Minimization of Multiway Decision Graphs for RTL Verification by Stochastic Optimization

YI FENG and XIAOYU SONG[1]
Department of Computer Science
Algoma University, Ontario, Canada P6A 2G4
[1]Department of ECE, Portland State University, Portland, OR 97207, USA
feng@algomau.ca

Abstract: - The complexity of digital hardware designs has increased substantially with new advancements in electronic designs. It is becoming increasingly difficult to verify their correctness. MDGs have been proved to be a very effective tool in automatic hardware verification of RTL designs. In this paper, a method of variable reordering for Multiway Decision Graphs (MDG) is discussed. We present an automatic dynamic variable ordering algorithm for MDGs to reduce the effects of the state explosion problem. By contrast to ROBDDs, in MDGs difficulties are created by the presence of first order terms. The method we present here merges the benefits of stochastic evolution and sifting. It will be utilized to minimize the MDG size throughout the verification procedure. The effectiveness of our method is demonstrated by the empirical results provided.

Key-Words: - Formal Verification, Multiway Decision Graphs, Variable Ordering, Stochastic Evolution

1 Introduction

Electronic designs are becoming increasingly ubiquitous and pervasive in nature, and as they do, they begin to find their way in to more critical and important applications such as medical diagnostic equipment, life support systems, hazardous industrial process control, and other such applications where bugs can create catastrophes or may even threaten human lives. As a result, it becomes important to develop systems which ensure bug-free designs we produce and use. The task of validating a new design is conventionally performed by the use of simulation tools.

Formal verification is playing an increasing role in contrast to traditional simulation. Formal verification methods intend to determine whether or not a particular implementation satisfies a specification by the use of mathematical reasoning [4]. Multiway Decision Graphs (MDGs) offer a suitable representation of Extended Finite State Machines. As such, they provide a useful tool for the automated formal hardware verification of Register Transfer Level (RTL) designs.

MDGs efficiently represent a class of formulas of a many-sorted first-order logic with a distinction of abstract and concrete sorts [1]. In an MDG, a data signal is represented by a single variable of abstract sort rather than by a vector of Boolean variables, and a data operation is represented by an uninterpreted function symbol. MDGs compactly encode sets of (abstract) states and transition/output relations for abstract description of state machines. The implicit enumeration technique is lifted from the Boolean level to the abstract level and referred to as implicit abstract enumeration. MDGs are thus much more compact than ROBDDs for circuits having complex and large datapaths. This increases the range of circuits that can be verified.

MDG-based verification still suffers from the problem of state explosion when handling realistic circuits. Of approaches designed to reduce the effects of state explosion, the most significant approach seeks to select variable order that minimizes the problem. As with ROBDD, an MDGs size is closely tied to the order of its variables. This paper discusses a dynamic ordering algorithm for MDGs. Algorithms for variable ordering from ROBDDs can not be used directly on MDGs since first-order terms may be present in an MDG.

Our method determines a new order during the verification process to reduce the size of MDGs by permuting the variables of a given MDG starting with an initial static order. The method is based on the sifting algorithm [9] by swapping two adjacent variables. The optimal position for each variable is determined by the sifting process. Although the method is very effective in reducing the MDG size, it is computationally intensive. In our discussion, we use sifting in a stochastic evolutionary approach.

2 Multiway Decision Graphs

The formal logic underlying MDGs is a many-sorted first-order logic, augmented with the distinction between abstract sorts and concrete sorts [1]. This distinction is motivated by the natural division of datapath and control circuitry in RTL designs.

Concrete sorts have enumerations that are sets of individual constants, while abstract sorts do not. Variables of concrete sorts are used for representing control signals, and variables of abstract sorts are used for representing datapath signals. Data operations are represented by uninterpreted function symbols. An n-ary function symbol has a type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, where $\alpha_1 \ldots \alpha_{n+1}$ are sorts.

The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let *f* be a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort then f is an abstract function symbol. If all the $\alpha_1 \ldots \alpha_{n+1}$ are concrete, *f* is a concrete function symbol. If α_{n+1} is concrete while at least one of $\alpha_1 \ldots \alpha_n$ is abstract, then we refer to *f* as a cross-operator. While abstract function symbols are used to denote data operations, cross-operators are useful for modeling feedback signals from the datapath to the control circuitry. Both abstract function symbols and cross-operators are uninterpreted, i.e., their intended interpretation is not specified.

A multiway decision graph (MDG) is a finite directed acyclic graph G where the leaf nodes are labeled by formulas, the internal nodes are labeled by terms, and the edges issuing from an internal node N are labeled by terms of the same sort as the label of N. Such a graph represents a formula defined inductively as follows: (i) if G consists of a single leaf node labeled by a formula P, then G represents P; (ii) if G has a root node labeled A with edges labeled $B_1 \ldots B_n$ leading to subgraphs $G_1' \ldots G_n'$, and if each G_i' represents a formula P_i, then G represents the formula $\vee_{1 \le i \le n}((A = B_i) \wedge P_i)$.

We refer to an occurrence of a variable in a term that labels an edge or in a cross-term that labels a node as a secondary occurrence, while an occurrence of a variable as the label of a node is a primary occurrence. The primary variables (resp. secondary variables) of a graph G are those that have primary (resp. secondary) occurrences in G.

Just as Bryant's ROBDD must be reduced and ordered, MDG must also be reduced and ordered. The concept of ordering in MDG concerns two orders: the standard term order and the custom symbol order. The standard term order is a lexicographical order of all the terms of the logic. The custom symbol order is a total order of a set of symbols that includes the concrete variables, abstract variables, and some but not necessarily all of the operators. It is selected specifically for each model, like in ROBDD. The custom symbol order is independent of the standard term order. In this paper, we only discuss custom symbol ordering.

MDG must obey a set of conditions to keep it well-formed [1]. Three constraints on ordering are introduced by these conditions:

1. If an abstract variable *a* appears as a secondary variable in an edge label of node *b*, then $a < b$.
2. If a variable *a* appears as a secondary variable in a cross-term having cross-operator *f*, then $a < f$.
3. The present and next state variables must be in a corresponding order.

3 Variable swapping in Multiway Decision Graphs

[12] presents a method for automatically generating a static variable order based on some heuristic rules. We will describe how to reduce the size of MDGs by reordering variables.

Dynamic reordering improves the variable order by a series of swaps of adjacent variables. The basic operation in reordering is thus that of variable swapping. We begin with an introduction of the implementation of the swap operation in MDG, then continue with a discussion of the effects of the swap operation on the variable order and the constraints imposed on variable swapping.

3.1 Implementation of a variable swapping Operation

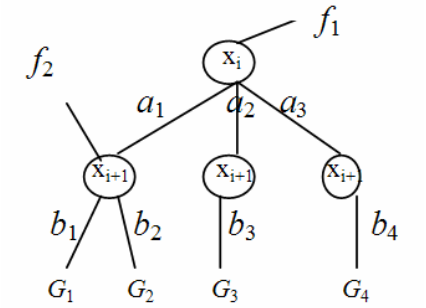
Variable swapping involves moving all MDG nodes at level *i* to level *i*+1 and nodes at level *i*+1 to level *i*. Level 0 is the root node of an MDG. All nodes at a level are labeled by the same variable. The level itself is also labeled by the same variable. The variables that label the levels must appear in a custom order. The level that a variable labels does not reflect its exact position in the order since secondary variables do not label the levels in the particular graph, but may in other MDGs that describe the design. All of them follow the same global order

Because an MDG node can be labeled by an abstract variable, a concrete variable or a cross-term, a variable swapping operation is of 3 kinds: a swap between two abstract variables, a swap between two concrete variables (or cross-terms) and a swap between a concrete variable (or cross-term) and an abstract variable.

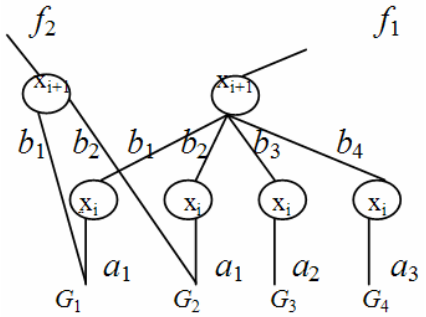
We first present swap between two abstract variables. Suppose we wish to swap abstract variables x_i and x_{i+1} in an MDG shown in Figure 1. The depicted MDG represents the following Direct Formulas (DFs) f_1 and f_2:

$$f_1 = ((x_i = a_1) \wedge (((x_{i+1} = b_1) \wedge G_1) \vee ((x_{i+1} = b_2) \wedge G_2))) \vee ((x_i = a_2) \wedge ((x_{i+1} = b_3) \wedge G_3)) \vee ((x_i = a_3) \wedge ((x_{i+1} = b_4) \wedge G_4))$$
$$f_2 = ((x_{i+1} = b_1) \wedge G_1) \vee ((x_{i+1} = b_2) \wedge G_2)) \quad (1)$$

Figure 1.(b) shows the MDG after swapping variables x_i and x_{i+1}. This MDG represents formula (3.2), which is equivalent to (1). DF f_2 remains undisturbed.



(a) Before swapping



(b) After swapping

Figure 1. Variable swap between two abstract variables

In first order logic, for any two formulas P and Q, there are the following axioms:

$$P \wedge Q \Leftrightarrow Q \wedge P,$$
$$P \wedge (Q \wedge R) \Leftrightarrow (P \wedge Q) \wedge R,$$
and $$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge Q).$$

Thus, we can deduce the following formulas from (1):

$$f_1 \Leftrightarrow (((x_i = a_1) \wedge ((x_{i+1} = b_1) \wedge G_1)) \vee ((x_i = a_1) \wedge ((x_{i+1} = b_2) \wedge G_2))) \vee ((x_{i+1} = b_3) \wedge ((x_i = a_2) \wedge G_3)) \vee ((x_{i+1} = b_4) \wedge ((x_i = a_3) \wedge G_4)))$$
$$\Leftrightarrow ((x_{i+1} = b_1) \wedge ((x_i = a_1) \wedge G_1)) \vee ((x_{i+1} = b_2) \wedge ((x_i = a_1) \wedge G_2)) \vee ((x_{i+1} = b_3) \wedge ((x_i = a_2) \wedge G_3)) \vee ((x_{i+1} = b_4) \wedge ((x_i = a_3) \wedge G_4))) \quad (2)$$

Note that in Figure 1, each edge issuing from nodes in level i must reach a node labeled by x_{i+1} before swapping and must reach a node labeled by x_i after swapping. The MDG well-formed conditions require that the set of abstract variables having primary occurrences along a path be the same for all paths in a given graph [1]. Thus, no edge issuing from nodes in level i reaches subgraphs in level i+2 directly. However, when swapping an abstract variable and a concrete variable (or cross-term) or two concrete variables (or cross-terms), there may exist an edge or edges issuing from the nodes in level i to the subgraphs in level i+2 directly.

Figure 2 shows variable swapping between an abstract variable and a concrete variable. x_i is of an abstract sort. x_{i+1} is of a concrete sort of enumeration $\{c_1, c_2, c_3, c_4\}$. Edge a_3 is issued from u (a node in level i) to subgraph G_4 directly without passing through level i+1. The MDG in Figure 2(a) represents the formula:

$$f_1 \Leftrightarrow ((x_i = a_1) \wedge (((x_{i+1} = c_1) \wedge G_1) \vee ((x_{i+1} = c_2) \wedge G_2))) \vee ((x_i = a_2) \wedge ((x_{i+1} = c_3) \wedge G_3)) \vee ((x_i = a_3) \wedge G_4) \quad (3)$$

Because $((x_{i+1} = c_1) \vee (x_{i+1} = c_2) \vee (x_{i+1} = c_3) \vee (x_{i+1} = c_4)) \equiv \mathbf{T}$, we can deduce the following equivalent formulas from formula (3) :

$$f_1 \Leftrightarrow ((x_i = a_1) \wedge (((x_{i+1} = c_1) \wedge G_1) \vee ((x_{i+1} = c_2) \wedge G_2))) \vee ((x_i = a_2) \wedge ((x_{i+1} = c_3) \wedge G_3)) \vee ((x_i = a_3) \wedge (((x_{i+1} = c_1) \wedge G_4) \vee ((x_{i+1} = c_2) \wedge G_4) \vee ((x_{i+1} = c_3) \wedge G_4) \vee ((x_{i+1} = c_4) \wedge G_4))) \quad (4)$$
$$\Leftrightarrow ((x_{i+1} = c_1) \wedge ((x_i = a_1) \wedge G_1)) \vee ((x_{i+1} = c_2) \wedge ((x_i = a_1) \wedge G_2)) \vee ((x_{i+1} = c_3) \wedge ((x_i = a_2) \wedge G_3)) \vee ((x_{i+1} = c_1) \wedge ((x_i = a_3) \wedge G_4)) \vee ((x_{i+1} = c_1) \wedge ((x_i = a_3) \wedge G_4)) \vee ((x_{i+1} = c_1) \wedge ((x_i = a_3) \wedge G_4)) \vee ((x_{i+1} = c_1) \wedge ((x_i = a_3) \wedge G_4))$$
$$\Leftrightarrow ((x_{i+1} = c_1) \wedge (((x_i = a_1) \wedge G_1) \vee ((x_i = a_3) \wedge G_4))) \vee ((x_{i+1} = c_2) \wedge (((x_i = a_1) \wedge G_2) \vee ((x_i = a_3) \wedge G_4))) \vee ((x_{i+1} = c_3) \wedge (((x_i = a_2) \wedge G_1) \vee$$

$$((x_i = a_3) \wedge G_4))) \vee$$
$$((x_{i+1} = c_4) \wedge (((x_i = a_3) \wedge G_1) \vee$$
$$((x_i = a_3) \wedge G_4))) \qquad (5)$$

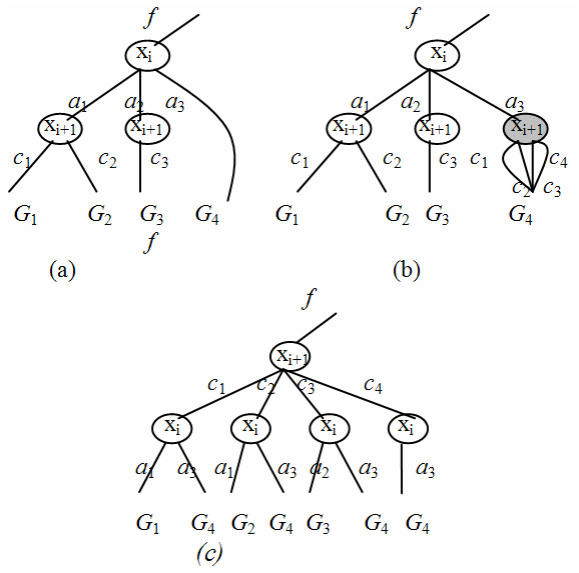


Figure 2 Variable swapping between an abstract variable and a concrete variable

A pseudo node w is added in Figure 2(b), which represents (4). Formula (4) can be further reduced to (5) where variables x_i and x_{i+1} are swapped. The MDG for (5) is shown in Figure 2(c).

Variable swaps between concrete variables (cross-terms) or a concrete variable (cross-term) and an abstract variable can be implemented in a similar way.

The variable swapping operation provides the basis for the reordering algorithm. It is easily seen that the swapping operation is completely local since only the nodes at level i and level i+1 need to be traversed.

3.2 The effects of the swapping operation on variable order

In an MDG, swapping two adjacent variables does not imply that they just exchange the positions in the ordering list. For example, suppose we apply two swap operations to the abstract variable x_3 in the MDG shown in Figure 5.4. The MDG is under the order $x_1 < a < b < x_2 < x_3$. x_1 is a Boolean variable and x_2, x_3 are abstract variables. a and b are secondary variables. There are four constraints in the MDG: $a < x_2$, $a < x_3$, $b < x_2$, and $b < x_3$.

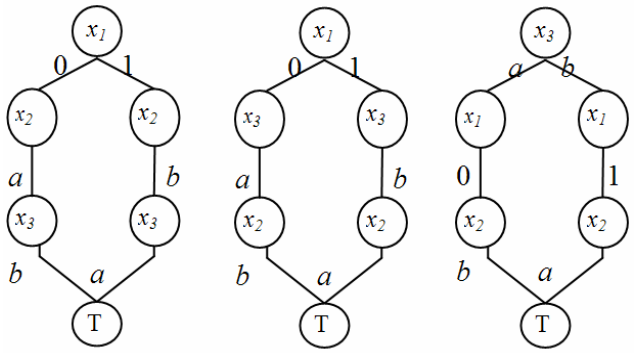


Figure 3 Effects of swap operation on the order

First, swapping x_2 and x_3 results in a new order $x_1 < a < b < x_3 < x_2$. Second, after swapping x_3 and x_1, we can not just swap their positions because of the constraints imposed between a, b and x_3. A new order $a < b < x_3 < x_1 < x_2$ is obtained by moving a, b with x_3 before x_1.

4 A Dynamic reordering algorithm in a stochastic evolution approach

The sifting algorithm presented in [10] has been proved to be successful at reordering BDDs and MDGs. During a sifting process, each primary variable in an MDG is examined in turn and is moved up and down in the order so as to take all positions successively. The variable is then returned to the position where the minimum size of the MDG was obtained. The process then continues with another variable. However, these operations tend to be time consuming.

We implemented sifting in a stochastic evolution approach to reduce unnecessary swaps. Stochastic evolution (SE), a technique dedicated to combinatorial optimization, was proposed by Saad and Rao in [13].

The basic idea of SE is to seek a global minimum of a cost function defined over a discrete domain D, called state space. Each state S is a mapping of a set of movable elements M into a set of locations L. A new state S' is generated by moving some elements in S. A move m can be simple or composed and must generate a unique new state S'. The gain of a move m is: gain(m) = cost(S) - cost(S'). Each move is accepted if the gain > random($-p$), where p is the parameter that allows negative gains in order to perform hill climbing.

The MDG ordering problem can be modeled as a permutation problem which can be solved using SE.

We choose $M=\{x_0, .., x_{n-1}\}$, the set of primary variables, and $L = \{1,..., n\}$. The state space is the set of all permutations of M. A state is a permutation of the primary variables. A move in the state space is generated by swapping two consecutive variables. The algorithm is described in Figure 4. The cost function **cost**(S) is the size of the MDG. S_0 is the static order generated by the algorithm proposed in [12]. p_0 was determined experimentally. The generation and acceptation of new states are done by the function perturb_SE. The update function modifies the control parameter p.

When sifting up/down a primary variable, we call function perturb-SE. In this function, we sift up/down a variable. After each swap, we check the new size of MDGs. If the size is smaller than the original MDG (positive gain), we will continue swapping this variable with the variable at the next position. If the size is greater than the original size (negative gain) and the negative gain is between 0 and $-p$, we will continue. Otherwise, sifting this variable will stop.

It is possible that the function perturb_SE always generates states that oscillate around a local minimum. To circumvent this problem we update p after each function call of perturb_SE. We will increase p by 5% if the cost turns small. We also set a possible best gain (threshold) that might be reached by our algorithm. Once that threshold has been reached, the whole reordering method will stop. The value of threshold will be decided by the number of primary variables and components of the circuit to be verified.

```
Algorithm  ORDERING_SE :

  S = S0       // S contains initial order
  SBest = S0    // save initial order
  p = p0      // initial control parameter

  for each variable xi in an MDG
    Cpre = cost(S);
    S = perturb _SE(S, p);
    Ccur = cost (S);
    update (p, Cpre, Ccur);

    if (cost(S) < cost (Sbest)) then
       Sbest = S;

  return (Sbest);
end;
```

Figure 4. ORDERING_SE algorithm.

5 Experimental results

We implemented the above algorithm in Prolog and integrated it into the MDG package [1]. We present experimental results on a number of IFIP benchmark circuits [5], an ATM switch fabric [2]. The experiments were carried out on a Sun Ultra 10 workstation with 333MHz and 1GB of memory. Both the experimental results in Table 1 and Table 2 demonstrate that the reordering algorithm using SE generated better results than the static ones and sifting in most cases. By comparing the results, we have seen that the basic sifting and our reordering algorithm can reduce the size of MDGs for most of benchmarks we experimented with. Compared to the basic sifting algorithm, our reordering algorithm results in about 52% time improvement and costs less than 1% in size increase on the average. Although reordering increases the computing time as compared with static ordering only, it can greatly reduce the size of MDGs.

6 Conclusions

In this paper, we discussed a new MDG minimization method using a genetic algorithm: stochastic evolution. We combined stochastic evolution with a powerful reordering method sifting. The experiments on IPIP benchmarks and an ATM switch fabric demonstrated the efficiency of the algorithms in reducing the size of MDG.

Table 1. Experimental results for IFIP benchmark circuits

Circuit	static		sifting		SE reordering	
	nodes	time(s)	nodes	time(s)	nodes	times(s)
minmax	166	0.2	164	1.2	164	0.5
tlc	245	1.2	245	11.0	248	5.6
gcd	375	0.3	369	4.3	369	3.1
tama	2211	1.4	2136	17.3	2152	6.8
filter	2957	4.1	2483	30.4	2485	14.2
queue	6551	3.6	6432	24.8	6433	12.8
buffer	2307	2.21	2307	33.2	2335	10.4

Table 2. Experimental results for property checking on an ATM switch fabric

Property	static		sifting		SE reordering	
	nodes (k)	time (s)	nodes (k)	time (s)	nodes (k)	time (s)
P1	19.5	42.4	12.1	640.1	12.4	300.1
P2	19.6	40.7	11.6	646.9	12.1	321.4
P3	22.2	52.1	9.6	833.4	10.3	446.9
P4	19.6	38.1	11.8	583.8	12.2	267.8

References

[1] E. Cerny et al., "Automated Verification with Abstract State Machines Using Multiway Decision Graphs," *Formal Hardware Verification: Methods and Systems in Comparison*, pp. 79-113, Springer-Verlag Publishers, 1997.

[2] K. Fisler and S. Johnson, "Integrating Design and Verification Environments through Logic Supporting Hardware Diagrams," *In Proc. of IFIP Conf. on Hardware Description Languages and their Applications*, Japan, pp. 669-674, 1995.

[3] M. Fujita, Y. Matsunaga and T. Kakuda, "On variable ordering of Binary Decision Diagrams for the Applications of Multi-level Logic Synthesis," *In Proc. of the European Conf. on Design Automation*, USA, pp. 2-5, 1991.

[4] A. Gupta, "Formal Hardware Verification Methods: A Survey," *Formal Methods in System Design*, vol. 1, pp. 151-238, 1992.

[5] T. Kropf, "Benchmark-Circuits for Hardware Verification," *In TPCD 94, Sringer-Verlag*, pp. 1-12, 1994.

[6] S. Malik et al., "Logic verification using Binary Decision Diagrams in a Logic Synthesis Environment," *In ICCAD*, USA, pp. 6-9, 1988.

[7] C. Meinel and A. Slobodova, "Speeding up Variable Reordering of OBDDs," *In ICCD*, USA, pp. 338-343, 1997.

[8] S. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer Academic Publishers, 1997.

[9] S. Panda, F. Somenzi, "Who are the variables in your neighborhood?," *In ICCAD*, USA, pp. 74-77, 1995.

[10] R. Rudell, "Dynamic variable ordering for Ordered Binary Decision Diagrams," *In ICCAD*, USA, pp. 42-47, 1993.

[11] C. Scholl, D. Moller, P, "Molitor and R. Drechsler. BDD Minimization using Symmetries," *IEEE Transactions on CAD*, pp. 81-100, 1999.

[12] Y. Feng and E. Cerny, "Variable Ordering on Multiway Decision Graphs," *ISCAS*, Phoenix, 2002.

[13] Y. Saab & B. Rao, "Combinational Optimization by Stochastic Evolution". *IEEE Trans. on CAD*, Vol. 10, No 4, April 1991.

[14] C. Scholl, D. Moller, P. Molitor and R. Drechsler, "BDD Minimization using Symmetries," *IEEE Transactions on CAD*, pp. 81-100, 1999.