Symbolic Representation with Ordered Function Templates

Amit Goel
Department of Electrical & Computer Engineering
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA. 15213
agoel@ece.cmu.edu

Gagan Hasteer
Innologic Systems
50a Charcot Avenue
San Jose, CA. 95131
gagan@innologic-systems.com

Randal E. Bryant
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA. 15213
Randal.Bryant@cs.cmu.edu

ABSTRACT

Binary Decision Diagrams (BDDs) often fail to exploit sharing between Boolean functions that differ only in their support variables. In a memory circuit, for example, the functions for the different bits of a word differ only in the data bit while the address decoding part of the function is identical. We present a symbolic representation approach using ordered function templates to exploit such regularity.

Templates specify functionality without being bound to a specific set of variables. Functions are obtained by instantiating templates with a list of variables. We ensure canonicity of the representation by requiring that templates are normalized and argument lists are ordered. We also present algorithms for performing Boolean operations using this representation. Experiments with a prototype implementation built on top of CUDD indicate that function templates can dramatically reduce memory requirements for symbolic simulation of regular circuits.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*verification, simulation*

General Terms

Algorithms, Verification

Keywords

Boolean functions, function templates, binary decision diagrams, symbolic simulation, logic design verification

1. INTRODUCTION

Reduced Ordered Binary Decision Diagrams(BDDs) [4] are a graph representation for boolean functions. For many practical cases, the size of the BDD representation is quite compact, thus enabling the efficient representation and manipulation of boolean functions. However, in several cases, the size of the BDD representation is prohibitively large, e.g., multipliers have provably exponential BDD size [2]. In other instances, even though the sizes of the BDDs for individual functions are small, the combined size of all of the BDDs becomes too large to represent in memory.

Shared BDDs alleviate the second problem somewhat, by using a multi-rooted DAG to represent all the functions of interest. Edge attributes further reduce the size requirements. In [10], the authors proposed three edge-attributes to reduce the size of the shared DAG:

1. Output inversion or complemented edges. This attribute allows a function and it's complement to use the same graph representation.
2. Input inversion. This attribute has the effect of swapping the two children of a variable node. Thus, the functions $f = x \cdot g + \neg x \cdot h$ and $\tilde{f} = x \cdot h + \neg x \cdot g$ can be represented by the same BDD.
3. Variable Shifter. This attribute was motivated by the observation that functions such as $(v_1 + v_2.v_3)$ and $(v_2 + v_3.v_4)$ are isomorphic except for a difference of one in the indices of the input variables. This attribute allows such functions to be represented by the same BDD by keeping the difference as an edge attribute. This idea was generalized to Differential BDDs in [1].

The variable shifter attribute allows the same node to be labeled by multiple variables. Other representation schemes also transform the input space of functions to obtain multiple variable labelings on nodes. In Graph Driven BDDs [11] and Free BDDs [5], the input space is transformed by an oracle graph which gives possibly different variable orderings on different computation paths. In Linearly Transformed BDDs [6], the input vector is linearly transformed based on a transformation matrix, thus labeling each node with the parity of a set of variables.

In this paper, we describe a simple representation for compact representation of shared BDDs using ordered function templates. Our representation, described below, also uses a re-mapping of input variables, but unlike the above approaches, there is no global oracle or transformation matrix. Instead, we *compact* the set of input variables for each function independently.

Consider a memory with four rows and three columns. If we write the symbolic values d_1, d_2 and d_3 to the row address

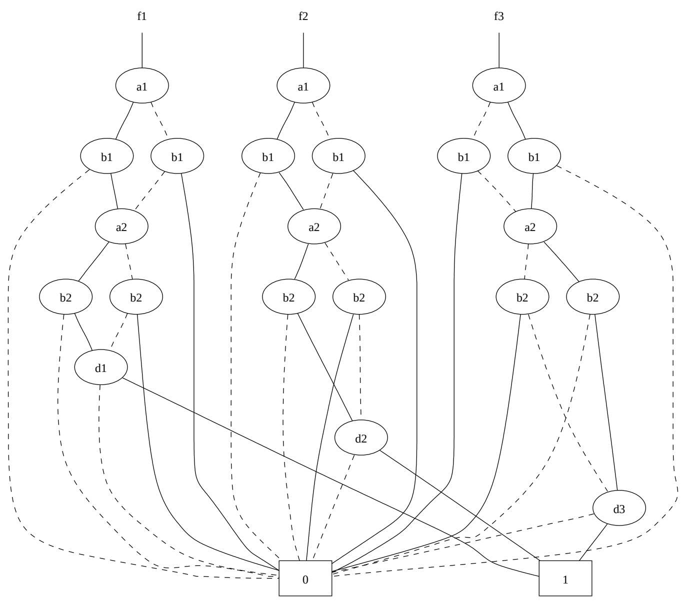

Figure 1: Output functions for a 4-row 3-column memory on reading address $[b_1, b_2]$ after writing data $[d_1, d_2, d_3]$ to address $[a_1, a_2]$.

$[a_1, a_2]$ in a memory initialized with all 0's and then perform a symbolic read at address $[b_1, b_2]$, the output functions for the three columns are $f_i = (a_1 \leftrightarrow b_1) \cdot (a_2 \leftrightarrow b_2) \cdot d_i$. The three functions differ only in the data variable d_i. Figure 1 shows the BDDs for the three output functions, with the variable order $[a_1, b_1, a_2, b_2, d_1, d_2, d_3]$ (address before data). Note that the variable-shifter attribute does not allow us to use the same BDD for all three functions, since the address bits are exactly the same in the three functions and only the data bit differs.

In the above example, if we let F stand for the function template $(x_1 \leftrightarrow x_2) \wedge (x_3 \leftrightarrow x_4) \wedge x_5$, then the three output functions can be obtained by the substitution $f_i = F[a_1/x_1, b_1/x_2, a_2/x_3, b_2/x_4, d_i/x_5]$. The template serves as a macro for functions we want to represent, which can be obtained by substituting real variables for the formal variables used in the template. Hence, a function can be represented by a tuple containing a function template and an argument list, e.g., $f_i = (F, [a_1, b_1, a_2, b_2, d_i])$, where the substitution is implicit.

In the above example, we use a different mapping of the input space for each function. In particular d_i maps to x_5 in the representation of f_i. This is in contrast to the global re-mapping schemes of [11, 5, 6].

We place restrictions on the representation of templates and argument lists to ensure canonicity of the representation – we will require that function templates are normalized and that the argument lists are ordered. With this restriction, the function template for a function is just a relabeled version of the function itself. Hence, not taking into account the overhead for the argument list, the size of the representation for a single function is exactly the same with or without the use of function templates. However, the size of the representation for multiple functions can decrease dramatically if, like in the example above, several functions use the same template. We further increase sharing by ensuring that smaller templates get used in constructing larger templates.

Our experiments indicate that the use of ordered function templates can give significant savings for regular circuit applications. In general, we expect the representation to be beneficial in applications where the sizes of individual BDDs are not large compared to the size of the shared representation. This is quite often the case in semi-formal verification using symbolic simulation.

While we describe how to use function templates with BDDs, the ideas are general enough that they can be used in conjunction with other underlying symbolic representations. In addition, we can extend the scheme to multi-terminal decision diagrams by using two argument lists, one for the variables and one for the terminal values.

The rest of the paper is organized as follows. In Section 2, we describe the ordered function template representation. Section 3 presents algorithms for performing Boolean operations on function represented with function templates. Section 4 describes our prototype implementation which was used to obtain the experimental results in Section 5. Our conclusions are presented in Section 6.

2. ORDERED FUNCTION TEMPLATES

As described in the Introduction, we want to represent any boolean function as a tuple of a function template and an argument list. In the rest of the paper, we will assume that function templates are defined over the variables $X = \{x_1, x_2, \ldots\}$ and the argument lists are defined over the variables $V = \{v_1, v_2, \ldots\}$. We will refer to the variables in X as formal variables and the variables in V as actual variables. Also, $l(i)$ refers to the i-th element of list l and $|l|$ is the number of elements in the list.

We restrict templates to a class of normalized functions over X to avoid multiple representations for the same function. Our normalization condition is that we use the smallest numbered variables. So, $x_1 + x_2$ is a normalized template whereas $x_1 + x_3$ and $x_4 + x_5$ are not normalized. To define a normalized function, we first define the support of a function F in the usual way.

Definition 1. The *support* of a function F over variables in X is given by $support(F) = \{x_i \mid F_{|x_i=0} \neq F_{|x_i=1}\}$.

The definition of the support of F corresponds to the set of variables used in the BDD representation of F. We now define a normalized function.

Definition 2. A function F over variables in X is *normalized* if and only if $\forall i > 1. x_i \in support(F) \Rightarrow x_{i-1} \in support(F)$. An *n-argument function template* is a normalized function F such that $|support(F)| = n$.

In order to make the representation canonical, we also require that argument lists be ordered with respect to some strict ordering $\prec$ on the actual variables V.

Definition 3. An *n-argument list* is an ordered list over V with n elements, i.e., $[v_{\pi(1)}, \ldots, v_{\pi(n)}]$ such that $v_{\pi(i)} \prec v_{\pi(i+1)}$, for all $1 \leq i < n$.

A tuple (F, l), where F is a function (not necessarily normalized) over X and l is a list of arguments (not necessarily variables from V), represents the function obtained by substituting the arguments in the list for the variables in X, i.e.,

$(F, [y_1, \ldots, y_n]) \equiv F[y_1/x_1, \ldots, y_n/x_n]$. We will not distinguish between the tuples and the functions they represent.

Definition 4. A *function instance* is a tuple (F, l) where F is an n-argument function template and l is an n-argument list for some n.

Thus a function instance is, essentially, a normalized tuple (F, l), i.e., F is a normalized function over X, l is an ordered list over V and the number of actual variables in l is the same as the number of formal variables in the support of F. These restrictions ensure canonicity of the representation.

Proposition 1. For $f = (F, l_f)$ and $g = (G, l_g)$ we have $f = g$ if and only if $F = G$ and $l_f = l_g$.

Proof outline. (If) Let $l_f = l_g = [v_{\pi(1)}, \ldots, v_{\pi(n)}]$. By definition we have $f = F[v_{\pi(1)}/x_1, \ldots, v_{\pi(n)}/x_n]$ and $g = G[v_{\pi(1)}/x_1, \ldots, v_{\pi(n)}/x_n]$. But $F = G$, giving us $f = g$.

(Only If) For any arbitrary n-length boolean vector $Y \in \mathcal{B}^n$, we have $f(Y) = g(Y)$. By applying the definitions and simplification, we get $f(Y) = F(Y)$ and $g(Y) = G(Y)$. Hence $F(Y) = G(Y)$ for arbitrary Y. Hence, $F = G$. From $f = g$, $F = G$ and the normalization condition on templates, it follows that $l_f = l_g$.

3. BOOLEAN OPERATIONS ON FUNCTION INSTANCES

We now describe how to perform operations on function instances. Given function instances $f = (F, l_f)$, $g = (G, l_g)$ and a binary operation $\odot$, we want to compute $h = (H, l_h)$ such that $h = f \odot g$. In order to do so, we first denormalize the function templates to ensure that formal variables correspond to the same actual variables in both operands, based on the expanded domain (the merged list of arguments). Then we can perform the binary operation on the denormalized functions. Finally, we eliminate unnecessary variables from the argument list and normalize the template.

The algorithm for applying a binary operator $\odot$ is shown in Algorithm 1. The algorithm denormalizes the templates (lines 2,3) to account for domain expansion (line 1), using Algorithm 2. It then computes an intermediate result by applying $\odot$ (line 4) to the denormalized templates. This intermediate result is not necessarily normalized because of possible domain contraction. Finally, it normalizes the result (line 5) using Algorithm 3.

Consider computing $h = f + g$, where $f = v_1 \cdot v_2 + v_3$ and $g = v_1 \cdot \neg v_2 + v_4$ represented by $(x_1 \cdot x_2 + x_3, [v_1, v_2, v_3])$ and $(x_1 \cdot \neg x_2 + x_3, [v_1, v_2, v_4])$ respectively.

In our example, the merged list is $\tilde{l}_h = [v_1, v_2, v_3, v_4]$. Based on this, we denormalize the template for g to $\tilde{G} = x_1 \cdot \neg x_2 + x_4$, by substituting x_4 for x_3 in G, since the real variable corresponding to x_3 in g is v_4 which is fourth in the merged list. The template for f remains unchanged, i.e., $\tilde{F} = x_1 \cdot x_2 + x_3$. The disjunction of the modified templates gives us $\tilde{H} = \tilde{F} + \tilde{G} = x_1 + x_3 + x_4$ which does not have x_2 in its support. Hence, we eliminate v_2, which corresponds to x_2, from the merged argument list to get the result list $l_h = [v_1, v_3, v_4]$, and normalize $\tilde{H} = x_1 + x_3 + x_4$ to get the result template $H = x_1 + x_2 + x_3$. The final computed tuple is $(H, l_h) = (x_1 + x_2 + x_3, [v_1, v_3, v_4])$ which is the desired result.

Algorithm 1 Apply $\odot$

```
Require: Function Instances $(F, l_f)$ and $(G, l_g)$
Ensure: $(H, l_h) = (F, l_f) \odot (G, l_g)$
 1: $\tilde{l}_h \leftarrow$ merge$(l_f, l_g)$
 2: $\tilde{F} \leftarrow$ denormalize$(F, l_f, \tilde{l}_h)$
 3: $\tilde{G} \leftarrow$ denormalize$(G, l_g, \tilde{l}_h)$
 4: $\tilde{H} \leftarrow \tilde{F} \odot \tilde{G}$
 5: $(H, l_h) \leftarrow$ normalize$(\tilde{H}, \tilde{l}_h)$
 6: return $(H, l_h)$
```

Algorithm 2 Denormalize

```
Require: Function Instance $(F, l_f)$ and expanded list $\tilde{l}_h$
Ensure: $(\tilde{F}, \tilde{l}_h) = (F, l_f)$
 1: $n \leftarrow |l_f|$
 2: $i \leftarrow 1$
 3: $j \leftarrow 1$
 4: while $i \leq n$ do
 5:   if $l_f(i) = \tilde{l}_h(j)$ then
 6:     $\tilde{x}_i \leftarrow x_j$
 7:     $i \leftarrow i + 1$
 8:   end if
 9:   $j \leftarrow j + 1$
10: end while
11: $\tilde{F} \leftarrow F[\tilde{x}_1/x_1, \ldots, \tilde{x}_n/x_n]$
12: return $\tilde{F}$
```

The denormalization and normalization represent the computational overhead in this scheme. In particular, the overhead comes largely from the substitution operations in line 11 of Algorithm 2 and line 13 of Algorithm 3 and the support computation in line 1 of Algorithm 3. Note that the denormalization and normalization could be done on demand in much the same way as BDD reduction. Our prototype implementation described below, however, is built as a set of wrappers around the CUDD package [12] using the algorithms shown.

4. IMPLEMENTATION

It is a well known fact that the size of the BDD representation for a function can change dramatically based on the

Algorithm 3 Normalize

```
Require: Tuple $(\tilde{F}, \tilde{l}_f)$
Ensure: $(F, l_f)$ is a function instance and $(F, l_f) = (\tilde{F}, \tilde{l}_f)$
 1: $S \leftarrow$ support$(\tilde{F})$
 2: $n \leftarrow |S|$
 3: $i \leftarrow 1$
 4: $j \leftarrow 1$
 5: while $i \leq n$ do
 6:   if $x_j \in S$ then
 7:     $l_f(i) \leftarrow \tilde{l}_f(j)$
 8:     $\tilde{x}_i \leftarrow x_j$
 9:     $i \leftarrow i + 1$
10:   end if
11:   $j \leftarrow j + 1$
12: end while
13: $F \leftarrow F[x_1/\tilde{x}_1, \ldots, x_n/\tilde{x}_n]$
14: return $(F, l_f)$
```

Circuit Information		BDD		Ordered Function Templates				
Name	Steps	Peak(K)	Time(s)	PeakTemp(K)	PeakArg(K)	TimeTot(s)	TimeOvhd	TimeSupp(s)
Cam-rtl	3	206	16	38	51	890	823	326
Cam-xtor	3	276	10	80	64	1370	1184	545
Sram-xtor	3	794	120	49	2	43	38	20
Reg-rtl	3	32375	2148	354	223	2911	2843	1118
Reg-xtor	3	9897	802	385	11	1310	1096	839
Fifo-rtl	16	3511	90	535	1	70	49	40
Snp-rtl	6	8241	27	1039	8	119	103	52
Ctrl-rtl	12	782	14	938	1	56	43	25
Mis-rtl	3	Memout	-	2942	4	220	189	124

Table 1: Symbolic Simulation with BDDs and Ordered Function Templates

BDD variable ordering used. In our representation, we have two orders: the BDD variable ordering used to represent the function templates and the ordering on the actual variables for argument lists. The effect corresponding to traditional BDD re-ordering can be obtained by re-ordering either of the two orders in our setup. If we keep the argument order fixed, the BDD variable order for formal variables behaves in the standard way. On the other hand, we can keep the BDD variable order fixed and re-order the arguments, in which case the function template used for a particular instance changes and gives us the same effect. We choose the second option, since it allows us to further increase sharing between different functions as explained below.

We chose a fixed variable ordering on X such that $x_{i+1} < x_i$. The reason for choosing this ordering is that we can increase sharing since larger templates (templates with more arguments) can reuse the smaller templates. Consider the Shannon decomposition of an n-argument template F with respect to x_n given by $F = x_n \cdot F_{|x_n=1} + \neg x_n \cdot F_{|x_n=0}$. It is likely that the co-factors of F with respect to x_n are normalized, and hence they would re-use smaller templates. On the other hand, if we placed x_1 at the top, then we get reduced sharing because the co-factors of F with respect to x_1 are definitely not normalized.

The argument lists are implemented as BDD cubes, which gives an easy merge procedure: the ordered merged list corresponding to l_f and l_g is given by the conjunction $l_f \cdot l_g$. We chose to implement the argument list cubes and the function templates in different BDD managers since we want to keep a fixed variable ordering for templates whereas we would like to re-order the argument lists. The re-ordering procedure will differ from traditional BDD variable re-ordering, though, since the success of the re-ordering on the cubes will be measured by its effect (in terms of increased sharing) on the templates.

Consider reordering the real variables by swapping the adjacent variables $v_{\pi(i)}$ and $v_{\pi(i+1)}$. Any function instance which does not have both $v_{\pi(i)}$ and $v_{\pi(i+1)}$ in its argument list remains unchanged. If both are used in $f = (F, l_f)$, then the representation for f after reordering is given by $(\tilde{F}, \tilde{l}_f)$ where $\tilde{F} = F[x_i/x_{i+1}, x_{i+1}/x_i]$ and $\tilde{l}_f(i) = v_{\pi(i+1)}, \tilde{l}_f(i+1) = v_{\pi(i)}$ and $\tilde{l}_f(p) = l_f(p), \forall p \notin \{i, i+1\}$. The basic swap operation can be used to build reordering strategies similar to those used for BDDs [9].

5. EXPERIMENTAL RESULTS

We used our implementation of ordered function templates to perform symbolic simulation of several memory related industrial circuits. The results are shown in Table 1.

The suffix **rtl** indicates that the design is at the register-transfer level while the **xtor** suffix indicates a transistor level design. The designs were simulated symbolically using all symbolic inputs, for the number of simulation steps shown in the Steps column. For the experiments with BDDs using CUDD, Peak indicates the peak live nodes in thousands and Time indicates the time taken in seconds for the symbolic simulation. For the experiments with Ordered Function Templates, PeakTemp indicates the peak live nodes used for the templates and PeakArg for the argument lists (cubes). TimeTot indicates the total time taken and TimeOvhd indicates the time taken for normalize and denormalize operations, including TimeSupp, the time taken for support computation (line 1 of Algorithm 3).

Ordered Function Templates give significant savings in peak live nodes for most of the data-path circuits (the first six rows). This is mainly due to extensive reuse of the templates.

Table 2 shows the distribution of output functions at the end of the symbolic simulation for Reg-rtl, e.g., there are two templates with support size 222. These two templates are used in 774 unique function instances. Note that only one template with one variable is ever needed, since any variable v_i is represented as $(x_1, [v_i])$.

Reg-rtl is a 64 column multiport register file with three read and three write ports with 16 bit addresses. The symbolic simulation is performed using ternary values [3] starting from a completely general state with all memory elements set to X. For this circuit, the behavior is very much like that for the example in the Introduction, except that we have two possible templates for each column because of the ternary simulation. The function instances with 194, 195 and 222 variables represent the functions obtained for

Variables	1	2	194	195	222
Templates	1	1	2	2	2
Functions	1098	52	258	774	774

Table 2: Distribution of Unique Templates and Functions for Reg-rtl at the end of symbolic simulation. The table shows the number of unique Function Templates and Functions used in the representation of the output functions and state of the circuit against the number of variables in their support.

Variables	1	38	40	44	45	46	47	49	52	54	55
Templates	1	4	2	2	3	2	1	8	3	3	8
Functions	120	4	2	2	3	2	1	8	3	3	8

Table 3: Distribution of unique Function Templates and Functions for Ctrl-rtl.

the state of the memory, the output at the three ports and the verification conditions in the symbolic testbenches. For each case we have just two templates, which are reused for each column on each read port. The gain is amplified by the fact that the supports of these functions are really large since the address decoding part of the functions includes addresses from the multiple ports on multiple write cycles. In addition, the support includes variables used for hierarchical compression of the circuit [8, 7]. With function templates, the ternary values on all 64 columns are represented by the same two templates, giving a dramatic reduction in memory requirements.

With Ctrl-rtl, which is a controller for a DRAM, the ordered functional template has a higher peak live node count than for BDDs. As Table 3 shows, in this case there is no external template reuse since the number of templates is equal to the number of function instances. In addition, since the distribution contains only functions with a large support size, there are no smaller templates for internal reuse either. The increase in peak live nodes occurs because of the creation of denormalized BDDs while performing Boolean operations.

In Snp-rtl (a priority encoder for an SRAM), too, there is no external reuse of templates. However, the distribution includes function instances over a wide range of support sizes. Hence, the internal reuse of smaller templates for larger templates results in almost an 8 times lower peak node count for templates.

For all circuits, except Mis-rtl, we used *good* variable orderings available from a regression suite. For Mis-rtl, which is an SRAM, we used the default variable order obtained from the input file. In this case, the symbolic simulation with BDDs does not complete with 2GB of memory, while with Ordered Function Templates we are able to complete the simulation, well within the memory limit.

For most cases, the Ordered Function Templates run is much slower than the BDD run. However, this is mainly because of the overheads incurred from normalization and de-normalization. In Sram-xtor and Fifo-rtl, the runtime is better than that for BDDs since the increased sharing enhances the effect of the computed cache. In these cases, the effect of the computed cache more than makes up for the overheads incurred. A clean implementation performing normalization and denormalization on the fly should reduce the run time significantly.

In summary, we think function templates can give significant memory savings in regular circuit applications where only a few templates are needed to represent the majority of functions of interest. In addition to memories, we expect them to perform well for data-flow circuits. We do not expect function templates to be useful for control circuits.

Another potential application is in verification with cut-points where the new variables introduced at the cut-points can cause problems with BDDs since all the functions created beyond the cut-point differ in their support from the functions up to the cut-point. With function templates, the two sets of functions would get mapped to the same formal variable space.

6. CONCLUSIONS

We have presented a representation for Boolean functions using normalized function templates and ordered argument lists and shown how to perform operations using this representation. We have built a prototype implementation as a set of wrappers around the CUDD package and used it to perform symbolic simulation of several memory related industrial circuits. Our experiments indicate that for regular circuits, the new representation can be effective in reducing the memory requirement.

Our current implementation is much slower than BDDs in most cases. However, the runtime performance for Ordered Function Templates can be improved by a better implementation which tightly integrates the normalization and denormalization with the apply procedure.

7. REFERENCES

[1] Anuchit Anuchitanukul, Zohar Manna, and Tomas E. Uribe. Differential BDDs. In *Computer Science Today*, pages 218–233. 1995.

[2] R. E. Bryant. On the compexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[3] Randal E. Bryant, Derek L. Beatty, and Carl-Johan H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proceedings of the 28th Design Automation Conference (DAC'91)*, pages 397–402, 1991.

[4] Randal.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] J. Gergov and C. Meinel. Boolean manipulation with free BDDs: An application in combinational logic verification. In Bjørn Pehrson and Imre Simon, editors, *Proceedings of the IFIP 13th World Computer Congress. Volume 1 : Technology and Foundations*, pages 309–314, Amsterdam, The Netherlands, August 28–September 1 1994. Elsevier Science Publishers.

[6] Wolfgang Günther and Rolf Drechsler. BDD minimization by linear transformations. In *Advanced Computer Systems*, pages 525–532, 1998.

[7] Innologic Systems Inc. http://www.innologic-systems.com.

[8] Alfred Koelbl, James Kukula, and Robert Damiano. Symbolic RTL simulation. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 47–50, 2001.

[9] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *IEEE /ACM*

International Conference on CAD, pages 42–47, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.

[10] S. Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 52–57, Los Alamitos, CA, June 1990. ACM/IEEE, IEEE Society Press.

[11] Detlef Sieling and Ingo Wegener. Graph driven BDDs—a new data structure for Boolean functions. *Theoretical Computer Science*, 141(1–2):283–310, 17 April 1995.

[12] F. Somenzi. CUDD: CU Decision Diagram Package Release, 1998.