Novel Applications of a Compact Binary Decision Diagram Library to Important Industrial Problems

● Stergios Stergiou ● Jawahar Jain

(Manuscript received May 28, 2009)

Fujitsu Laboratories of America has, over the course of many years, worked to develop the frontier of binary decision diagram (BDD) technology under a project called ParDD. Our technology allows us to partition Boolean functions, represent them very compactly, and process them on a massively parallel computing platform. It has been used to create numerous applications in the field of electronic design automation. Recently under this project we have developed a novel BDD library where the storage requirement of each node closely tracks the total size of the stored representation. The compact nature of this data structure allows the solution of interesting problems to which BDDs have seldom been applied before. For example, we have used our library to create a compact inverted index, an essential matrix for indexing documents in any corpus, including the World Wide Web. We have also characterized its performance for Web query satisfaction in the context of Web searches as well as for the creation of compact representations of access control lists, a core component of Internet routers.

1. Introduction

In computer science, many problems can be formulated in terms of Boolean functions. A binary decision diagram (BDD)[1] is a directed acyclic graph used to compactly represent a Boolean function. It includes two special "terminal" nodes that represent the Boolean functions 1 and 0. Each non-terminal node, which corresponds to a subfunction f, is labeled by a Boolean variable v and has two outgoing edges. Edge "1" points to the sub-BDD that represents function $v \cdot f$, while edge "0" points to the sub-BDD for function $\overline{v} \cdot f$. The two edges point to different nodes.

A reduced ordered BDD (ROBDD) is a BDD with two additional restrictions. First, all paths from its root to the leaves examine variables in the same variable order[2]. Second, there should be no isomorphic subgraphs. These restrictions lead to a canonical representation for a given variable order.

BDD graphs can be manipulated efficiently. Any Boolean operation between two graphs can be completed in time that is at most quadratic in the size of the given graphs. For a large variety of functions that naturally arise in real-life applications, their BDDs have been observed to be compact. Compactness and efficiency have led to many BDD applications in areas such as design simulation, synthesis, verification, automatic test generation, artificial intelligence, data mining, software security, and fault tolerant computing.

ROBDDs provide efficient representations for many functions of practical interest. Unfortunately, some applications require the representation of functions that have only an exponential ROBDD size. This limits the complexity of problems that can be attacked using ROBDDs.

1.1 Partitioned ROBDDs

Fujitsu Laboratories of America (FLA) has developed a more efficient representation in its ParDD project through the use of partitioned ROBDDs (POBDDs)[3)-5)], which are especially effective for large designs. In this approach, different partitions of the Boolean space are allowed to have different variable orderings, and only one partition needs to be in memory at any given time.

To handle the complexity of large industrial designs, we have proposed algorithms that modify the Boolean space partition of POBDDs at run-time, thus avoiding memory explosion. Theoretical evidence[5),6)] suggests that representations using this approach can be exponentially more compact than ROBDDs as well as any approach using a fixed number of partitions.

We incorporated this dynamic repartitioning scheme in reachability[7)] based invariant checking as well as model checking for a portion of computation tree logic.[8),9)] Because of the partitioned nature of POBDDs, FLA has been able to develop methods that allow efficient mathematical models[10),11)] as well as highly effective use of symmetric multiprocessor architectures[12)] and large computational grids[13)] where super-linear gains over classical approaches have been observed in proving falsification. The adaptive nature of our partitioning approach also leads to order-of-magnitude more efficient runtime in proving design correctness.

1.2 Nano decision diagrams

In the ParDD project we have recently deviated from the classical BDD approach of a fixed data structure per vertex. Instead, we maintain the necessary bookkeeping information as compactly as possible as a function of the OBDD size.

Let n be the number of variables and d be the number of nodes of a given BDD. Then s_n = [log (n)] bits are sufficient to index a variable. Moreover, if nodes are stored consecutively in memory, s_d = [log (d)] bits are sufficient to identify their location. On the basis of the above observations, each node is structured as follows:

variable: s_n bits	1-edge: s_d bits	0-edge: s_d bits

For comparison, traditional decision diagram libraries specify conservative upper bounds for variable and index bits, typically 24 and 32 bits, respectively. To reflect the lighter memory footprint of our new decision diagram structure, we named it the nano decision diagram (nanoDD).

2. NDD library

Our nanoDD library (NDD library) provides an implementation of nanoDDs. It has been designed and implemented from scratch to additionally support a variation of BDDs called zero-suppressed BDDs (ZDDs).[14)] ZDDs have been shown to be more compact in terms of the number of nodes required to support a given Boolean function, provided that the function's ON-set is relatively sparse.

The NDD library implements all classical 2-operand operations, as well as the operations *Constrain*, *Restrict*, and *ITE*. It implements variable reordering through the classical *Sifting* algorithm. Another novel aspect of this library is that it supports the execution of all operations within a user-specified *context*. Within each context, the user assigns 2-operand operations to variables. Whenever the creation of a new node is requested, the library checks whether the variable of the new node is assigned to an operation. If so, the assigned operation is applied to the new node's children and the result is returned instead. Contexts seamlessly encapsulate universal and existential quantification schedules such that they can be used with all operations while maintaining a simple programming interface.

3. BDD-based inverted index representation

An inverted index is a data structure that operates on a collection of documents and is used to efficiently identify the subset of documents that include a specific keyword. It can be stored as a collection of lists, each of which corresponds to a unique keyword w_i and includes the numerical identifiers of the documents that contain w_i.

The size of the inverted index can grow quite large, with direct implications for the required storage space and access time. Therefore, in many cases, each list is stored in a compressed manner that allows it to be quickly and incrementally decompressed. In this paper, we analyze nanoDDs for representing inverted indices. Below we give some background information about the mainstream list compression scheme and BDDs.

For each list, the corresponding Boolean function is constructed and the BDD for it is built with a traditional BDD package using the ZDD representation. There are two aspects to representing lists with decision diagrams. The first concerns the mapping of list elements to a Boolean function and the second is related to the way the decision diagrams are stored on disk.

3.1 Encodings

3.1.1 Lists as Boolean functions

Let us represent list [23, 33, 37, 54] as a Boolean function. In binary, the list elements are [010111, 100001, 100101, 110110].

3.1.2 Binary encoding

The Boolean function that represents the list with the minimum number of variables is obtained by simply assigning each variable to each significant bit weight. For example, the above list corresponds to function

$$f = \overline{x}_1 x_2 \overline{x}_3 x_4 x_5 x_6 + x_1 \overline{x}_2 \overline{x}_3 \overline{x}_4 \overline{x}_5 x_6 + x_1 \overline{x}_2 \overline{x}_3 x_4 \overline{x}_5 x_6 + x_1 x_2 \overline{x}_3 x_4 x_5 \overline{x}_6.$$

3.1.3 Linear encoding

An alternative representation would be to assign a different variable for each document id. However, this representation is impractical because the number of documents can be quite large. Moreover, node sharing is no longer possible (unless multiple lists are represented by a single Boolean function.)

3.1.4 Base-2^k encoding

Let us represent the list elements in a 2^k base. This allows linear and binary encoding to be combined. For each of the base-2^k digits, we use 2^k distinct variables to represent them in a one-hot manner. For example, assume that we want to encode number 54, which is 312 in base-4. Each of the digits is one-hot encoded, giving 1000 : 0010 : 0100. Therefore, element 54 is encoded as

$$g = x_1 \overline{x}_2 \overline{x}_3 \overline{x}_4 \overline{x}_5 \overline{x}_6 x_7 \overline{x}_8 \overline{x}_9 x_{10} \overline{x}_{11} \overline{x}_{12}.$$

This increase in the number of variables may initially appear inefficient, but in fact it leads to better sharing and more compact representation, especially when ZDDs are used.

3.2. Performance characterization

3.2.1 Corpus

In order to benchmark the NDD library, we created an inverted index for the largest possible set of Web pages available online. This set, which was downloaded from Stanford's WebBase project, contains more than 94 million Web pages. By comparison, the first Google implementation had only 25 million.

Table 1
Inverted index statistics.

Processed pages	94 million
Unique terms	114 million
Processed terms	22 000 million
List-based inverted index	163 GB

For each Web page, we extracted its text terms. Every sequence of up to 50 characters was kept. In total, almost 22 billion terms were processed. For each unique term, we calculated the set of pages in which it appeared. There were more than 114 million unique terms.

The collection of all sets of pages for all unique terms is the inverted index. The size of the complete inverted index in the classic, uncompressed list implementation is 163 GB (**Table 1**).

3.2.2 NanoDD inverted index

We computed the complete nanoDD-based inverted index for the Stanford crawl. The computation time was less than 25 hours. By comparison, it took almost 4 days to parse the corpus and generate the list-based inverted index. Therefore, while the time required to build the nanoDD inverted index is not negligible, we do not consider it to be an issue. The resulting nanoDD inverted index was less than 25% of the size obtained using the classical list-based approach. The clear major benefit of this result is the cost decrease emanating from the direct reduction of disk space required for storing the inverted index.

3.2.3 Comparison with existing BDD packages

To compare the performance of our NDD library with the state-of-the-art BDD package (Colorado University Decision Diagram, CUDD), we implemented a tool that generates a ZDD-based inverted index using the open-source CUDD library. Our results are as follows.

In terms of computation time, our nanoDD approach was almost 8 times as fast as the CUDD implementation. In terms of memory requirements, our inverted index was less than 1/6 the size of the CUDD one. Note that the decrease in size is not due to the structure of the obtained ZDDs because, by construction, they are canonical (and therefore the same in both approaches). It is due to the structure of the nodes. For CUDD, the size is at least 16 bytes per node (depending on the version used) independent of the actual function stored. For nanoDDs, the size varies depending on the nodes needed to store a given function and can range from 2 to 8 bytes. In both approaches, the intermediate memory required for caches was not taken into account since the cache contents are not stored as part of the result.

We also compared our NDD library with CUDD[15] and Cal[16] on ACM/SIGDA combinational circuits. We use static variable orders as computed by a depth-first search traversal of the circuit. Only non-trivial circuits that could be completed in either of the libraries within 1800 s are shown. The results are given in **Table 2**.

4. NanoDD-based Web searches

The major benefit of using nanoDD inverted indices for Web searches is that they maintain manipulability. To achieve this, we designed special manipulation operations that are more suited to the task of Web searching. In addition,

Table 2
Comparison with CUDD 2.4.2 and Cal 2.1 on ACM/SIGDA circuits. Time is given in seconds and memory (mem) in MB. R_t is the time ratio, R_m is the space ratio, and $R_p = R_t * R_m$.

Circuit			NDD		CUDD					Cal				
Name	In	Out	Time	Mem	Time	Mem	R_t	R_m	R_p	Time	Mem	R_t	R_m	R_p
C3540	50	22	0.49	64.3	0.59	64	120%	100%	120%	1.05	52.1	214%	81%	174%
i10	257	224	0.5	63.2	0.62	74.2	124%	117%	146%	1.19	64.7	238%	102%	244%
C6288	32	32	225.17	5149.9	402.36	6312.6	179%	123%	219%	468.67	6278.2	208%	122%	254%
C2670	233	140	9.05	285.6	21.17	402.6	234%	141%	330%	22.79	564.2	252%	198%	497%

we adapted the storage scheme of the nanoDDs to the application.

4.1. Operations

The basic operation performed between K ordered lists is a conjunction, which is implemented as a K-way merge. Specifically, elements are read one at a time starting from the head of the lists until all common elements have been detected.

For example, let us detect the common elements between the following two lists.

*list*1 : 10, 20, 23, 36, 47, 52

*list*2 : 16, 18, 23, 47

We maintain pointers p_1, p_2 to the list elements, which initially point to elements "10" and "16", respectively. Since p_1 points to an element that is smaller than the one that p_2 points to, it moves forward to element "20". Now p_2 points to a smaller element, so it advances to "18". Since "18" is also smaller than "20", p_2 proceeds to "23". Now, p_1 proceeds to "23" and this common element is output. At this stage, the two pointers move forward to elements "47" and "47", respectively. Again, element "47" is output. Since p_2 has reached the end of list 2, no more common elements can be detected, so the process is complete.

We note that the basic operation implemented for traversing lists is essentially *get_next_element (L)*. In reality, the operation that we would like to implement efficiently for nanoDDs is *get_next_element_greq (L, element)* to detect the next element in list *L* that is greater than or equal to *element*.

get_next_element_greq (L, element)

We maintain an array of variable assignments *A* that is updated while traversing the nanoDD. The first element stored in the nanoDD is obtained by performing a depth-first traversal starting from the root node and initially following 0-edges until terminal node 1 is reached.

For each visited node, we monitor the variable id and the id of the edge that was followed. Variables that do not appear in the path from the root to the terminal node 1 are initially assigned the value 0.

Whenever *get_next_element_greq (L, element)* is called, the binary representation of *element* is checked against array *A*, and the number of common variable assignments from the root is detected. The algorithm backtracks until the first non-common variable from the top (or the root if there are no common assignments) and traverses the nanoDD according to the remaining assignments imposed by element.

Let us see how *get_next_element_greq* works for the simple decision diagram shown in **Figure 1**.

This decision diagram represents function $f = x_1x_3x_4 + x_1\bar{x}_3\bar{x}_4$, so it encodes list [8, 11, 12, 15] The first element is obtained by the traversal shown in **Figure 2**.

The variable assignments are therefore $(x_1, x_2, x_3, x_4) = (1, 0, 0, 0)$, giving the first list element "8".

If we wanted to access the next element in the list, we would search for the next element greater than "8" with *get_next_element_greq (L, 9)*. Then the algorithm would backtrack to variable x_3, since the first three variable assignments between (1, 0, 0, 0) and (1, 0, 0, 1) are the same, and continue along the path shown in **Figure 3**.

The current variable assignments are (1, 0, 1, 1), giving us element "11". We can obtain the remaining list elements in a similar manner.

The power of decision diagrams in the context of searches stems from the fact that elements of the underlying list can be skipped over if their presence is of no importance. Assume for example that the conjunction between lists [8, 11, 12, 15] and [7, 13, 15] is desired.

The first elements of both lists are obtained.

Since "8" is larger than "7", the next element greater that or equal to "8" is searched for in the second list and element "13" is obtained. Next, *get_next_element_greq (L, 13)* is applied to the first list. At this point, the algorithm detects that (1, 1, 0, 1) (corresponding to "13") has only the first variable common with (1, 0, 0, 0) (which corresponds to "8"). Subsequently, it directly backtracks to variable x_1 and traverses down the nanoDD, setting variables following x_1 in a manner consistent with the requested assignment (1, 1, 0, 1) and eventually ending up at (1, 1, 1, 1).

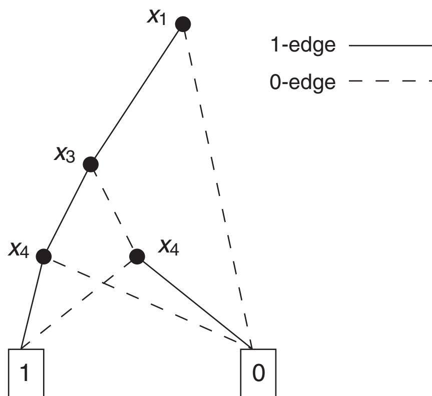

Figure 1
Example of simple decision diagram.

4.2 Storage scheme

A single nanoDD node requires exactly $2s_d + s_n$ bits. Nodes are stored consecutively in memory or on disk in the order that the depth-first traversal visits them, where 0-edges are followed before 1-edges. In this way, we can incrementally extract information from a nanoDD on disk. Terminal nodes need not be explicitly stored since they can be assigned fixed "virtual" positions.

4.3 Performance characterization

4.3.1 AOL query data

AOL recently released anonymized information about actual search queries performed by its search engine, which is essentially a front-end for Google. As these are actual user queries, they provide the best opportunity for benchmarking the performance of our operations.

4.3.2 Experimental results

We averaged the query times for a random set of 10 000 queries from the AOL query set, which are k-terms or longer, for $k = [2 \dots 6]$ to investigate how the system performs for progressively more complex queries. For each set of 10 000 queries, both the nanoDD-based manipulation code and the list-based manipulation code were executed. The

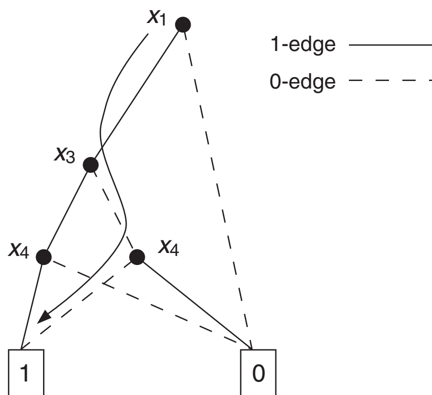

Figure 2
Traversal for obtaining first element.

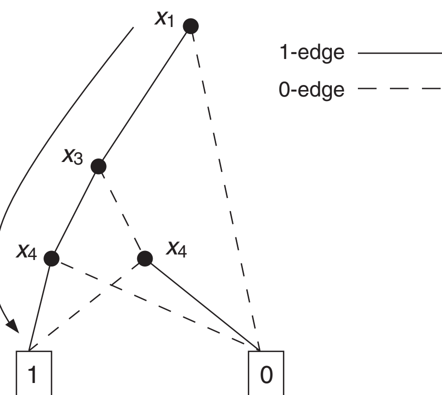

Figure 3
Continuation path.

Table 3
Comparison of list-based and nanoDD-based manipulation times.

k	Performance ratio between nanoDDs and lists
2	115%
3	125%
4	130%
5	140%
6	150%

Table 4
Energy consumption of nanoDDs as a percentage of list-based approach.

k	Energy as % of list-based approach
2	64%
3	59%
4	57%
5	53%
6	49%

performance ratio is given in **Table 3**.

4.4 Energy analysis

At an abstract level, the energy consumption of a typical computing node of a search engine comprises two components: the silicon factors (CPU, chipset, and memory) and the hard drive factor. We will assume an average power dissipation of 60 W for the CPU and 8 W per hard drive. Since the size of the inverted index is less than 25% of the size of the list-based representation, let us assume average power consumption of 92 W for the list-based inverted index setup and 68 W for the nanoDD-based setup. This corresponds to a single hard drive node for nanoDDs and four hard drive nodes for explicit lists. Thus, the power consumption of the nanoDD setup is 74% of the list-based setup.

If we factor in the benefits due to the higher query performance, we see that the energy consumption is reduced as listed in **Table 4**. The energy reduction is justified because the operations require less time to complete.

5. NanoDD-based access control list compression

5.1 Problem formulation

In its simplest incarnation, the problem of Internet router access control is as follows: A list of source/destination IP address tuples is maintained, which denotes the packets that are not allowed to be forwarded through the router. Let this list be $L = \{<IP_s, IP_d>_i\}$. IP addresses are typically 32-bit-long integers, so each tuple is characterized by a 64-bit number.

For each tuple i we construct a minterm m_i that depends on 64 variables. For example, tuple

11011110101011110010000000000111
00100000000001101101111010101111

is represented by minterm

$$x_1x_2\bar{x}_3x_4x_5 \cdots \bar{x}_{60}x_{61}x_{62}x_{63}x_{64}.$$

Next, we construct function *f*, where

$$f(x_1,\cdots,x_{64}) = \bigvee_{i=1}^{|L|} m_i.$$

We subsequently build the ZDD for function f and perform a depth-first search on the resulting directed acyclic graph to obtain a sum-of-products representation for f in the form of $f = \bigvee_{i=1}^{|L|} p_i$. Each product term p_i depends on a subset of the 64 variables and can therefore be represented in positional notation by using three symbols (0, 1, X). For example, product term $x_1\bar{x}_4x_6$ is represented in positional notation by

1*XX*0*X*1*XXX* ⋯ *XX*.

When presented with a source/destination IP tuple T, the access control list (ACL) subsystem must decide whether or not the corresponding packet should be forwarded. This operation is done by evaluating function f with variable assignments obtained from tuple T. For example, if tuple T is 10011 ⋯ 10 in binary, then f (1, 0, 0,

1, 1, ⋯, 1, 0) is calculated.

5.2 Experimental evaluation

5.2.1 Border gateway protocol

The border gateway protocol (BGP) is the core routing protocol of the Internet. It works by maintaining a table of IP networks or “prefixes” that designate network reachability among autonomous systems. It is described as a path vector protocol. BGP makes routing decisions based on path, network policies, and/or rulesets. For the purposes of our experimental evaluation, to construct function f, we collected 200 000 rules originating from AT&T’s network.

5.2.2 Experimental results

We used our NDD library to compute and store function f. The obtained compaction was more than 400%. Naturally, our technique is useful mainly for the cases where there is no hardware acceleration for ACL support since the cost of extracting information from the BDD would otherwise outweigh the benefit of the obtained compaction.

6. Conclusion

At FLA, the ParDD project first focused on creating partitions of Boolean functions to represent them very compactly and process them efficiently on a massively parallel computing platform. This technology was used to create numerous applications in the field of electronic design automation. To find BDD applications in other fields, we were motivated to develop a novel BDD library whose node structure can be dynamically adjusted to match the size of the stored representation. The compact nature of this data structure spawned solutions to interesting problems where BDDs have seldom been applied.

Apart from compact node size, and thus a smaller memory footprint, nanoDDs were found to yield significantly faster BDD manipulations, as demonstrated on ACM/SIGDA benchmark circuits as well as for Fujitsu proprietary designs. Given the wide variety of problems in which BDDs are used, gains in both space and time are a compelling proof of the significance of FLA’s ParDD project technology.

References

1) S. B. Akers: Binary decision diagrams. *IEEE Transactions on Computers*, Vol. C-27, No. 6, pp. 509–516 (1978).
2) R. E. Bryant: Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, Vol. 35, No. 8, pp. 677–691 (1986).
3) J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham: Functional partitioning for verification and related problems. Proceedings of Brown/MIT VLSI Conference, 1992.
4) A. Narayan, A. Isles, J. Jain, R. Brayton, and A. Sangiovanni-Vincentelli: Reachability analysis using partitioned-ROBDDs. Proceedings of IEEE/ACM International Conference on Computer-Aided Design, 1997.
5) A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli: Partitioned-ROBDDs—a compact, canonical and efficiently manipulable representation for Boolean functions. Proceedings of IEEE/ACM International Conference on Computer-Aided Design, 1996.
6) B. Bollig and I. Wegener: Partitioned BDDs vs. other BDD models. Proceedings of the International Workshop on Logic and Synthesis, 1997.
7) O. Coudert and J. C. Madre: A unified framework for the formal verification of sequential circuits. International Conference on Computer Aided Design, 1990, pp. 126–129.
8) E. M. Clarke and E. A. Emerson: Design and synthesis of synchronization skeletons using branching time temporal logic. Proc. IBM Workshop on Logics of Programs, Lecture Notes in Computer Science, Vol. 131, 1981, pp. 52–71.
9) K. L. McMillan: Symbolic model checking. Kluwer Academic Publishers, 1993.
10) S. Iyer, D. Sahoo, E. A. Emerson, and J. Jain: On partitioning and symbolic model checking. Proceedings of the International Symposium of Formal Methods, 2005.
11) S. Stergiou and J. Jain: Disjunctive transition relation decomposition for efficient reachability analysis. Proceedings of the IEEE International High Level Design Validation and Test Workshop, 2006.
12) D. Sahoo, J. Jain, S. Iyer, D. Dill, and E. A. Emerson: Multi-threaded reachability. Proceedings of the 42nd Design Automation Conference, 2005.
13) S. Iyer, J. Jain, D. Sahoo, and E. A. Emerson: Under-approximation heuristics for grid-based BMC. Proceedings of the 4th International Workshop on Parallel and Distributed Methods in Verification, 2005.
14) S. Minato: Zero-suppressed BDDs for set

manipulation in combinatorial problems. Design Automation Conference, 1993, pp. 272–277.
15) F. Somenzi: CUDD: CU decision diagram package—release 2.4.2, 2009.
16) R. Ranjan: CAL: Binary decision diagram package—release 2.1, 1998.

Stergios Stergiou
Fujitsu Laboratories of America
Dr. Stergiou received a B.S. degree in Computer Science from the Dept of Informatics, Athens, Greece, an M.S. degree in Computer Science from the Dept. of Computer Engineering and Informatics, Patra, Greece, and a Ph.D. degree in Computer Science from the National Technical University of Athens, Greece. He joined FLA in 2006 and has been engaged in research and development of formal verification and logic synthesis algorithms as well as Web-related technologies. He is the author of more than 20 publications.

Jawahar Jain
Fujitsu Laboratories of America
Dr. Jain received M.S. and Ph.D. degrees in Electrical and Computer Engineering from the University of Texas at Austin in 1989 and 1993, respectively. Thereafter, he worked as a joint post-doctoral fellow at the University of Texas at Austin and Texas A&M. He joined FLA in 1994 and conducted research in the area of electronic design automation for more than ten years. He has more than 50 publications and 40 patents (granted or pending). He is currently a Senior Research Fellow actively researching applications of symbolic manipulation technologies in diverse areas including Web-based services. One of his key interests is novel applications of information processing technologies in the field of health care. He has co-authored important publications in the area of partitioned BDDs and their applications to symbolic model checking with Turing Award winner Professor Emerson of the University of Texas at Austin.