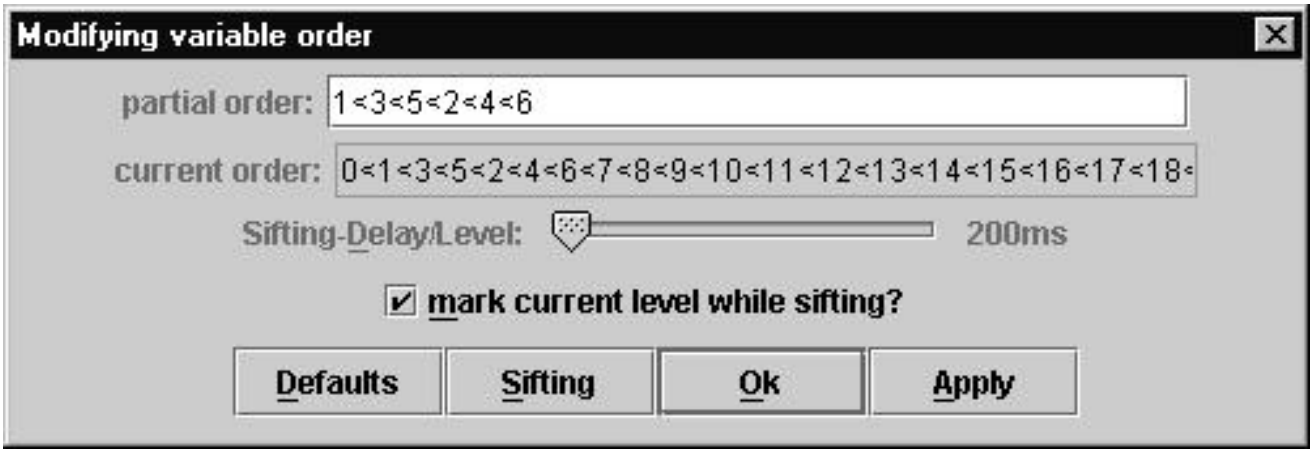


Figure 8: Setting the global variable ordering

[2] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conf.*, pages 40–45, 1990.

[3] R.E. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[4] R. Drechsler and B. Becker. *Binary Decision Diagrams – Theory and Implementation.* Kluwer Academic Publishers, 1998.

[5] S.J. Friedman and K.J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Design Automation Conf.*, pages 348–356, 1987.

[6] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 38–41, 1993.

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

[8] S. Höreth. *TUDD: Darmstadt University Decision Diagram Package Release 0.8b.* University of Darmstadt, Germany, 1999.

[9] S.E. Hudson. *Constructor of Useful Parsers, http://www.cs.princeton.edu/~appel/modern/java/CUP/.* 1998.

[10] D.E. Long. *Long-Package Sun Release 4.1 Overview of C Library Functions.* 1993.

[11] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Int'l Conf. on CAD*, pages 74–77, 1995.

[12] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47, 1993.

[13] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.1.2.* University of Colorado at Boulder, 1997.