# VisBDD – A Web-based Visualization Framework for OBDD Algorithms

## ABSTRACT

This paper presents a web-based visualization framework visBDD, intended for learning and understanding basic algorithms related to Ordered Binary Decision Diagrams (OBDDs). With visBDD the key algorithm for OBDD-synthesis, the ITE-algorithm can be explored in depth and step-by-step by the user, while all involved data structures can be made visible at different levels of abstraction. The framework is intended as well as for students as also for professionals, who want to gain more insight into the powerful OBDD data structure that is used in many verification and synthesis tools, for a better understanding of the underlying mechanisms and the problem complexity, without being dependent on reading textbooks.

## Keywords

www-based tools for visualization of OBDDs and education, OBDDs, algorithm visualization

## 1. INTRODUCTION

Visualization of algorithms is concerned with illustrating the behavior and the fundamental mechanisms and operations of the program during runtime. In general, displays as this have proven to be rather useful not only in education, but also for research as well as for understanding in different areas that are dependent on the development and analysis of such algorithms.

In this paper we present a visualization framework - visBDD, intended for analyzing algorithms for Ordered Binary Decision Diagrams (OBDDs). Dating back to the late 1950s [1,2] and being introduced as a canonical representation for Boolean functions by Bryant in 1986 [3], OBDDs are established as the state-of-the-art data structure in computer aided design of very large scale integrated (VLSI) circuits. Almost every step in this design cycle can be mapped to the task of manipulating Boolean functions and for performing these tasks efficiently in an automated way with a computer, OBDDs are very well suited, because they are compact, efficiently to manipulate, and canonical. For a detailed overview on OBDDs and their application in verification of combinatorial circuits, the analysis of sequential systems, or symbolic model checking see [4].

Usually, the research engineer in VLSI synthesis or verification does not care much, which data structure serves as the core of the program packages he is using. Often these tools are used in a one-push-button-technology way, without any knowledge of what is happening to the data structure in the background, because in many cases only effects are taking place that are analyzed on a higher level of abstraction. But, sometimes these effects are caused by intermediate computations on the underlying data structure and thus, deeper knowledge about the algorithmic behavior and the cause and effects regarding this particular data structure are mandatory.

Esp. for OBDD based tools, the computation of a simple Boolean operation might cause an exponential blow-up of the involved structures and the size of the result is extremely dependent on the chosen ordering of the input variables. For this reason, the engineer has to dig deep down into the OBDD basic algorithms to fully understand complexity issues related to algorithmic effects. The engineer has to read textbooks, trying to grasp the necessary concepts and ideas, and has to apply them to his own example. There is no question that if he masters all required steps, he is sort of an expert on that topic. But, the process is difficult and time consuming. Well suited visualizations of algorithms, connected with the possibility to process own examples can save a lot of time, keeping understanding of concepts much simpler. Of course, such a framework might not only be used in research and development, but also for all other educational purposes.

Today, there exist a lot of possibilities for the representation of learning materials based on computers. Visualization of learning content is rather important for the success in learning. Various studies [5] claim that esp. animations can increase the learning performance in comparison with pure textual representation of the learning content. In particular this applies to the teaching of algorithms and data structures.

Although, the textual description in words is probably the most exact way to define a subject, the process of apprehension is most difficult. In particular, this fact applies to OBDDs and the related algorithms. Many textbooks on OBDD technology use graphics to illustrate the algorithmic behavior. But, the realization of the whole dynamic process of an algorithm still remains rather difficult.

At the University of Toronto, Ronald Baecker developed a video for the illustration of a sorting algorithm [6]. This video was one of the first algorithm visualizations, but extremely expensive and complex in its production. In the 80s, two other important systems for visualization were developed: BALSA [7] and TANGO [8]. They were designed to illustrate data structures of a given algorithm. With later versions, like XTANGO [9] and Polka [10], it was possible to create concurrent animations for rendering object oriented algorithms.

But, these implementations suffered from being not platform independent, what is of the upmost importance for today's WWW-community. By using Java technology [11] it is possible to create animations for almost all computer systems and to make them accessible for everybody via the World Wide Web (WWW).

We decided not to use existing animation tools, because of the required system independence and the high level of specialization of the OBDD algorithms.

In contrast to other OBDD illustration tools, visBDD does not simply generate pictures of OBDD structures. The main objective of visBDD lies on the demonstration and explanation of the entire process of OBDD synthesis, while the user is able to navigate through different levels of abstraction. visBDD is a web-based application, designed in Java, and is freely accessible [12].

The paper is structured as follows. Section 2 recapitulates the most important facts about OBDDs and introduces the concepts of the main OBDD-algorithms. In Section 3, visBDD – our visualization tool for OBDD algorithms – is described in detail. Section 4 gives a short review about other competitive visualization tools for OBDDs and their shortcomings. Section 5 concludes the paper with an outlook on future work.

## 2. ORDERED BINARY DECISION DIAGRAMS (OBDDS)

All tasks in computer aided design of VLSI circuits are related to the manipulation of Boolean functions.

Let $X=\{x_0,\ldots,x_{n-1}\}$ be a set of Boolean variables and let $\pi: X \rightarrow \{0,1,2,\ldots,|X|=n-1\}$ be a bijective mapping of the variable indices. An Ordered Binary Decision Diagram (OBDD) is a rooted, directed, acyclic graph with the following properties: There are two distinct terminal nodes labeled with the Boolean constants 0 and 1. All non terminal nodes (sinks) are labeled with a Boolean variable $x_i$, $i \in \{0,\ldots,n-1\}$ and have two outgoing edges labeled with 0 and 1, respectively. The order in which the variables occur in the diagram is consistent with the variable order given by $\pi$, i.e. if there is an edge leading from a node labeled with $x_i$ to a node labeled with $x_j$, then $\pi(x_i)<\pi(x_j)$ must hold. On all paths from the root to a sink in the OBDD all variables must occur at most once.

To compute the function value of a function *f* given in terms of an OBDD for a given variable assignment $(a_0,\ldots,a_{n-1})$, $a_i \in \{0,1\}$ one is following a path starting in the root, switching at each node to the edge given by the according variable assignment $x_i=a_i$. The label of the reached sink determines the value of the function on that specific input.

For being a canonical representation of Boolean functions, the OBDD has to fulfill the following additional properties:

- The OBDD does not contain any isomorphic subgraphs, and
- The OBDD does not contain any redundant nodes, i.e. nodes, where both edges are pointing to the same successor node.

In each node v labeled with the variable $x_i$ of an OBDD G the Shannon-expansion of the function $f_v$ w.r.t. the variable $x_i$ is computed:

$$f_v = x_i f_v\,|_{xi=1} + \overline{x_i} f_v\,|_{xi=0}$$

$f_v|_{xi=1} = f_v(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_{n-1})$ is denoting the *positive cofactor* of $f_v$, which corresponds to the function represented by the 1-successor node of v. $f_v|_{xi=0} = f_v(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_{n-1})$ is denoting the *negative cofactor* of $f_v$, respectively, which corresponds to the function represented by the 0-successor node of v.

For the construction of an OBDD from a given circuit description, the symbolic simulation of the circuit is computed starting at the circuit's primary inputs, traversing the circuit gate by gate in direction to its primary outputs. The OBDD for each gate is computed by the application of the Boolean operator represented by the gate to the OBDDs of the gates' predecessors. For efficiency reasons, in symbolic simulation with OBDDs, all Boolean functions are mapped to a single general operation, which is able to express all Boolean operations, the so *called If-Then-Else-operator* (ITE) [14]. ITE is a three parameter function, computing *if x, then y, else z*,

$$\mathrm{ITE}(x,y,z) = x\cdot y + \overline{x}\cdot z$$

The ITE operation refers exactly to the operation that is performed in an OBDD node. To compute the ITE operation for three input OBDDs, ITE is evaluated recursively w.r.t. the top variable $x_i$ of the involved OBDDs, i.e. the variable that is first according to the given variable order $\pi$.

$$\mathrm{ITE}(f,g,h) = (x_i, \mathrm{ITE}(f|_{xi=1}, g|_{xi=1}, h|_{xi=1}), \mathrm{ITE}(f|_{xi=0}, g|_{xi=0}, h|_{xi=0}))$$

The recursion stops, if the first argument is constant, if the second and the third argument are both constant, or if the second and the third argument are equal.

```
Input:  f,g,h
Output: res = ITE(f,g,h)

ITE(f,g,h) {
  if (res = terminal_case(f,g,h) {
     return(res);
 }
 if (res = in_computed_table(f,g,h)){
     return(res);
 } else {
     x = top_variable(f,g,h);
     new_left = ITE(fx=1,gx=1,hx=1);
     new_right = ITE(fx=0,gx=0,hx=0);
     if (new_right == new_left){
       res = new_left;
     } else {
       res=create_node(x,new_left,new_right);
     }
     insert_in_computed_table(f,g,h,res);
  }
  return(res);
}
```

**Figure1**: **Outline of OBDD ITE-Algorithm in Pseudocode**

An outline of the ITE-algorithm in pseudocode is given in Fig. 1. Note that we make use of a cache for storing already computed results that is denoted as computed table. In fact, the maintenance of the computed table is rather important, because the cache is responsible for cutting down the runtime of the algorithm from exponential to polynomial time. The size of the resulting OBDD crucially depends on the chosen variable order $\pi$. For a simple function as e.g. the addition of two n-bit numbers, the size of the resulting OBDD ranges from linear *O(n)* to exponential *O(2^n)* in the number of variables, depending on the chosen variable order.

Thus, an in-depth insight in the basic algorithms and properties of OBDDs is mandatory for using this data structure, or also for using tools that are based on OBDDs in the most efficient way. Here, the benefits of a well suited visualization tool are obvious. By displaying all OBDDs that are required for a single synthesis step and performing the ITE algorithm step by step, while the user is able to adjust the variable order and to examine the effects that are taking place meanwhile for temporary OBDDs, a better understanding of these fundamental issues becomes much easier.

## 3. VisBDD – A VISUALIZATION TOOL FOR OBDD ALGORITHMS

Within the scope of the visBDD project, it is aspired to represent the very important OBDD-ITE algorithm in a appropriate graphical way. With the ITE algorithm it is possible to calculate the resulting OBDD for the application of an arbitrary Boolean operator * to two input OBDDs that are representing given Boolean functions.

To facilitate the most convenient way for the user to supply a Boolean function for the OBDD computation, we decided to make use of Boolean formulas. The formula entered by the user is partitioned into its basic operations and the result is calculated with the help of the ITE-algorithm. The whole computation procedure is dynamically visualized. The user of visBDD has the possibility to see the basic source code of the algorithm. Within the source code, he is able to navigate in all directions, i.e. forward and backward as well as up and down in the recursion. All parameters and data structures are visualized in a corresponding manner. Further, it is very important for the user to know, in which part of the computation the current step is located. This is realized with a derivation tree of the entered formula. The recursion graph of the ITE-algorithm is equivalent to the result of the computation and is constructed step by step. For all computation steps, the new evolving part of the graph is displayed without changing the general overall layout of the graph. Most visualization tools do not support these features. This is the main reason, why we decided to start the implementation of our own visualization framework. The requirement of acting on the user's own initiative respective of self-determined exploration of the algorithm is rather important for the implementation in practice. It should be in the hands of the user, to decide to continue with the next or the previous step. Also, he should have the possibility to step over a complete recursion to avoid to many details. These requirements force us to manage the visualization process in the following way: The whole computation is performed in advance. During the computation, all facts that are important for the visualization have to be memorized.

During the learning process the activation of different senses of the learner is rather important. Hereby, the usage of the visual and the auditory sense have been proven to be the most important. Due to this fact, we also decided to supply audio comments to the computation steps.

### 3.1 The computation process

To start the computation, the user has to enter a Boolean formula and a fixed variable order for the OBDD.

In a first step, a scanner probes for variables and operators, and afterwards, a syntax analyzer constructs the derivation tree (see Fig 2).

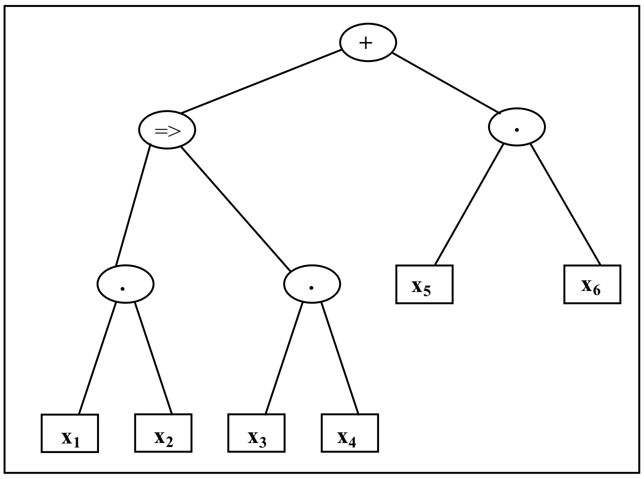


**Figure 2: The derivation tree for the formula**

**$x_1 \cdot x_2 \Rightarrow x_3 \cdot x_4 + x_5 \cdot x_6$**

By using the derivation tree it is simple to construct the corresponding OBDD with the ITE-operator by traversing the derivation tree in a depth-first-search (dfs) manner. Each Boolean operator is translated into an ITE-operator call. E.g. $x_1 \cdot x_2$ can be computed by ITE($x_1$,$x_2$,0). The OBDD for a Variable x can be determined as ITE(x,1,0).

In the given example visBDD creates OBDDs for each existing variable first. Then, the ITE-algorithm is executed consecutively for each operator in the derivation tree until the root node is reached.

For the computation and the visualization of the OBDD the following base classes are required:

- node representation,
- drawing layered graphs,
- parsing formulas,
- running OBDD algorithms,
- drawing trees,
- user interaction.

Next, we give a short description of the base classes, before we show, how to use the program.

### 3.2 The Node Class

The Class `Node` is used to store all OBDD nodes in an appropriate data structure. Each node is labeled with a reference to an input variable and contains two pointers to the 1-successor and the 0-successor, respectively. Most OBDD algorithms use graph traversal methods. For the application of these methods a flag is required to indicate if the node already has been visited or not. To distinguish leaf nodes and branching nodes, leaf nodes are labeled with a negative index.

The class `Node` contains various methods for graph manipulation, as, e.g. methods for graph traversal and computation of dimensions. Also a textual representation of an OBDD is implemented in the class `Node` (see Fig. 3).

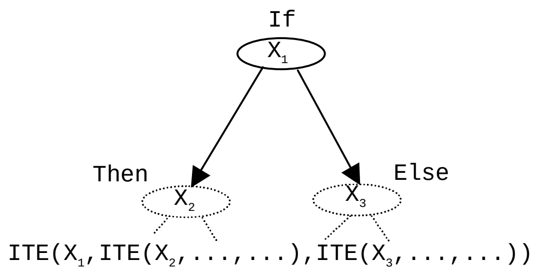


**Figure 3: ITE representation for binary graph structure**

The textual output of the binary graph structure can be used for initializing a unique table that serves as a storage container for nodes or the maintenance of a log file.

Also methods for drawing layered graphs are provided within the class `Node`. Fig. 4 shows an arbritrary OBDD graph structure.

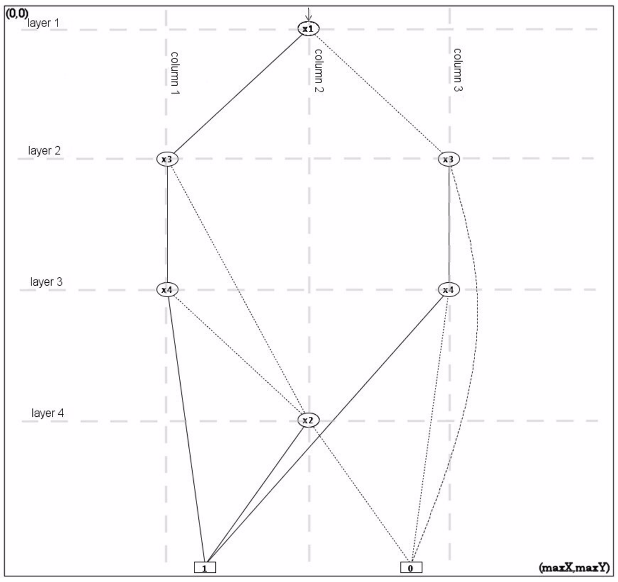


**Figure 4: OBDD representation of $x_1 \cdot x_2 + x_3 \cdot x_4$ according to $\pi$: $x_1 < x_3 < x_4 < x_2$**

For readability, graphs of OBDDs meet the following characteristics:

1. There exists just one root node with the identifier min{$\pi$}.
2. All nodes with the same identifier are located in the same level.
3. The levels are sorted according to the variable order.
4. Horizontally, the nodes are uniformly distributed.
5. Vertically, the levels are uniformly distributed.
6. There exist just two leaf nodes that are on fixed positions.
7. An edge is not allowed to be tangent to other nodes, but its terminal nodes.

Characteristics 1 to 6 are rather simple to implement by using the given variable order and the available graph traversal methods. Characteristic 7 can be solved using a working list algorithm as shown in Fig. 5.

```
Input:  Points X,Y and Collision Table CT
Output: List startList, which denotes the
        collision free connection
List startList = {};
List goalList ={};
startList.push(X);
goalList.push(Y);

while (goalList.notEmpty()) {
        X = startList.getLast();
        Y = goalList.getFirst();

        if (checkCollision(X,Y,CT)) {
                Z = getByPass();
                goalList.push(Z);
        } else {
                startList.push(goalList.pop());
        }
}
```

**Figure 5: Working List algorithm for collision detection**

The algorithm takes two coordinates *X,Y* and a list *CT* with known collision points as input parameters. Then, it is tested, if on the edge between the points *X* and *Y* are collisions. If a collision is found, a bypass point *Z* is calculated. Then, the algorithm checks for collisions in *(X,Z)* and *(Z,Y)*. The computation of a collision free edge is finished, if the *goalList* is empty.

A very important feature of the drawing methods is the possibility to draw the graph up to a predefined node without facing layout changes that conflict with easy readability. This feature can be utilized to construct the recursion graph step by step. The idea is to draw parts of the final graph only, while the entirely layouted graph is remaining invisible in the background. In each computation step, a drawing border is set. Therefore, the layout does never change, regardless, where the drawing border is set.

## 3.3 The BDD Class

In the `BDD` class all methods for OBDD manipulation are implemented. The unique table, the variable order, as well as additional variables and counters are initialized by its constructor.

### *3.3.1 Creating and Saving OBDDs*

All OBDDs, including temporary and final nodes, are stored in an unique table that is implemented as an array. Each slot of the array contains a pointer to the root node of an OBDD. For managing dynamic variable reordering, it is advantageous to maintain one unique table for each single variable, where only nodes labeled with this paritcular variable are stored.

### *3.3.2 ITE Algorithm*

For all possible binary Boolean operations, the corresponding ITE-operator calls are well defined. If the parser locates an arbitrary Boolean operation * applied to the OBDDs F and G, the corresponding ITE-operator call will be generated. E.g. the Boolean OR operation is defined as

```
Node Or(Node f, Node g) {
        return ite(f, ONE, g);
}
```

In visBDD, the algorithm for ITE is almost unchanged compared to the origanal algorithm shown in Figure 1. In addition, the method *setHistory(...)* saves all important details for the computation in a separate container. This includes pointers to all parameters, help texts, pointer to return values and final graphs, the drawing border, the source code line number, and a multifunctional action code that will be saved. Using this action code, we have the possibility to trigger special events, like e.g. audio comments.

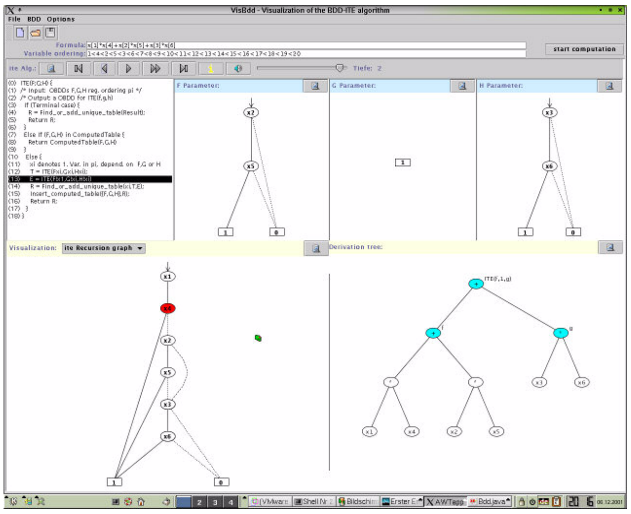
## 3.4 The Parser Class

In order to translate the Boolean formula into an OBDD, a scanner and parser implementation is required. The package CUP/JLex [13] is freely available for Java and can be used for that purpose. After defining a small grammar for Boolean formulas, an interface was added for interaction with the OBDD classes. In the first step, each Variable $x_i$ is translated into an OBDD via `getNode(i)`. According to the derivation tree, for each operator the parser calls the ITE-algorithm with the two OBDDs as operands to be combined.

## 3.5 Using the Program

visBDD is implemented as an Java applet and can be executed within any Web browser or simply as an Java 1.2 Jar file. The main window of visBDD is shown in Fig. 6.

visBDD includes a graphical user interface, which consists of a menu, a button bar, a code navigation bar, the source pane, and various visualization output panes.



Formula: x[1]*x[4]+x[2]*x[5]+x[3]*x[6]
Variable ordering: 1<4<2<5<3<6<7<8<9<10<11<12<13<14<15<16<17<18<19<20
Tiefe: 2
Visualization: ite Recursion graph

```
(0)  ITE(F,G,H) {
(1)  /* Input:  OBDDs F,G,H reg. ordering pi */
(2)  /* Output: a OBDD for ITE(f,g,h)
(3)    If (Terminal case) {
(4)      R = Find_or_add_unique_table(Result);
(5)      Return R;
(6)    }
(7)    Else If (F,G,H) in ComputedTable {
(8)      Return ComputedTable(F,G,H)
(9)    }
(10     Else {
(11)      xi denotes 1. Var. in pi, depend. on  F,G or H
(12)      T = ITE(Fxi,Gxi,Hxi);
(13)      E = ITE(Fx̄i,Gx̄i,Hx̄i)
(14)      R = Find_or_add_unique_table(xi,T,E);
(15)      Insert_computed_table({F,G,H},R);
(16)      Return R;
(17)   }
(18) }
```

**Figure 6: Main window of visBDD**

The code navigation bar, as shown in Fig. 7, provides the following functions:

- Information about the current step,
- jump to the beginning of the computation,
- step back / forward in the computation,
- step over the recursion, display the result only,
- go to the final step of the computation,
- give information about the overall computation,
- play an audio comment.

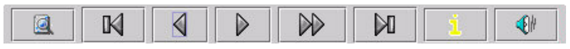
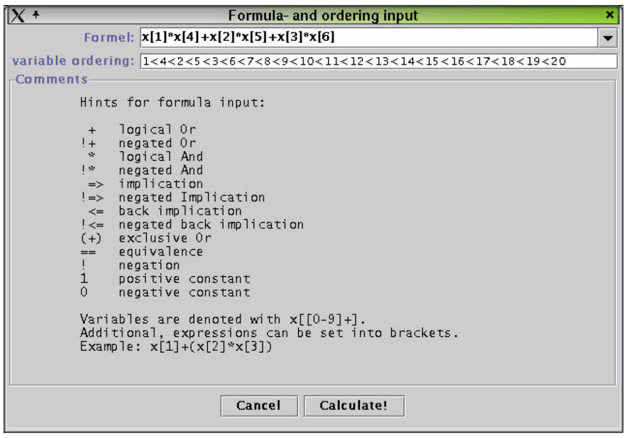


**Figure 7: The code navigation bar**

The input of the formula and the variable ordering can be given in the main window of visBDD or with a separate input dialog box. In the input dialog, the user can choose one of several provided example formulas or he can enter a formula of his own choice. An online help on different possibilities for entering formulas is also provided. The formula input dialog is shown in Fig. 8.



Formel: x[1]*x[4]+x[2]*x[5]+x[3]*x[6]

variable ordering: 1<4<2<5<3<6<7<8<9<10<11<12<13<14<15<16<17<18<19<20

Comments

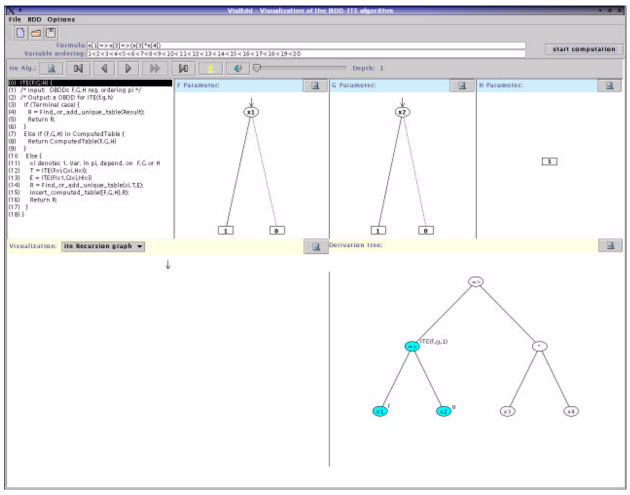
```
Hints for formula input:

 +   logical Or
!+   negated Or
 *   logical And
!*   negated And
 =>  implication
!=>  negated Implication
 <=  back implication
!<=  negated back implication
(+)  exclusive Or
==   equivalence
!    negation
1    positive constant
0    negative constant

Variables are denoted with x[[0-9]+].
Additional, expressions can be set into brackets.
Example: x[1]+(x[2]*x[3])
```

**Figure 8: The input dialog for formulas and variable ordering**

After a formula is entered, visBDD starts the OBDD computation (Fig. 9). In the following screenshots the computation of the formula $x_1 \rightarrow x_2 \rightarrow (x_3 \cdot x_4)$ with visBDD is shown.



Formula: x[1]=>x[2]=>(x[3]*x[4])

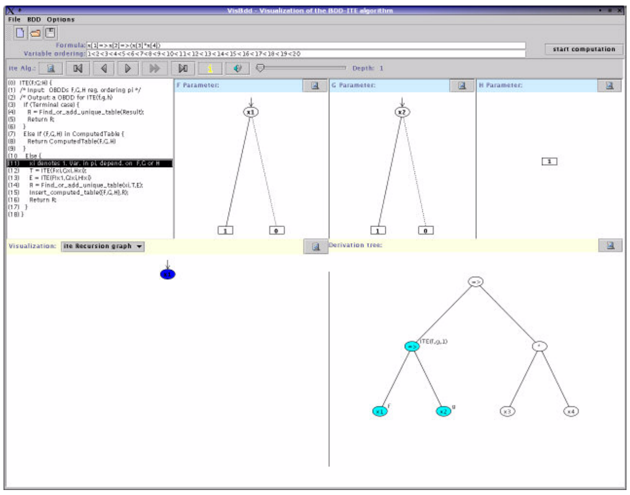
Variable ordering: 1<2<3<4<5<6<7<8<9<10<11<12<13<14<15<16<17<18<19<20

```
ITE(F,G,H) {
/* Input: OBDDs F,G,H reg. ordering pi */
/* Output: a OBDD for ITE(f,g,h)
 if (Terminal case) {
   R = Find_or_add_unique_table(Result);
   Return R;
 }
 Else if (F,G,H) in ComputedTable {
   Return ComputedTable(F,G,H)
 }
 Else {
   xi denotes 1. Var. in pi, depend. on F,G or H
   T = ITE(Fxi,Gxi,Hxi);
   E = ITE(F!xi,G!xi,H!xi);
   R = Find_or_add_unique_table(xi,T,E);
   insert_computed_table((F,G,H),R);
   Return R;
 }
}
```

**Figure 9: The start of a computation**

Now, the user can step through the source code line by line and start his analysis. The current code line is highlighted with a black bar. As mentioned before, ITE is a three-paramter function. Thus, the three input-OBDDs of ITE are displayed on the right next to the source code pane. The navigation with the code navigation bar enables the user to gain insight in the dynamic behavior of the algorithm in a rather natural way. The navigation is quite similar to a source code debugger. But, in contrast to a debugger, visBDD supplies also detailed graphical illustrations and additional information on data structures. As a very important feature visBDD allows the user to define his own schedule on how deep he wants to step into the recursive algorithm. If the current source line is a recursive call of ITE, it is also possible to step over the recursion and to see the result directly.

During the visualization process, the recursion graph is displayed in the lower left window step by step (see Fig. 10). In this window, the user can switch between the view of the ITE recursion graph view and the result of an ITE call.



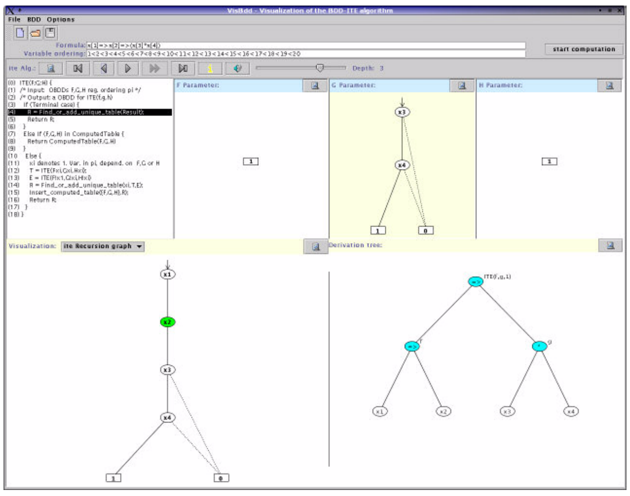
Formula: x[1]=>x[2]=>(x[3]*x[4])

Variable ordering: 1<2<3<4<5<6<7<8<9<10<11<12<13<14<15<16<17<18<19<20

```
ITE(F,G,H) {
/* Input: OBDDs F,G,H reg. ordering pi */
/* Output: a OBDD for ITE(f,g,h)
 if (Terminal case) {
   R = Find_or_add_unique_table(Result);
   Return R;
 }
 Else if (F,G,H) in ComputedTable {
   Return ComputedTable(F,G,H)
 }
 Else {
   xi denotes 1. Var. in pi, depend. on F,G or H
   T = ITE(Fxi,Gxi,Hxi);
   E = ITE(F!xi,G!xi,H!xi);
   R = Find_or_add_unique_table(xi,T,E);
   insert_computed_table((F,G,H),R);
   Return R;
 }
}
```

**Figure 10: Step by Step visualization with different markup colors.**

Inside this graph, current top variables are marked with colors. For positive and negative cofactor computation different colors are used (green and red) as shown in Fig. 11. A derivation tree is also shown in the lower right window. The actual part of the computation is marked in the tree. With this information the user is always able to realize in which part of the computation he is currently located.



Formula: x[1]=>x[2]=>(x[3]*x[4])

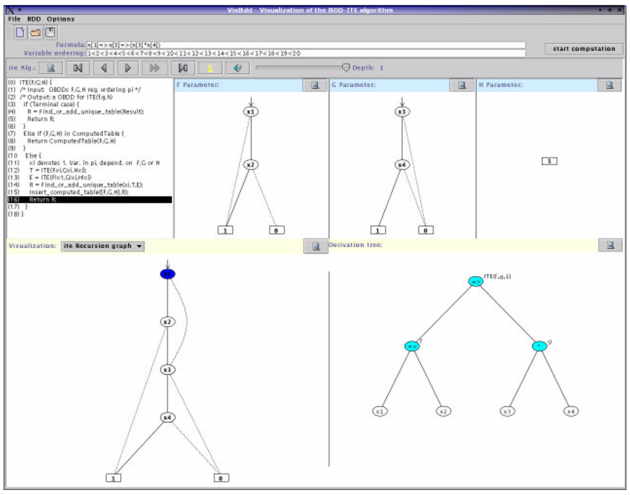
Variable ordering: 1<2<3<4<5<6<7<8<9<10<11<12<13<14<15<16<17<18<19<20

```
ITE(F,G,H) {
/* Input: OBDDs F,G,H reg. ordering pi */
/* Output: a OBDD for ITE(f,g,h)
 if (Terminal case) {
   R = Find_or_add_unique_table(Result);
   Return R;
 }
 Else if (F,G,H) in ComputedTable {
   Return ComputedTable(F,G,H)
 }
 Else {
   xi denotes 1. Var. in pi, depend. on F,G or H
   T = ITE(Fxi,Gxi,Hxi);
   E = ITE(F!xi,G!xi,H!xi);
   R = Find_or_add_unique_table(xi,T,E);
   insert_computed_table((F,G,H),R);
   Return R;
 }
}
```

**Figure 10: The end of a negative cofactor computation. The result parameter is marked and added to the result graph.**

For each visualization pane, additional textual information can be displayed. In the case that a visualization output pane is getting too small, it can be enlarged by clicking on its magnification icon in the upper right corner.

By performing the entire OBDD computation in advance we have the possibility to navigate in any direction, while the graphical layout of the displayed OBDDs does not change in general (see Fig. 11). Thus, it is much easier for the user to understand and to follow up the computation. If the graphical layout would be permanently adapted for fitting best into the given window size, the OBDD graph would look rather nice, but for the user it would be difficult to relate two consecutive computation steps.



Formula: x[1]=>x[2]=>(x[3]*x[4])
Variable ordering: 1<2<3<4<5<6<7<8<9<10<11<12<13<14<15<16<17<18<19<20

```
ITE(F,G,H) {
/* Input: OBDDs F,G,H reg. ordering pi */
/* Output: a OBDD for ITE(f,g,h)
 If (Terminal case) {
   R = Find_or_add_unique_table(Result);
   Return R;
 }
 Else if (F,G,H) in ComputedTable {
   Return ComputedTable(F,G,H)
 }
 Else {
   xi denotes 1. Var. in pi, depend. on F,G or H
   T = ITE(Fxi,Gxi,Hxi);
   E = ITE(Fix1,Gixi,Hixi)
   R = Find_or_add_unique_table(xi,T,E);
   insert_computed_table({F,G,H},R);
   Return R;
 }
}
```

**Figure 11: The computation is complete. The result graph grew step by step without changing the layout (compare to Fig. 10).**

In addition, the OBDD computation can be triggered to run autonomously. The user has the possibility to stop the animation at any time and to continue the computation stepwise again.

VisBDD is the first "easy to use" visualization tool for OBDD algorithms.

## 4. OTHER VISUALIZATION TOOLS FOR OBDDS

The most important reason, why visBDD has been developed, was the need to visualize the process that takes place during OBDD synthesis algorithm execution. Other existing graphical tools for OBDD illustration are almost only focusing on the representation of already computed OBDDs, neglecting the fact, what happened during OBDD computation time.

`Dot` and `neato` [16] are well known drawing tools for directed and undirected graphs. Being a general purpose graph drawing toolset, these programs of course can be used to draw OBDDs. In fact, the CUDD OBDD program package [17], which is the most popular open source high capacity OBDD package, uses `dot` as an output filter to illustrate resultant OBDDs. But, `dot` is only a static graph layout generator and in particular not capable to perform an entire algorithm visualization.

At the university of Freiburg (Germany) a visualization tool for OBDDs was developed (ROBDD) [18]. Being a native Java application the platform independence of the package is guaranteed. Besides the exploration of the ITE-algorithm, also the effect of variable reordering algorithms can be analyzed with ROBDD. But, compared to visBDD, the possibility to explore and to analyze the single steps of the ITE-algorithm is missing. Additionally, also the required degrees of freedom for the user to navigate through the algorithm are not fulfilled.

Höreth developed a BDD manipulation package (TUDD) [19] at the University of Darmstadt (Germany) that is capable of managing various types of binary decision diagrams. In addition, he provided the possibility to illustrate the optimization process of the BDDs by displaying snapshot of BDDs during variable reordering time. But, this illustration is only static and the user has no posibility to interact.

At the university of Trier (Germany) a BDD portal-web-site is supported: www.bdd-portal.org [15]. Besides providing a huge archive on all research topics related to OBDDs and derivatives of OBDDs, the portal also offers an experimentation testbed for performing OBDD benchmark testing on an independent platform. Additionally, an illustration and visualization tool, the BDD-Calculator is provided, where samples of OBDD computations and OBDD optimization can be analyzed. Boolean formulas directly entered by the user can as well be processed as benchmark circuits taken from a library or supplied by the user. The BDD-Calculator serves only as an illustration tool, displaying resultant OBDDs with the help of `dot` and giving statistical information about the results of the conducted experiments.

Of course we are aware of the fact that we have only given an incomplete sample of all available graphical OBDD illustration frameworks. But, these are some of the best known examples of graphical representation tools for OBDDs. In contrast to all of them, visBDD enables the user to learn about the features and to understand the properties of the OBDD synthesis algorithm. By experiencing the effects of changing variable orders or other parameters, the user is able to gain a deeper insight into OBDD technology and its characteristics.

## 5. CONCLUSION AND FUTURE WORK

We have presented a framework for the visualization of OBDD algorithms. Currently, only the ITE-algorithm that is used for OBDD synthesis is implemented. But, the generic approach chosen for our implementation offers the possibility of an easy integration of additional OBDD-algorithms, as e.g. the `constrain`-operator or the `restrict`-operator [19, 20]. Also a visualization of optimization algorithms for OBDDs, as e.g. `sifting` [21] is possible.

VisBDD is already used by students for lectures and lab courses on OBDDs. But, it is also intended to be an accessory application in the toolbox of the professional engineer.

VisBDD is accessible via the WWW [12] and is integrated as a supplementary tool into the BDD-portal-website www.bdd-portal.org [15], where also other OBDD tools and links to most of the already mentioned OBDD-illustration-tools are provided.

## 6. REFERENCES

[1] C. Y. Lee: *Representation of switching circuits by binary decision-programs*, The Bell Systems Technical Journal 38, 1959, pp. 985-999.

[2] S. B. Akers: *Binary Decision Diagrams*, IEEE Trans. on Computers 27, 1978, pp. 509-516.

[3] R. E. Bryant: *Graph Based Algorithms for Boolean Function Manipulation*, IEEE Trans. on Computers 35, 1986, pp. 677-695.

[4] Ch. Meinel, T. Theobald: *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications*, Springer, Heidelberg, 1998.

*[5]* D. Lewalter: *Lernen mit Bildern und Animationen (Learning with Pictures and Animations)* Waxmann, Münster 1997 (in german)

[6] *R. Baecker, D. Sherman: Sorting* out Sorting, University of Toronto, 1981.

[7] M.H. Brown, R. Sedgewick*: A System for Algorithm Animation*, Computer Graphics July 1984, pp. 177-186.

[8] J.T. Stasko: Tango: *A Framework and System for Algorithm Animation*, IEEE Computer 23, September 1990, pp. 27-39.

[9] J.T. *Stasko: Animation algorithm with XTANGO*, SIGACT News 23, Spring 1992, pp.67-71.

[10] POLKA: http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html

[11] JAVA: http://java.sun.com

[12] visBDD is available at: http://www.bbd-portal.org/visBDD

[13] JLEX and CUP: http://www.cs.princeton.edu/~appel/modern/java/JLex/ http://www.cs.princeton.edu/~appel/modern/java/CUP/

[14] K.S. Brace, R.L. Rudell, R.E. Bryant: *Efficient Implementation of a BDD package*, in Proc. of the 27th ACM/IEEE Design Automation Conference, 1990, pp. 40-45.

[15] The BDD Portal website: http://www.bdd-portal.org/

[16] E.R. Ganser, S.C. North: *An open graph visualization system and its application to software engeneering*, Software – Practice and Experience, 30(11), 2000, pp. 1203-1233.

[17] F. Somenzi: *Efficient manipulation of decision diagrams*, Int. Journal on Software Tools for Technology Transfer (STTT),3(2), 2001, pp. 171-181. CUDD-Package is available at: http://vlsi.colorado.edu/pub/

[18] J. Römmler, R. Drechsler: *Implementierung und Visualisierung eines BDD-Packages in Java (implementation and visualization of a BDD package with java)*, in german, paper and package available at: http://ira.informatik.uni-freiburg.de/.index.de.html

[19] S. Höreth: *A word-level graph manipulation package*. Int. Journal on Software Tools for Technology Transfer (STTT),3(2), 2001, pp. 182-192. TUDD package available at: http://www.rs.e-technik.th-darmstadt.de/~sth/download.html

[20] C. Meinel, A. Wagner: *WWW.BDD-PORTAL.ORG* Proc. IWLS'2000, Dana Point (USA), 2000. The BDD-Portal Website: http://www.bdd-portal.org/

[21] O. Coudert, C. Berthet, J.C. Madre: *Verification of synchronous sequential machines based on symbolic execution*, Workshop on Automatic Verification Methods for Finite State Systems. LNCS 407, 1989, pp. 365-373.

[22] O. Coudert, C. Berthet, J.C. Madre: *Verification of sequential machines using Boolean functional vectors*, IMEC-IFIP Workshop Applied Formal Methods for Correct VLSI Design, 1989, pp. 111-128.

[23] R. Rudell: *Dynamic Variable Ordering for Ordered Binary Decision Diagrams*, in Proc. IEEE Int. Conference on Computer Aided Design (Santa Clara, CA), 1993, pp. 42-47.