Manipulation of Regular Expressions Under Length Constraints Using Zero-Suppressed-BDDs

Shinya ISHIHARA Shin-ichi MINATO

Logic Design Systems Research Group
Advanced LSI Laboratory
NTT LSI Laboratories
3-1,Morinosato Wakamiya, Atsugi-city,
Kanagawa Pref., 243-01 Japan

Tel: +81-462-40-2443 Tel: +81-462-40-2308
Fax: +81-462-40-4322 Fax: +81-462-40-4322
e-mail: shinya@aecl.ntt.jp e-mail minato@aecl.ntt.jp

Abstract— We present a new technique that broadens the scope of BDD application. It is a method for manipulating regular expressions that represent sets of sequences including repetitions of symbols. In general, sequences in the set represented by a regular expression have an infinite length and this makes representing and manipulating them difficult.

In this paper, we introduce length constraints into a representation of regular expressions. Under these constraints, our method can represent and manipulate large-scale sets of sequences of regular expressions compactly and uniquely and greatly accelerates operations of regular expressions.

As regular expressions can represent behaviors of a finite state machine, our technique provides a useful analysis method of finite state machines and can be applied to formal hardware verification techniques.

I. Introduction

Recently, formal LSI design verification methods have attracted much attention [3, 1]. A typical approach is to show the correctness of designs by comparing the behavior of an implementation and its specification. Both an implementation and its specification are represented by finite state machines. Finite state machines play important roles as models in formal verification, so a method that enables their efficient manipulation is required. Previously proposed methods represent transition relations of finite state machines to analyze their behavior. Although the Binary-Decision-Diagram-based approaches [5, 15] are very sophisticated, there are cases when transition relations cannot be represented in reasonable time and space. In this paper, we propose an approach based on regular expressions for representing and manipulating finite state machines. Finite state machines are equivalent to regular expressions. Sequences in a set represented by a regular expression are equivalent to the transitions of its corresponding finite state machine. Regular expression inequivalence is NP-hard [13], [14] and of course finite state machine inequivalence is NP-hard too. To relax the complexity of manipulating regular expressions, we introduce length constraints into the representation of regular expressions. Length constraints enable us to manipulate regular expressions as finite sets of limited length sequences. We propose the way to represent sets of sequences as sets of combinations and a method of manipulating sets of combinations using Binary Decision Diagrams (BDDs).

BDDs are graph-based representations of Boolean functions and enable us to manipulate Boolean functions efficiently in terms of time and space [2]. There are many cases in which conventional algorithms can be significantly improved by using BDDs [8, 3]. As our understanding of BDDs has deepened, the range of applications has broadened. Besides Boolean functions, we are often faced with manipulating sets of combinations in many LSI design problems. By mapping a set of combinations into Boolean space, it can be represented as a characteristic function using a BDD. This lets us manipulate a huge number of combinations implicitly, which has never been practical before. Based on implicit set representation, new two-level logic minimization methods have been developed [4, 9]. These techniques are also used to solve a kind of covering problem [7]. Of course, transition relations of a finite state machine can be represented as a characteristic function using BDDs [5, 15].

A zero-suppressed BDD (ZBDD) is a new type of BDD adapted for the implicit set representation [12]. It can manipulate sets of combinations more efficiently than conventional BDDs, especially when dealing with sparse combinations. We have recently studied cube set algebra for manipulating sets of combinations [11], and proposed efficient algorithms for computing cube set operations based on ZBDDs. This technique is useful for many practical activities related to LSI design, including multi-level logic synthesis [10] and fault simulation.

In this paper, we present a new technique of manipulating regular expressions that represent sets of sequences including repetitions of symbols under length constraints by using ZBDDs. This method can represent regular expressions with large-scale sets of sequences compactly and uniquely. In this method, we can flatten regular expressions into canonical forms of sets with millions of sequences, which have never been represented before. Constructing canonical forms of sets of sequences immediately leads to equivalence checking of regular expressions. Since the calculus of regular expressions is a basic model for manipulating finite state machines, our method is expected to be useful for the formal verification of LSIs.

We will first explain regular expressions and ZBDDs.

Next we will present the method for representing regular expressions under length constraints with ZBDDs and discuss the algorithms of operation for regular expressions. Finally, we will show the implementation of our method and experimental results.

II. Preliminary

A. Regular Expressions

A regular expression provides a way of describing certain sets of sequences, and is defined over a certain set of symbols. Here, we suppose that regular expressions are built up by applying union, concatenation, and closure operations to the sets. To be more precise, let $\Sigma=\{\sigma_1,\cdots,\sigma_k\}$ be a finite set of symbols. Then:

- Each of the expressions $\sigma_1,\cdots,\sigma_k$ is a regular expression on Σ, as are λ (the null string) and ϕ (the empty set).
- If S and T are regular expressions on Σ, then so is their union S+T and their concatenation ST.
- If S is a regular expression on Σ, its closure S* ($=\lambda$ + S + SS + $\cdots$) is also a regular expression.
- Only those expressions that can be obtained by a finite number of applications of the above operations are regular expressions on Σ.

The set Σ is usually called the alphabet of the expressions based on it. In this paper, the notation **L(R)** is a set of strings which regular expression R represents, **|s|** represents the length of a sequence s, and $\mathbf{L(R)}_i$ is a subset of L(R), that is $\{s|s\in L(R),|s|=i\}$.

There is often a case when two quite different-looking regular expressions can represent the same set of sequences. For example, both a* + a*bb(b + aa*bb)*(λ + aa*) and (bbb* + a)* represent the same set of sequences that do not contain isolated b's. Similarly, both b*ab*(ab*ab*)* and (b*ab*a)*b*ab* represent the same set of sequences that contain an odd number of a's. In LSI designs, the implementation and specification is represented by different finite state machines, and the corresponding regular expressions are usually different. Therefore, checking the equivalence of two quite different-looking regular expressions is a useful way for us to verify LSI designs and that is our aim in this paper.

B. Zero-suppressed BDDs

Zero-suppressed BDDs (ZBDDs) are a new type of BDD adapted for representing sets of combinations [12]. They are based on the following reduction rules:

- Eliminate all nodes with the 1-edge pointing to the 0-terminal node. Then connect the edge to the other sub-graph directly (Fig. 1).
- Share all equivalent sub-graphs in the same manner as with conventional BDDs.

Notice that, contrary to conventional BDDs, we do not eliminate nodes whose two edges both point to the same node. This reduction rule is asymmetric for the two edges

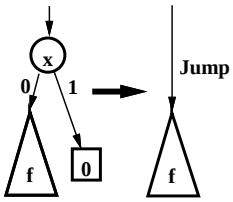

Fig. 1. Reduction Rule for ZBDDs.

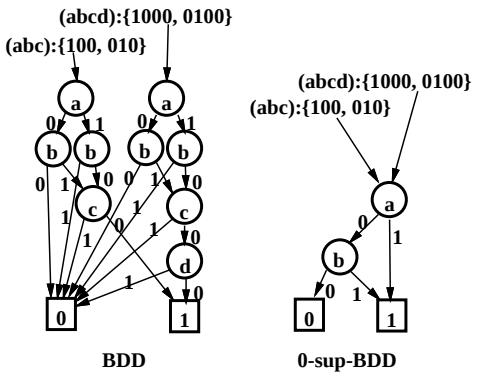

Fig. 2. Effect of the Reduction Rule.

because the nodes remain when their 0-edge points to a terminal node. When the number and order of the variables are fixed, ZBDDs provide canonical forms for Boolean functions.

Fig. 2 illustrates conventional and ZBDDs representing sets of combinations. Using the reduction rule, these two BDDs are automatically reduced into the same form, free of irrelevant variables. ZBDDs are more effective for sparser combinations, which means that only a few objects out of many are included in each combination in the set.

The methods for manipulating ZBDDs are defined as set operations and differ slightly from those for conventional BDD manipulation. First, we generate trivial graphs and then construct more complex ones by applying basic operations such as union, intersection, and difference. We can execute these operations in a time almost proportional to the size of the graphs, just as with conventional BDDs. (see [12] for detailed algorithms.)

Using ZBDDs, we can represent and manipulate Boolean expressions efficiently. For example, when expanding the expression $(a+b+c)(e+d+f)(g+h+i)\cdots$ into a sum-of-products form, an exponential number of product terms appears for the number of variables; however, a ZBDD implicitly represents these product terms in a linear number of nodes, as shown in Fig. 3. In this graph, each path from the root to the 1-terminal corresponds to each product term in the expression. In this way, we can represent a huge number of product terms within a practical memory space. In this paper, we define Z(X) as representation of set X of combinations by using

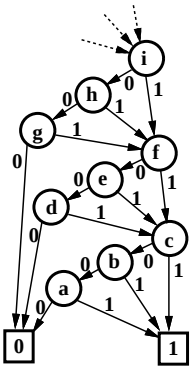

Fig. 3. ZBDD for (a+b+c)(d+e+f)(g+h+i)$\cdots$.

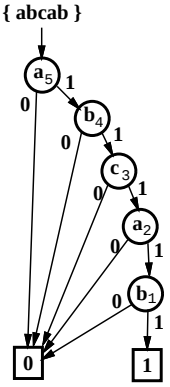

Fig. 4. ZBDD for the sequence "abcab"

ZBDD.

III. Representation of Regular Expressions using ZBDD

Here, we show how we represent and manipulate regular expressions using ZBDDs. A method we developed for manipulating Boolean expressions is also applicable to regular expressions. However, Boolean expressions are different from regular expressions in the following points:

- Boolean expressions can be represented as a set of combinations easily, and these can be dealt with efficiently by using ZBDDs. But sets of sequences that regular expressions represent are often infinite sets with infinitely long sequences, and regular expressions cannot be dealt with by using ZBDDs as they are.
- Boolean expressions cannot deal with repetitions of symbols. (xx=x, xyx=xy in Boolean expression.)
- Boolean expressions cannot represent permutations. (xy=yx in Boolean expression.)

In this section, we present a method for representing regular expressions by solving these problems.

A. Length Constraints

As we explained above, sets of sequences that regular expressions represent are usually infinite sets that have infinitely long sequences of symbols. ZBDDs cannot deal with such sets, so we introduce length constraints into them. Now, suppose that the length constraint is a non-negative integer ℓ, we consider only the subset of sequences whose length is shorter than or equals to ℓ. The size of this subset is at most the ℓ-th power of the size of the alphabet that appears in the regular expression. Sequences whose lengths are i correspond to behaviors of the finite state machine until the i-th step. And so, considering only the subset of the sequences whose lengths are shorter than or equal to ℓ is equivalent to considering all behavior of the corresponding finite state machine until the ℓ-th step. This is useful for formal verification.

From now on, when we do not refer to the length constraint, we assume it is a non-negative integer ℓ.

B. Representation of Permutation

The way of representing repetitions and permutations in ZBDDs is another problem. We introduced the length constraints above and considered only the subset of sequences of regular expression. However, this subset cannot be dealt with by using ZBDDs as it is, because it is a set of permutations including repetitions of symbols.

We propose preparing ℓ items (x_1,$\cdots$,x_ℓ) for each symbol (x $\in \Sigma$). When dealing with more than one sort of symbol, such as x, y, and z, we prepare respective ℓ items, i.e. $x_1, \cdots, x_\ell, y_1, \cdots, y_\ell, z_1, \cdots, z_\ell$. The sequence "abcab" is "abc" in Boolean algebra, but now "abcab" is expressed as a combination of items "$a_5b_4c_3a_2b_1$" (Fig. 4). Item x_i stands for the existence of symbol x at the i-th position of the sequences in reverse order. (We can represent "abcab" as "$a_1b_2c_3a_4b_5$", but for implementation we adopt the reverse order. And also for implementation, variable ordering of ZBDDs is based on a rule that item A whose subscript is bigger than item B has a higher position in ZBDDs than item B.)

This technique enables us to represent sets of sequences as sets of combinations of the items. Now we can use ZBDDs to represent them.

C. Data Structure

With the above representation methods, we can represent regular expressions by using ZBDDs. We provide a method of handling a set of sequences efficiently in ZBDDs. Our idea is to divide the set of sequences into an array of subsets. To put it more concretely, for regular expression R and length constraint ℓ, we divide the set of sequences L(R) into the array of (ℓ+1) subsets $L(R)_0$, $L(R)_1$, $\cdots$, $L(R)_\ell$. Then we represent them by using the array of ZBDDs. From now on, we use $Z(R)_i$ as a representation of $L(R)_i$ by using a ZBDD, and use the array [$Z(R)_0$, $Z(R)_1$, $\cdots$, $Z(R)_\ell$] to represent regular expression R.

For example, we represent the regular expression "ab*+a*b" under the length constraint $\ell = 3$. The set of

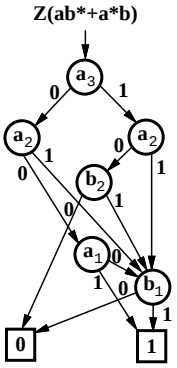

Fig. 5. Z(ab*+a*b).

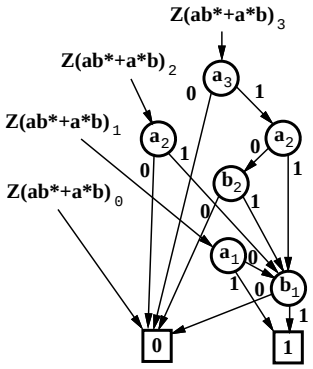

Fig. 6. The array of ZBDDs for "ab*+a*b".

sequences which "ab*+a*b" expresses is {a, b, ab, aab, abb, aaab, abbb, $\cdots$}. Because of the length constraint, let us consider only the subset of the sequences whose lengths are shorter than or equal to 3, i.e. {a, b, ab, aab, abb}. Fig. 5 shows the representation of this subset as it is. As we explained above, we use the array of ZBDDs. {a, b, ab, aab, abb} is divided into $L(ab^*+a^*b)_0 = \{\}$, $L(ab^*+a^*b)_1 = \{a, b\}$, $L(ab^*+a^*b)_2 = \{ab\}$ and $L(ab^*+a^*b)_3 = \{aab, abb\}$ according to their length, and these subsets are represented by using a ZBDD. Fig. 6 shows the array of ZBDDs which represents the regular expression "ab*+a*b" under the length constraint $\ell = 3$.

IV. Algorithms of Regular Expression Operations

Regular expressions can be manipulated by three operations, such as union, concatenation, and closure. Based on this knowledge, we first generate ZBDDs for trivial expressions that consist of a single symbol, and then apply these operations to construct more complicated regular expressions.

An example is shown in Fig. 7, 8, and 9. To generate

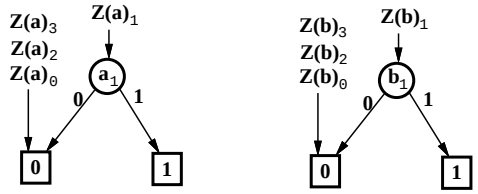

Fig. 7. The array of ZBDDs for "a" and "b".

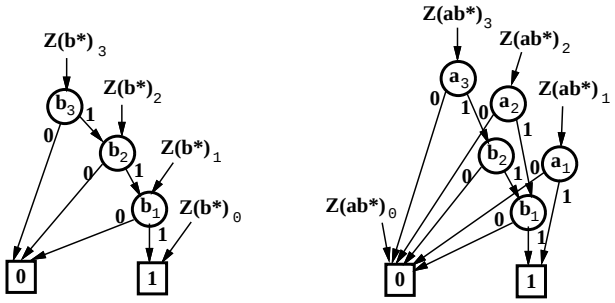

Fig. 8. The array of ZBDDs for "b*" and "ab*".

a ZBDD for the expression "ab*+a*b", we first generate graphs for "a" and "b"(Fig. 7). Then we apply union, concatenation, and closure operations according to the expression. Fig. 8 shows the representation of "b*", which is the closure of "b", and the representation of "ab*", which is the concatenation of "a" and "b*". Fig. 9 shows the representation of "a*", which is the closure of "a", and the representation of "a*b", which is the concatenation of "a*" and "b". Finally, we compute the union of "ab*" and "a*b", and we get the array of ZBDDs shown in Fig. 6.

After generating ZBDDs, we can immediately check the equivalence between two regular expressions under length constraints. Moreover, we can easily evaluate, for instance, the regular expressions in terms of the number of sequences of the set.

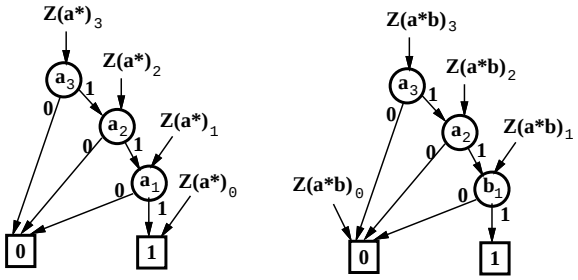

Fig. 9. The array of ZBDDs for "a*" and "a*b".

In this section we present efficient algorithms for the three regular expression operations of union, concatenation and closure using ZBDDs.

A. Union

We first show the algorithm for union. For regular expressions S and T, we assume that we will get a regular expression R as a result, i.e., R = S+T. This operation is a basic part of other operations. Regular expressions S and T are represented as an array $Z(S)_0$, $Z(S)_1$, $\cdots$, $Z(S)_\ell$, and an array $Z(T)_0$, $Z(T)_1$, $\cdots$, $Z(T)_\ell$ respectively by using ZBDDs. Computing a union of $Z(S)_i$ and $Z(T)_i$ by using the ZBDD union operation, we can get $Z(R)_i$. The computing time of union operation is proportional to number of ZBDD nodes.

B. Concatenation

Concatenation is the most significant regular expression operation. In order to implement it efficiently, we adopt the array of subsets of sequences as the data structure.

We assume a regular expression R is a concatenation of regular expressions S and T. A ZBDD $Z(R)_i$ for a subset of sequences $L(R)_i$ is obtained by

- computing concatenations $Z(S)_jZ(T)_k$, where $j+k = i$, $0 \leq j \leq i$, $0 \leq k \leq i$, and
- computing the union of (i+1) concatenations we get above, i.e. a union of $Z(S)_0Z(T)_i$, $\cdots$, $Z(S)_jZ(T)_k$, $\cdots$, $Z(S)_iZ(T)_0$.

A concatenation $Z(S)_jZ(T)_k$ is computed by using the shift operation and Cartesian product operation of ZBDDs. All items of $Z(S)_j$ are shifted up by k steps, then the Cartesian product with all items of $Z(T)_k$ is produced. To put it more concretely, a concatenation of $Z(S)_2 = \{a_2b_1, b_2a_1\}$ and $Z(T)_1 = \{a_1, b_1\}$ is computed by

- shifting $\{a_2b_1, b_2a_1\}$ up by 1 step to get $\{a_3b_2, b_3a_2\}$.
- making a Cartesian product with $\{a_1, b_1\}$ to get $\{a_3b_2a_1, a_3b_2b_1, b_3a_2a_1, b_3a_2b_1\}$.

The Cartesian product is a basic operation of ZBDDs and its computing time is proportional to the number of ZBDD nodes. So is the shift operation. As we explained above, the concatenation operation is implemented by applying the union and shift and the Cartesian product, the number of times of which is proportional to the second power of the length constraint. Therefore the computing time of concatenation operation is $O(N \cdot \ell^2)$, where N is the number of nodes of ZBDDs and ℓ is the length constraint. In practice, the cache function of ZBDDs makes the computing time shorter.

C. Closure

Using two operations above, we can compose a regular expression closure algorithm. The closure of regular expression R is $\lambda + R + RR + \cdots\cdots$. Under length constraints we consider only $\lambda + R + RR + \cdots + \underline{R\cdots R}(\ell)$ because sequences whose length is greater than ℓ are out of consideration. This expression can be factorized as $\lambda + R(\lambda + R(\lambda + \cdots R(\lambda + R)))\cdots)$. Applying operations of union and concatenation by turns, we can represent a regular expression closure by using ZBDDs. If an intermediate result is saturated during calculation, we can stop at that time and get the result. For example, when calculating "(a*)*", we can find "λ + a*(λ + a*)" = "λ + a*" i.e. saturation. We then immediately get the representation of "(a*)*" without calculating ℓ times.

The closure operation is implemented by applying union and concatenation, the number of times of which is proportional to the length constraint. Therefore the computing time of the closure operation is $O(C \cdot \ell) = O(N \cdot \ell^3)$, where C is the number of concatenations, ℓ is the length constraint, and N is the number of ZBDD nodes.

D. Miscellaneous

When we represent regular expressions using ZBDDs, they are represented in canonical form, and we can easily perform various operations by using the functions of ZBDDs. For example, checking the equivalence of two regular expressions can be done by only comparing two ZBDD arrays. Checking implication is easily implemented by using the subtraction operation of ZBDDs. The number of sequences can be enumerated by checking the number of ZBDD paths, and the computation time is proportional to the number of nodes. These operations are useful in various areas, including formal verification techniques for finite state machines.

V. Implementation and Experiments

Based on the techniques described above, we implemented a program for manipulating regular expressions under length constraints. Our program is written in C++ language on a SPARC Station 2 (SunOS 4.1.3, 64MByte Memory). To evaluate our method, we constructed ZBDDs for several regular expressions and for several length constraints.

The results are shown in Table I, where ℓ shows the length constraints, cardinality shows the number of sequences, nodes shows the number of the nodes in the ZBDDs, and time shows the total time for generating the ZBDDs. As shown in Table I, within a feasible time and space, we can generate ZBDDs for regular expressions which have extremely large-scale sets of sequences, some of which consist of millions of words. This has never been practical in conventional representation. Our method requires a memory space proportional to the number of words. With a careful look at the time column, it is apparent that computing time for regular expressions is $O(\ell^3)$, where ℓ is the length constraint. This is in agreement with the fact that the closure operation shown in section 4 requires a computation time that is proportional to the third power of the length constraint.

Our method greatly accelerates the operation of regular expressions and enlarges the scale of applicability. It is especially effective when dealing with many sorts of variables, a feat that has been difficult with conventional methods.

VI. Conclusion

We have proposed an elegant way to represent regular expressions under length constraints using ZBDDs and

TABLE I
RESULTS FOR REGULAR EXPRESSIONS.

Regular Expression	ℓ	cardinality	nodes	time
a*b*c*	32	6,545	96	7.53
	64	47,905	192	58.77
	96	156,849	288	199.8
(a+b+c)*	32	2.8×10^{15}	96	2.56
	64	5.2×10^{30}	192	19.2
	96	2.8×10^{45}	288	65.65
((a*+b)*+c)*	32	2.8×10^{15}	96	17.62
	64	5.2×10^{30}	192	146.35
	96	2.8×10^{45}	288	498.3
(a+bb)*(b+(aa)*)*cc	32	21,919,487	197	11.89
	64	$(> 2^{32})$	421	90.15
	96	$(\gg 2^{32})$	645	291.58
a*+a*bb(b+aa*bb)*(λ+aa*)	32	109,870,575	124	17.15
	64	$(> 2^{32})$	252	134.78
	96	$(\gg 2^{32})$	380	457.71
(bbb*+a)*	32	109,870,575	124	10.3
	64	$(> 2^{32})$	252	81.45
	96	$(\gg 2^{32})$	380	278

have shown efficient algorithms for manipulating those expressions. The experimental results show that we can implicitly manipulate regular expressions under length constraints with large-scale sets of sequences in a feasible time and space. An important feature is that our representation provides us with the canonical form of regular expressions under length constraints. As the regular expressions calculus is a basic model for manipulating finite state machines, our method is very useful in LSI CAD, especially in the formal verification of LSIs. Under length constraints ℓ, we observe all strings whose length are less than or equal to ℓ, and so we can confirm all behaviors of the finite state machine until the ℓ-th step. Therefore we can verify sequential circuits. For example, we can check whether or not a microprocessor runs the same way as specified until the ℓ-th step.

Our method verifies finite state machines until a certain ℓ step, but cannot give the complete validation for infinite behavior. However, there are many cases where constraints that are long enough bring the same results as formal verification. In the future we will consider the relation between length constraints and the complexity of regular expressions. Moreover, we will try to speed up the manipulation of regular expressions and to manipulate them without length constraints.

ACKNOWLEDGEMENTS

We wish to acknowledge valuable discussions with Mr. Takahara.

REFERENCES

[1] A. Aziz, F. Balarin, S.-T. Cheng, R. Hojati, T. Kam, S. C. Krishnan, R. K. Ranjan, T. R. Shiple, V. Singhal, S. Tasiran, H.-Y. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. HSIS: A BDD-Based Environment for Formal Verification. In *Proc. of ACM / IEEE DAC*, pages 454–459, June 1994.

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

[3] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential circuit verification using symbolic model checking. In *Proc. of ACM / IEEE DAC*, pages 618–624, June 1990.

[4] O. Coudert, J. Madre, and H. Fraisse. A new viewpoint of two-level logic optimization. In *Proc. of ACM / IEEE DAC*, pages 625–630, June 1993.

[5] Olivier Coudert and Jean Christophe Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proc. of ICCAD*, pages 126–129, Nov 1990.

[6] Frederick C. Hennie. *FINITE-STATE MODELS FOR LOGICAL MACHINES*. John Wiley & Sons, Inc., 1968.

[7] B. Lin and F. Somenzi. Minimization of symbolic relations. In *Proc. of ICCAD*, pages 88–91, Nov 1990.

[8] Y. Matsunaga and M. Fujita. Multi-level logic optimization using binary decision diagrams. In *Proc. of ICCAD*, pages 556–559, Nov 1989.

[9] P. McGeer, J. Sanghavi, R. Brayton, and A. L. Sangiovanni-Vincentelli. ESPRESSO-SIGNATURE: A new exact minimizer for logic functions. In *Proc. of ACM / IEEE DAC*, pages 618–624, June 1993.

[10] S. Minato. Fast weak-division method for implicit cube representation. In *Proc. of Synthesis and Simulation Meeting and International Interchange (SASIMI, Japan)*, pages 423–432, Oct 1993.

[11] S. Minato. Calculation of unate cube set algebra using zero-suppressed BDDs. In *Proc. of ACM / IEEE DAC*, pages 420–424, June 1994.

[12] Shin-ichi Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pages 272–277, June 1993.

[13] L. J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA., 1974.

[14] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proc. 5th Ann. ACM Symp. on Theory of Computing*, pages 1–9, New York, 1973. Association for Computing Machinery.

[15] Herve J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. of ICCAD*, pages 130–133, Nov 1990.