Controlling the Memory During Manipulation of Word-Level Decision Diagrams

Sebastian Kinder Görschwin Fey Rolf Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{kinder,fey,drechsle}@informatik.uni-bremen.de

Abstract

Decision Diagrams (DDs) are efficient data structures that are frequently used for formal verification, test or synthesis of circuits and systems. The main drawback of the data structure is the potential memory blow-up caused by certain functions. Therefore streaming has been proposed as a technique to efficiently evaluate operations on Binary DDs. The maximal memory usage can be controlled and therefore calculations can be carried out with a small amount of memory.

In this paper we show how streaming can be extended to Word-Level Decision Diagrams (WLDDs) and for which type of diagrams streaming is applicable. A detailed description of streaming for Multi-Terminal BDDs (MTBDDs) is given. Experiments show the efficiency and the small memory needs for operations on MTBDDs.

1. Introduction

The size of circuits and systems doubles according to Moore's law every 18 months. In the same way the size of problems occuring in logic synthesis, verification or test pattern generation increases rapidly. Therefore efficient data structures and algorithms are needed. One frequently used data structure are *Binary Decision Diagrams* (BDDs) [4], as Boolean operations can be carried out efficiently on this representation. The main drawback is the potential memory blow-up caused by certain classes of functions, e.g. multipliers [4]. Several other types of diagrams have been proposed to improve the behaviour. The two main differences from BDDs are the decomopsition type and the extension to word-level representations. For example *Word-Level Decision Diagrams* (WLDDs) have been shown to be beneficial for purposes such as data-path verification [8, 11]. But exponential trade-offs between different types of diagrams can be shown [2], i.e. while for some functions one type is much better suited, but the opposite holds, when other functions are considered. In this sense there does not exist a single type of DD that is superior to all others.

Thus, the main memory can be insufficient regardless of the type of DD that is used. Techniques to controll the memory usage are needed. Streaming of BDDs has been proposed as a technique to handle large BDDs with a small amount of dynamic memory [14]. The basic idea is to reduce the memory needs by keeping only subgraphs in the main memory while most of the data is stored on a hard disk. As a result a trade-off between run-time and memory needs becomes possible for calculations on BDDs.

In this paper the streaming technique is extended to WLDDs. The representation of other types of *Decision Diagrams* (DDs) for streaming is discussed and the influence of the decomposition type on the efficiency of streaming is analyzed. Furthermore the implementation of streaming for the word-level is studied examplarily with *Multi-Terminal BDDs* (MTBDDs) [7], that are also known as Algebraic DDs [1]. Based on this study the main memory needed to carry out a particular operation can be estimated in advance and further restricted when necessary. When run-time limits are necessary a partial evaluation is possible by only considering a prefix of the stream representation.

The paper is structured as follows: Preliminaries are given in Section 2. The streaming for BDDs is revisited in Section 3. In Section 4 the extension of streaming to other types of diagrams is discussed. Section 5 gives a detailed insight how streaming for MTBDDs is realized. Experimental results are given in Section 6. Conclusions are presented in Section 7.

2. Preliminaries

In the following several functions are needed for decomposition:

$$
\begin{aligned}
f_{0_x} &= f(x_1,\ldots,x_i=0,\ldots,x_n)\\
f_{1_x} &= f(x_1,\ldots,x_i=1,\ldots,x_n)\\
f_{2_x} &= \partial f/\partial x
\end{aligned}
$$

For Boolean functions $f_{2_x} = f_{0_x} \oplus f_{1_x}$ holds. In the following the index x is not printed, when it is clear from the context.

A DD is a directed acyclic graph $G = (V, E)$. A node in the graph represents a function $f : \mathbb{D}^n \to \mathbb{E}$, i.e. a function over n variables from the domain $\mathbb{D}$ into the function domain $\mathbb{E}$. Leaf nodes represent constant functions $f = a, a \in \mathbb{E}$. Each internal node v is labled by a variable $var(v) \in \{x_1,\ldots,x_n\}$. At v the function f is decomposed into subfunctions with respect to $x_i = var(f)$. The children *0-child*, *1-child*, ...*k-child* of v represent the subfunctions that are independet of x_i. The variable domain $\mathbb{D}$, the function domain $\mathbb{E}$ and the type of decompositions used depend on the type of DD. Additionally additive and/or multiplicative weights are assoche edges for some types of DDs.

The DD is called ordered, when all variables occur in the same order on all paths to the terminals. The DD is called reduced, if it does neither contain isomorphic subgraphs nor nodes that can be removed without changing the function represented by the graph. Depending on the diagram type additional restrictions besides being ordered and reduced are necessary to retrieve a canonic representation of functions. In the following a node and the function represented by a node are denoted by the same symbol.

BDDs represent Boolean functions, i.e. $\mathbb{D} = \mathbb{E} = \mathbb{B} = \{0, 1\}$. The Shannon decomposition is applied at internal nodes. A node f labled with $x = var(f)$ has two successors $f \to$*0-child* $= f_{0_x}$ and $f \to$*1-child* $= f_{1_x}$.

MTBDDs are a word-level extension to represent functions $f : \mathbb{B}^n \to \mathbb{R}$. The same decomposition type as in BDDs is used.

FDDs [13, 10] have the same structure as BDDs, but instead of the Shannon decomposition the positive or negative Davio decomposition is used.

3. Streaming BDDs

An algorithm for BDD manipulation based on the streaming data model was introduced in [14]. The basic idea is to serialize the BDD structure into a stream file. The scanning of such a file corresponds to a depth-first traversal of a BDD. In this way a logic operation can be computed without using internal memory.

To serialize a BDD, nodes are represented by a pair of children in parentheses and an id written as:

(0-child 1-child):nodeID

An explicit notation of the input variable for each node is not necessary because the context of parentheses indicates the corresponding variable. When the same *nodeID* appears successively twice in a stream, the second appearance can be suppressed. For example, *((T1 T0):N1 N1)* can be reduced to *((T1 T0):N1)*.

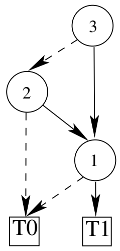


Figure 1. Decision Diagram

Example 1. *Figure 1 shows a BDD. The nodes are labled with their nodeID. The corresponding stream is:*
((T0 (T0 T1):1):2 1):3.

The stream is read sequentially and the subgraph rooted at the *0-child* is read before the subgraph rooted at the *1-child*. Therefore the corresponding BDD is traversed in a depth-first order. Given two BDDs F and G in stream representation an operation $H = F \circ G$ is carried out by succesively reading the streams and generating the result of the operation. This is indicated in Figure 2. The usual apply operation for BDDs as shown in Figure 3 is used to process streams.

Writing all subgraphs always completely into the stream would cause an exponential length of the stream in most cases. Therefore for each stream an internal table is used to store the subgraph corresponding to a certain *nodeID*. Thus, a reoccurring *nodeID* can be processed by switching from reading the stream to a traversal of the table in memory. On the other hand the size of the internal table is limited. When the number of nodes occuring in the BDD exceeds the number of nodes stored in the table, nodes are replaced to represent a new subgraph, i.e. a particular *nodeID* is *recycled*. If a node is recycled all nodes above this node become temporary nodes and can not be associated to a *nodeID*. Thus, the internal table contains the complete subgraph that is rooted

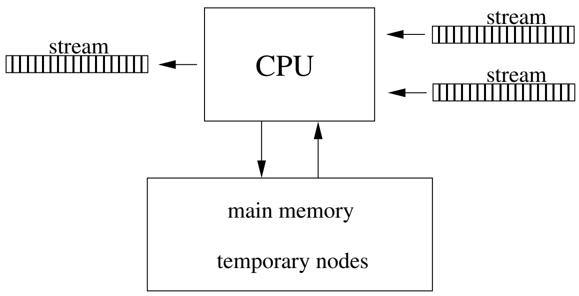


Figure 2. Streaming computation

```
nodeID apply (F,G,op) {
  if (terminalCase(F,G,op,&R)
    return R;
  if (lookup_computedTable(F,G,op,&R))
    return R;
  v= top_variable(F,G);
  R0= apply(Fv=0, Gv=0, op);
  R1= apply(Fv=1, Gv=1, op);
  if (R1 == R0) return R1;
  R= find_or_add_unique(v, R0, R1);
  insert_computedTable(F, G, op, R);
  return R;
}
```

Figure 3. Apply operation on BDDs

at a *nodeID*. The recycling process leads to non-canonicity of the stream representation. To maintain the efficiency of operations a cache for already computed operations is maintained. A result from the cache is only valid, if none of the corresponding nodes has been recycled intermediately. For a more detailed discussion we refer the reader to [14] and to Section 5.

4. Extension to the Word-Level

So far the streaming technique has only been applied to BDDs. But to allow for more compact representations the extension of the technique to other DD types is favourable. Two preconditions have to be fullfilled by DDs to allow for the application of streaming:

1. The diagram has to be represented as a stream.
2. The operations have to be carried out by reading the input streams.

The first precondition is fullfilled by all DDs that are considered in the following. A directed acyclic graph is the common basic structure of the DDs, only the number of children may differ, e.g. for MVDDs [12] or TEDs [6]. The extension to diagrams that have more than two children per node is straight-forward. A node can be represented as

(0-child 1-child . . . k-child):nodeID

The supression of successive equal *nodeID*s is only applicable, when all children of a node are equal and all nodes have the same number of successors. Otherwise equal *nodeID*s have to be repeated to guarantee, that the original graph can be reconstructed. Another generalization of BDDs are multiplicative and additive edge weights. The basic node representation can also be extended to annotate weights:

(0-child 0-add 0-mult . . . k-child k-add k-mult):nodeID

The second precondition is more restricitve due to the serialization inherent in the streaming concept. Reading parts in the stream corresponds to traversing subgraphs of the diagram. After having read a part of the stream only subgraphs that correspond to nodes in the internal tables are available. But due to recycling, this is usually not the complete diagram. To evaluate the same subgraph twice the stream has to be read more than once. The decomposition used for a paritcular diagram type determines the scheme of reading.

At first consider BDDs. The Shannon decomposition $f = \overline{x} \cdot f_0 + x \cdot f_1$ is used to construct BDDs. Therefore each Boolean operation $\circ$ on two BDD nodes f and g can be recursively calculated on the children

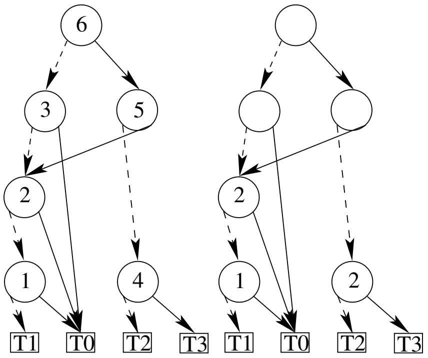
$$f{\rightarrow}\textit{0-child} = f_0, f{\rightarrow}\textit{1-child} = f_1,$$
$$g{\rightarrow}\textit{0-child} = g_0, g{\rightarrow}\textit{1-child} = g_1 :$$

$$\begin{aligned} f \circ g &= (\overline{x} \cdot f_0 + x \cdot f_1) \circ (\overline{x} \cdot g_0 + x \cdot g_1) \\ &= \overline{x} \cdot (f_0 \circ g_0) + x \cdot (f_1 \circ g_1) \end{aligned}$$

Therefore BDDs can be efficiently evaluated as streams, because subgraphs can be treated independently. This result transfers to other types of diagrams, that are based on the Shannon decomposition and its derivatives, as e.g. Multi-Terminal BDDs or Multi-Valued DDs.

On the other hand the Davio decomposition has been proposed for FDDs. For example the positive Davio decomposition is defined as follows: $f = f_0 \oplus x \cdot (f_0 \oplus f_1)$. Correspondingly the children of a node where the positive Davio decomposition is applied are given by: $f \rightarrow \textit{0-child} = f_0$, $f \rightarrow \textit{1-child} = f_2 = f_0 \oplus f_1$. Consider the AND-concatenation of two Davio nodes:

$$\begin{aligned} f \cdot g &= (f_0 \oplus x f_2) \cdot (g_0 \oplus x g_2) \\ &= (f_0 \cdot g_0) \oplus x(f_2 \cdot g_2) \\ &\quad \oplus (f_0 \cdot g_2) \oplus (f_2 \cdot g_0)) \end{aligned}$$



(a) $maxID = 6$ (b) $maxID = 2$

Figure 4. Example Diagrams

In this case subgraphs can not be treated independently, i.e. the *0-child* has to be considered to calculate the *1-child*. Therefore reading the stream only once is too restrictive for processing the stream representation of FDDs. In fact the concatenation of two FDDs with a number of nodes polynomial in the number of variables can lead to an FDD with an exponential number of nodes [3]. Therefore parts of a stream may have to be read an exponential number of times. Again, this result transfers to other diagram types that do not allow to treat subgraphs independently, e.g. KFDDs [9], *BMDs [5] and TEDs [6].

In the following we present a case study for MTBDDs.

5. Streaming MTBDDs

In this section the concept from [14] is extended to MTBDDs. An example for an MTBDD is given in Figure 4. To construct this diagram a hash table is needed to identify already existing nodes. If the table is large enough, the MTBDD stream is a serialized representation for the conventional MTBDD. If the capacity of the table is insufficient, conventional packages cannot continue their execution, but a streaming MTBDD can still be calculated. If a subgraph of the MTBDD is missing in the hash table, a duplicate will appear in the stream. The size of the hash table is specified by a *maxID* which is given as a prefix of the stream. Each node is identified by an integer from 1 to *maxID*.

In streaming MTBDDs two different kinds of nodes occur, namely *StoredNodes* and *TempNodes*. A *StoredNode* refers to a node which is stored in the table, a *TempNode* is a temporary node which is immediately lost. *StoredNodes* cannot have *TempNodes* among their successors. A *TempNode* occurs when the table size is insufficient and a *nodeID*

```
nodeID apply(str1,str2,op){
  nodeID1=getCurrentNodeID(str1);
  nodeID2=getCurrentNodeID(str2);
  if(isInComputedTable(nodeID1,
                         nodeID2))
    return result;

  readLow(str1);
  readLow(str2);

  Rlow=apply(str1,str2,op);
  if(storedNode(Rlow))
    time1=getTimestamp(Rlow);

  readHigh(str1);
  readHigh(str2);

  Rhigh=apply(str1,str2,op);
  if(storedNode(Rhigh))
    time2=getTimestamp(Rhigh);

  readIndex(str1);
  readIndex(str2);

  if(Rlow= Rhigh && time1==time2){
    if(tempNode(Rlow)
      printResultToFile(outStr);
    nodeID1=getCurrentNodeID(str1);
    nodeID2=getCurrentNodeID(str2);
    insertIntoComputedTable(nodeID1,
                              nodeID2);
    return Rlow;
  }
  nodeID1=getCurrentNodeID(str1);
  nodeID2=getCurrentNodeID(str2);
  R=findOrAddInUniqueTable(Rlow,
                            Rhigh);
  if(newEntry(R)){
    printResultToFile(outStr);
    insertIntoComputedTable(nodeID1,
                              nodeID2);
    return R;
  } else
    return R;
}
```

Figure 5. Apply for streaming MTBDDs

that has been used previously is recycled. In such a case the MTBDD stream is not a unique representation of the original function. If the table size is insufficient the algorithm allows to reuse "orphan" *nodeID*s, i.e. nodes that are not referenced from other nodes. If there are multiple candidates to be recycled, the one waiting longest is chosen. Example 2 shows two different MTBDD streams with different *maxID*s for the same function.

Example 2. *The two MTBDDs shown in Figure 4 represent the same function. Six nodes are sufficient to completely store the MTBDD in the internal table. Figure 4(a) shows the nodeIDs as they correspond to the stream with maxID= 6:*

6((((T1 T0):1 T0):2 T0):3 (((T2 T3):4) 2):5):6

When the maxID= 2 the following stream is resulting:

2((((T1 T0):1 T0):2 T0) (((T2 T3):2) (1 T0):2))

The mapping of nodes to nodeIDs is visualized in Figure 4(b). It is not possible to store all nodes in the table. The third node cannot be stored and becomes a TempNode. For the forth node the orphan nodeID 2 *can be reused, because it is not referenced from other nodes. The next candidate to be reused would be nodeID* 1 *but this is referenced by the next node and cannot be reused.*

5.1. Algorithm

The apply algorithm for streaming MTBDDs is presented in Figure 5. This apply algorithm works similar to the apply for conventional DDs. The cofactors cannot be computed in the beginning, but have to be read from the stream every time an apply operation is called. If a calculated result has temporary nodes as children or if it is not already stored in the unique table, it has to be written into the stream immediately.

5.2. Estimation of Memory Usage

Different structures are necessary to implement this algorithm. For each input stream there has to be a table with *maxID* entries. For the output stream there are two structures to efficiently check wether a node is up-to-date and if it is unique in this context. One is implemented as a hashtable and the other as a regular table. A second hashtable is needed for the computed table.

In Table 1 the size of each entry and the stored information can be seen. Line s_{in} gives information for a node of the input stream, s_{comp} gives the information for an entry

Table 1. Sizes for Entries

	size	fields
s_{in}	20 Byte	*0-child ID*, *1-child ID*, *variable*, *timestamp*
$s_{computed}$	36 Byte	*nodeID1*, *nodeID2*, *result*, *timestamp* for each ID
$s_{out(t)}$	25 Byte	*0-child ID*, *1-child ID*, *variable*, *timestamp*, *reference_count*
$s_{out(h)}$	24 Byte	*0-child ID*, *1-child ID*, *nodeID*, *timestamp*

in the computed table, and $s_{out(t)}$ and $s_{out(h)}$ are the entries which are needed for the output stream.

The maximal amount of main memory needed can now be estimated as follows, where $\#comp$ denotes the maximum number of entries in the computed table:

$$\begin{aligned} mem\_usage \;=\; & (maxID_{in1} + maxID_{in2}) \cdot s_{in} \\ & + maxID_{out} \cdot (s_{out(t)} + s_{out(h)}) \\ & + \#comp \cdot s_{computed} \end{aligned}$$

Example 3. *Assuming each input stream and the output stream have a maxID* $= 250000$ *and #comp* $= 250000$*. The memory usage would be approximately 30 MB. The entire memory needs would be 30 MB plus a few MBs for internal structures to keep information during the recursive steps.*

6. Experimental Results

The experiments were carried out on a AMD Athlon XP 2200+ with 512 MB running at 1800 Mhz under Linux.

We built pairs of randomized polynomials with CUDD [15] and dumped them as streams with the *maxID* equal to the number of nodes occuring in the diagram. We did not limit the available memory, such that the functions *f* and *g* could be calculated completely. Afterwards we recalculated each stream with $maxID = 250000$, such that the memory limit is not exceeded while reading the input-streams. Then, the available memory was limited to 64MB with the linux command “ulimit”. Finally, $h = ADD(f, g)$ was calculated using CUDD and streaming MTBDD with limited memory.

Table 2 shows the results. Shown is information about the three function f, g and h. The subcolumns #nodes and #paths give the number of nodes and the number of paths of each MTBDD, respectively. For both approaches the memory usage, the CPU time in seconds and the elapsed time in seconds are given.

CUDD finished only 4 of the tests with the memory limit, while the new approach based on streaming MTBDD fin-

Table 2. Comparison between CUDD and streaming MTBDD

	f		g		h		CUDD			streaming MTBDD		
	#nodes	#paths	#nodes	#paths	#nodes	#paths	mem_used	time	el_time	mem_used	time	el_time
1	1964	31962	4617	212396	11649	408595	$>$ 64 MB			24052 kB	0.09 s	0.13 s
2	172565	$5.79 \cdot 10^6$	25199	$4.41 \cdot 10^6$	355522	$1.18 \cdot 10^7$	24636 kB	0.58 s	0.59 s	41496 kB	62.15 s	62.50 s
3	689	$1.87 \cdot 10^6$	31371	$1.11 \cdot 10^6$	34518	$2.91 \cdot 10^6$	10940 kB	0.01 s	0.01 s	25936 kB	0.53 s	0.58 s
4	263269	$2.04 \cdot 10^7$	466	$1.47 \cdot 10^7$	396120	$3.76 \cdot 10^7$	$>$ 64 MB			42488 kB	164.33 s	170.69 s
5	595332	$3.53 \cdot 10^6$	119282	$2.20 \cdot 10^7$	1665366	$2.98 \cdot 10^7$	$>$ 64 MB			43200 kB	1903.67 s	1969.65 s
6	717858	$8.20 \cdot 10^8$	636101	$6.93 \cdot 10^8$	6477925	$1.67 \cdot 10^9$	$>$ 64 MB			42968 kB	7666.66 s	7882.48 s
7	346	16016	484235	$9.93 \cdot 10^7$	586225	$9.93 \cdot 10^7$	$>$ 64 MB			42688 kB	180.69 s	181.78 s
8	34163	$6.57 \cdot 10^6$	17508	$4.06 \cdot 10^7$	825034	$1.07 \cdot 10^8$	$>$ 64 MB			41652 kB	572.83 s	602.87 s
9	573941	$2.66 \cdot 10^7$	1506774	$6.03 \cdot 10^7$	7413668	$1.35 \cdot 10^8$	$>$ 64 MB			43276 kB	21446.80 s	22018.00 s
10	53308	$1.32 \cdot 10^7$	1218920	$7.00 \cdot 10^7$	2169367	$9.20 \cdot 10^7$	$>$ 64 MB			42560 kB	4359.77 s	4375.44 s
11	194670	$1.60 \cdot 10^7$	577404	$1.03 \cdot 10^8$	2360824	$1.52 \cdot 10^8$	$>$ 64 MB			43164 kB	4498.52 s	4528.59 s
12	65507	$1.91 \cdot 10^7$	296	1789	88217	$2.13 \cdot 10^7$	54652 kB	0.04 s	0.04 s	32248 kB	2.77 s	2.88 s
13	106620	$1.25 \cdot 10^8$	330727	$6.74 \cdot 10^7$	822495	$2.33 \cdot 10^8$	45200 kB	2.18 s	2.23 s	42292 kB	1030.01 s	1033.45 s
14	63505	$5.31 \cdot 10^6$	2154057	$1.34 \cdot 10^9$	2520890	$1.30 \cdot 10^9$	$>$ 64 MB			42808 kB	3165.56 s	3175.95 s
15	1927569	$1.27 \cdot 10^8$	42921	$2.20 \cdot 10^7$	1994150	$1.50 \cdot 10^8$	$>$ 64 MB			42804 kB	2402.92 s	2411.18 s
16	1102157	$1.40 \cdot 10^7$	776349	$5.20 \cdot 10^7$	6791577	$1.40 \cdot 10^8$	$>$ 64 MB			43264 kB	21704.50 s	21873.80 s

ished all of them. As expected CUDD was faster when both methods finished, but on the other hand streaming MTBDD never used more than 43 MB of the memory.

7. Conclusions and Discussion

A technique to control the memory usage for operations on WLDDs has been presented. This technique is an extension of the streaming BDD technique that has been introduced in [14]. As a result trading run-time for memory during evaluation of DDs becomes possible. A detailed case study has been carried out for MTBDDs and the estimation of memory usage has been shown. Experiments show the efficiency of the approach.

References

[1] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their application. In *Int'l Conf. on CAD*, pages 188–191, 1993.

[2] B. Becker, R. Drechsler, and R. Enders. On the computational power of bit-level and word-level decision diagrams. In *ASP Design Automation Conf.*, pages 461–467, 1997.

[3] B. Becker, R. Drechsler, and R. Werchner. On the relation between BDDs and FDDs. *Information and Computation*, 123(2):185–197, 1995.

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

[5] R. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Design Automation Conf.*, pages 535–541, 1995.

[6] M. Ciesielski, P. Kalla, Z. Zheng, and B. Rouzyere. Taylor expansion diagrams: A compact canonical representation with applications to symbolic verification. In *Design, Automation and Test in Europe*, pages 285–289, 2002.

[7] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi terminal binary decision diagrams: An efficient data structure for matrix representation. In *Int'l Workshop on Logic Synth.*, pages P6a:1–15, 1993.

[8] E. Clarke and X. Zhao. Word level symbolic model checking – a new approach for verifying arithmetic circuits. Technical Report CMU-CS-95-161, 1995.

[9] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. Perkowski. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. In *Design Automation Conf.*, pages 415–419, 1994.

[10] R. Drechsler, M. Theobald, and B. Becker. Fast OFDD based minimization of fixed polarity Reed-Muller expressions. In *European Design Automation Conf.*, pages 2–7, 1994.

[11] P. Kalla, M. Ciesielski, E. Boutillon, and E. Martin. High-level design verification using taylor expansion diagrams: First results. pages 13–17, 2002.

[12] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams for logic synthesis and verification. *Multiple Valued Logic - An International Journal*, 1997.

[13] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *European Conf. on Design Automation*, pages 43–47, 1992.

[14] S. Minato and S. Ishihara. Streaming BDD manipulation for large-scale combinatorial problems. In *Design, Automation and Test in Europe*, pages 224–228, 2001.

[15] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.1*. University of Colorado at Boulder, 2001.