Efficient BDD Algorithms for FSM Synthesis and Verification

Rajeev K. Ranjan* Adnan Aziz† Robert K. Brayton
Department of Electrical Engg. and Computer Sc.
University of California at Berkeley
Berkeley, CA 94720

Bernard Plessier Carl Pixley
Motorola Inc., MD OE321
6501 Wm Cannon Drive West
Austin, TX 78735

Abstract

We describe a set of BDD based algorithms for efficient FSM synthesis and verification. We establish that the core computation in both synthesis and verification is forming the image and pre-image of sets of states under the transition relation characterizing the design. To make these steps as efficient as possible, we address BDD variable ordering, use of partitioned transition relations, and use of clustering. We provide an integrated set of algorithms and give references and comparisons with previous work. We report experimental results on a series of seven industrial examples containing from 28 to 172 binary valued latches.

1 Introduction

The advent of modern VLSI CAD tools has radically changed the process of designing digital systems. The first CAD tools automated the final stages of design, such as placement and routing. As the low level steps became better understood, the focus shifted to the higher stages. In particular logic synthesis, the science of optimizing designs (for various measures such as area, speed, or power) specified at the gate level, has shifted to the forefront of CAD research. Another area rapidly gaining importance is design verification, the study of systematic methods for formally proving the correctness of designs.

Currently, a major area of research in logic synthesis is the automatic optimization of sequential hardware, i.e. finite state machines (FSM's) described as a netlist of gates and latches. This includes exploiting sequential don't cares, state minimization, FSM equivalence checking, and sequential ATPG [6, 15, 17, 20].

Formal design verification is a term given to the process of mathematically proving that a system possesses a given set of properties. Two popular paradigms for automated verification are language containment and model checking. In language containment, the property is specified as a set of acceptable output traces; verification consists of checking that the set of traces generated by the system is contained in the set of acceptable traces. In model checking, the property is a formula from a temporal logic; verification consists of checking that the system models the formula.

Both sequential synthesis and formal verification algorithms typically proceed by traversing the state transition graph (STG) of the design. Since large designs invariably consist of a set of interacting components, the number of states in the design is the product of the number of states in each component. This combinatorial blow up explosion is referred to as state explosion. [9] pioneered the use of Binary Decision Diagrams (BDD's) to implicitly manipulate the product state space in the context of implementation verification. Since then BDD's have been extended to manipulate transition systems in the area of design verification [2, 5, 7, 10, 12]. However, the size of BDD's arising in synthesis and verification computations continues to be a bottleneck.

In this paper we argue that the core computations in synthesis and verification are that of taking a given set of states and finding all states which can reach/be reached the given set in one step. These operations are referred to as the image and pre-image; our major contribution in this paper is an integrated set of BDD algorithms that perform this operation as efficiently as possible. Specifically, we address the following:

- BDD variable ordering techniques
- Use of clustering, i.e. partitioning the design into tractable components
- Ordering of the clusters, i.e. finding a sequence in which to process the clusters

We give reference and comparisons to past work with the details of these techniques. One salient feature of our algorithms is that they are completely automatic. The methods are interdependent and can be used together in various combinations. We report experimental results on a benchmark of seven relatively large industrial designs.

* Supported by Motorola Grant
† Supported by SRC 93-DC-008

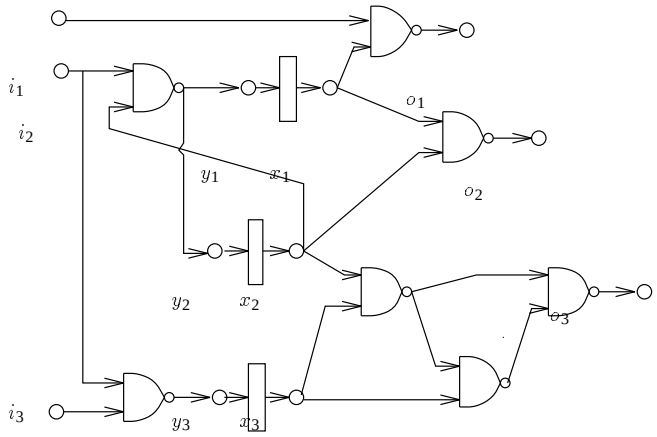

Figure 1: Gates + Latches = Sequential Network

The remainder of this paper is structured as follows. Section 2 reviews the basic definitions and terminology. In Section 3 we describe for each algorithm the underlying theory and discuss the experimental results. We comment on the implications of our results and indicate avenues for future work in Section 4.

2 Definitions and Terminology

We now make the notion of a finite state machine and our model for sequential hardware precise. We also describe the representation of FSM's using BDD's.

Definition 1 A *Finite State Machine (FSM)* M is a quintuple, $(Q, I, O, \lambda, \delta)$, where Q is the set of states, I is the set of input values, O is the set of output values, λ is the output function, and δ is the next state function. The output function λ is a completely-specified function with domain $(Q \times I)$ and range O. The next state function is a completely-specified function with domain $(Q \times I)$ and range Q.

Given a FSM $(Q, I, O, \lambda, \delta)$, its *transition relation* is the function $\mathbf{T} : S \times I \times S \longrightarrow \{0,1\}$; $\mathbf{T}(x, i, y) = 1 \leftrightarrow y = \delta(x, i)$. A set of states $A \subset Q$ can be associated to its *characteristic function* $\mathbf{A} : Q \rightarrow \{0,1\}$; $\mathbf{A}(x) = 1 \leftrightarrow x \in A$

A hardware *design* D consists of a set of interconnected latches and gates, as illustrated in Figure 1. For the purposes of this paper, a design with n latches, m output wires and t input wires is characterized by an associated FSM with state space $Q_D = \{0,1\}^n$, input space $I = \{0,1\}^t$, and output space $O = \{0,1\}^m$; the next state and output functions are defined by the corresponding logic. Note that each latch k has its own next state function $\delta_k : \{0,1\}^n \times \{0,1\}^t \rightarrow \{0,1\}$.

We will routinely refer to the states in Q_D as the states of the design. In the sequel, all FSM's will be assumed to be derived from hardware designs; we will use $\vec{i}$ to denote the input vector, $\vec{x}$ the present state vector, and $\vec{y}$ the next state vector. Define the transition relation of latch k to be the function $\mathbf{T}_k(\vec{x}, \vec{i}, y_k) = 1 \leftrightarrow \delta_k(\vec{x}, \vec{i}) = y_k$. Then for an FSM on n latches, the transition relation can be factored as follows:

$$\mathbf{T}(\vec{x}, \vec{i}, \vec{y}) = \prod_{k=1}^{n} \mathbf{T}(\vec{x}, \vec{i}, y_k) \quad (1)$$

It is routine to encode the FSM transition relation $\mathbf{T}$, the transition relation of the individual latches $\mathbf{T}_k$, as well as state sets $\mathbf{A}$ using BDD's. This approach has the advantage of compactly representing a large number of commonly encountered functions, thus combating state explosion. Furthermore, operations such as tautology checking, negation, conjunction, and existential quantification can be efficiently performed using BDD's.

Definition 2 Given an FSM with transition relation T, and a set of states A, the *image* of A is the set of states with characteristic function $\mathbf{Img}(\vec{y}) = (\exists \vec{x}, \vec{i})\,[\mathbf{T}(\vec{x}, \vec{i}, \vec{y}) \cdot \mathbf{A}(\vec{x})]$; the *pre-image* of A is the set of states with characteristic function $\mathbf{Pre}(\vec{x}) = (\exists \vec{y}, \vec{i})\,[\mathbf{T}(\vec{x}, \vec{i}, \vec{y}) \cdot \mathbf{A}(\vec{y})]$.

In [18] we provide a partial survey of the synthesis and verification algorithms that are built using the image and pre-image operations. These include computing local observability don't cares in combinational circuits, reached state computation, equivalent state computation, FSM equivalence checking, sequential ATPG, computing paths to fair cycles, and CTL model checking.

Reached state computation is the process of taking a designated initial state, and finding all states reachable from this state, i.e. all states s_r such that there is a sequence of inputs which lead the FSM into s_r from the initial state. Mathematically, this is performed by the following fixed point calculation:

$$\begin{aligned} \mathbf{A}_0(\vec{x}) &= \mathbf{Init}(\vec{x}) \\ \mathbf{A}_{k+1}(\vec{y}) &= \mathbf{A}_k(\vec{y}) \;+\; (\exists \vec{x}, \vec{i})\,[\mathbf{T}(\vec{x}, \vec{i}, \vec{y}) \cdot \mathbf{A}_k(\vec{x})] \end{aligned}$$

where $\mathbf{Init}$ is the characteristic function of the set consisting of the initial state. The limit of the $\mathbf{A}_k$'s is the characteristic function of the reached state set. Clearly, the core computation is an image computation.

Remark: Non-determinism is often used to describe *abstracted* systems (where a complex component has been replaced by a simpler one whose behavior contains the behavior of the original system), or under-specified systems (where decisions related to the actually implementation have not been made). Non-determinism can easily be modelled by adding new unconstrained inputs to get an equivalent deterministic FSM; furthermore, this can be done automatically. In [18]

we show that determining the least number of inputs to "determinize" non deterministic behavior is NP–complete, and describe a heuristic solution to this problem. The upshot is that there is no loss of generality in restricting our attention to deterministic systems.

3 Algorithms

In this section we present various BDD based algorithms to efficiently perform the core computations. In Section 3.1, techniques are discussed to achieve good BDD variable orderings. Section 3.2 presents the use of clustered transition relations. The approach used for ordering the clusters is detailed in Section 3.3.

To illustrate the effectiveness of these algorithms and to contrast them with some previous approaches, we use a set of seven benchmark examples. We perform reachability analysis on these examples (Table 1) to demonstrate the effectiveness of the algorithms for the image computations. Preliminary experiments indicate that these algorithms are efficient for inverse image computations as well. We will report a complete set of the results on inverse image computations in the final version of the work.

All examples were run on a DEC5900/260 workstation with 400MBytes memory. A limit of 10000 seconds of CPU time and 400MBytes of data size were used while running the experiments.

3.1 Ordering of BDD variables

As mentioned earlier, our symbolic verification algorithms use BDDs as the underlying data structure. The success of such algorithms depends critically on the size of the resulting BDDs, which is very sensitive to the variable ordering chosen. Given a logic function, the problem of finding the ordering that leads to a minimum sized BDD for the function is algorithmically intractable. Hence we need to apply some heuristics [3, 15, 20].

In the *dynamic reordering* scheme [19], the BDD package automatically reorders variables to minimize the total number of BDD nodes. Starting with a good heuristic ordering leads to better results. Since invoking dynamic reordering takes a significant amount of time, we found the following two parameters to be useful in controlling BDD size and improving computational efficiency. The first parameter, the "base value", is the total number of nodes in the BDD manager at which the reordering starts. The second parameter, the "increment value", is the amount by which the number of BDD nodes in the manager must increase between two successive invocations of reordering. These parameters can be chosen at the prompt and can be changed dynamically in the course of computation.

3.1.1 Results and Discussion

In our framework, we provide options for using ordering heuristics given in [3] and [20]. For our experiments we chose the heuristic in [3] as it was shown to outperform the other.

To demonstrate the effectiveness of dynamic ordering where the initial ordering is either random or based on a good heuristic, we performed some experiments.

We observe from Table 2 that for large examples, use of static ordering alone often leads to large BDD sizes. In the examples shown in the table, only one (2MDLC) could be completed using static ordering. The smaller BDD sizes for case C as compared to case B indicate that dynamic ordering should be used along with good heuristic initial ordering.

We also provide the ability to read in a manual ordering from a file. This feature especially becomes useful when we want to use the variable ordering previously generated by some heuristic or by dynamic reordering. As mentioned earlier, dynamic reordering is computationally expensive and thus bypassing it by using a previously generated ordering provides a significant computational advantage. Results shown in Table 2 indicate that using a previously generated ordering can achieve up to 10x speed improvement. However, we observe that for 2MDLC, time taken in case D is more than that taken in case A. This is because, in case D, the ordering is obtained after invoking dynamic reordering during reachability. This results in the smaller size for reached set representation (see column $|R|$). However, it need not be the best BDD ordering for transition relation representation and reachability computation.

In addition, we provide the ability to read in partial orders and heuristically complete them to obtain good initial orderings. Thus, if incremental changes are made to the design, the previous ordering can be adapted to be used for the updated design. This can substantially improve the dynamic reordering performance.

3.2 Use of Clustered Transition Functions

As described in §2, the transition relation of the design is a single BDD [3] which is the conjunction of the latch transition relations. As the complexity of the design grows, the size of this BDD often explodes. Hence computing the image and pre-image directly as given in definition 2 becomes infeasible for large designs.

The factorization of the transition relation given in equation 1 can be exploited to avoid building the complete transition relation. In particular, quantification of variables present in a subset of the conjoined terms can be performed iteratively.

A vector of BDD's is used in [10, 20]; each element of the vector represents the corresponding latch transition relation. Coudert [10] proposed reducing image computations to range

Example	# Latches	# Gates	Depth	# States	Description
sbc	28	927	10	154593	ISCAS'89 sequential benchmark (a snooping bus controller).
Gigamax	45	994	15	2.77×10^8	Cache coherency protocol description for hardware implementation of Gigamax distributed multiprocessor [16].
BDLC*	144	4775	103	1.66×10^{35}	Abstracted Byte Data Link Controller (BDLC);Manages the Tx-Rx protocol between microprocessor and a serial bus. Contains the abstract description of BIT module.
BDLC	172	6639	6998	2.85×10^{45}	Unabstracted version of the previous example.
2MDLC	83	2596	1006	65958	Two BIT modules interacting via serial bus using BDLC protocol.
BIU	154	3018	23	1.06×10^{37}	Abstracted version of a Bus Interface Unit from a commercial microprocessor.
Every	63	838	1279	8.33×10^8	Cache flush controller module of a commercial microprocessor.

Table 1: Benchmark examples used in this work.

Example	L	Different Cases									
		Case A			Case B			Case C			Case D
		$\|T\|$	$\|R\|$	Time	$\|T\|$	$\|R\|$	Time	$\|T\|$	$\|R\|$	Time	Time
2MDLC	83	3350	76158	155	*time out*	–	–	3350	8382	923	265
BDLC*	144	22122	*space out*	–	13110	8979	2558	12524	9454	1866	368
BDLC	172	*space out*	–	–	23501	*time out*	–	25379	*time out*	–	1515
BIU	154	36734	*time out*	–	13098	3271	3409	10471	4536	1779	93

Case A: Only static ordering performed.
Case B: Dynamic ordering performed with a random static ordering.
Case C: Both static and dynamic ordering performed.
Case D: Saved ordering file used.
L: Number of binary latches.
$|T|$: Shared BDD size of the transition relation.
$|R|$: BDD size of the reached set.
Time: Time in seconds to perform reachability.
space out : Exceeded the memory limit
time out : Exceeded the time limit

Table 2: Results for various ordering heuristics.

computations by exploiting the property of the constrain operator; the range computation is performed by recursive cofactoring. Efficiency comes from caching intermediate results and exploiting disjoint support. Touati [20] suggested a similar approach based on forming the product as a balanced binary tree. Image computation or pre-image computation is carried out iteratively using latch transition relations. As the number of latches in the system grows, the computation time increases.

Typically the BDD for each latch is small. In a design with a large number of latches, iteration over each latch, while being space efficient, is often time consuming. Forming clusters of the latches, while consuming slightly more space, can reduce the number of iterations substantially. In this section we develop this idea.

We group sets of latch transition relations together to form a vector of clustered transition relations. Suppose the original vector of transition relations corresponding to latches is given by $\mathbf{T}_k = \mathbf{T}_k(\vec{x}, \vec{i}, y_k)$ for $k = 1, 2, \ldots n$. Then the image of $\mathbf{A}(\vec{x})$ is given by,

$$\mathbf{Img}(\mathbf{A}(\vec{x})) = (\exists\, \vec{x},\, \vec{i})\, [\, \mathbf{A}(\vec{x}) \cdot \prod_k \mathbf{T}_k(\vec{x}, \vec{i}, y_k)\,] \qquad (2)$$

While forming clusters of latches, we take the product of the corresponding latch transition relations. If there are K clusters $C_1, C_2, \cdots C_k$ of latches, then the image computation can be equivalently written as,

$$\mathbf{Img}(A(\vec{x})) = (\exists\, \vec{x},\, \vec{i})\, [\, \mathbf{A}(\vec{x}) \cdot \prod_k \mathbf{T}_{C_i}\,] \qquad (3)$$

where $\mathbf{T}_{C_k} = \prod_{j \in C_k} \mathbf{T}_j(\vec{x}, \vec{i}, y_j)$.

In [4], Burch also proposed the use of clustered transition relations to represent circuits more efficiently. Latches were grouped together to form clusters but no automatic way to form clusters was given. Their technique possibly required user expertise, based on circuit structure.

3.2.1 Proposed Clustering Technique

In our approach the user specifies a limit on the BDD size of individual clusters (Partition Size Limit). The latch transition relations are ordered using one of the heuristics given in Section 3.3. Then the latch transition relations of latches are conjoined in this order until the product size surpasses the user specified limit. At this point the current cluster is complete and is stored in an array. Then, the clustering continues starting from the next latch.

3.2.2 Results and Discussion

Table 3 shows our results on clustering by BDD size.

We make the following observations: setting higher limits obviously leads to fewer clusters but the total number of BDD nodes taken by the clusters becomes bigger. From Equation [3], we observe that the image computation is performed by taking the product of cluster transition relations sequentially (we will refer to them as sequential iterations). The time taken in forming this product is a function of number of clusters as well as the cluster sizes. This results in the total CPU time being a convex function of partition size limit. Heuristically this can be reasoned as follows.

Using a limit of one yields a procedure which uses the least amount of space but results in maximum number of clusters (equal to the number of latches in the system) implying maximum number of sequential iterations. As the threshold is raised, the number of iterations is reduced, while BDD sizes of the operands increase. In the beginning, the reduction in the number of iterations offsets the increase in BDD sizes (and hence greater computation complexity). Hence initially run time is reduced as the cluster size is increased. But later, the BDD computation time starts to dominate the savings due to decreased number of iterations and we observe an increase in runtime. This is true for all the examples, except ones for which the design's transition relation is not very big (e.g. 2MDLC).

3.3 Ordering of Clustered Transition Relations

Since the system behavior is represented in terms of clusters of transition relations, the core verification operations (image and reverse image computation) are performed iteratively, one cluster at a time. Suppose $A(\vec{x})$ represents the set of states, and $T_k(\vec{x}, \vec{i}, \vec{y}_k)$ represents the transition relation of the k^{th} cluster; then the image of $\mathbf{A}(\vec{x})$ under the set of transition relations is mathematically given by,

$$\begin{aligned}\mathbf{Img}(\mathbf{A}(\vec{x})) \quad = \quad & (\exists\, \vec{x}, \vec{i})\, [\, \mathbf{A}(\vec{x}) \cdot \mathbf{T}_1(\vec{x}, \vec{i}, \vec{y}_1) \cdot \\ & \mathbf{T}_2(\vec{x}, \vec{i}, \vec{y}_2) \cdots \cdot \mathbf{T}_k(\vec{x}, \vec{i}, \vec{y}_k)\,]\end{aligned}$$

Since transition relations can be moved out of the scope of the existential quantification if they do not depend on any of the variables being quantified, for a given ordering of the transition relations, the above equation can be rewritten as,

$$\begin{aligned}\mathbf{Img}(\mathbf{A}(\vec{x})) \quad = \quad & (\exists\, \vec{x}_k, \vec{i}_k)\, [\, \mathbf{T}_k(\vec{x}, \vec{u}, \vec{y_k}) \\ & (\exists \vec{x}_{k-1}, \vec{i}_{k-1})\, \mathbf{T}_{k-1}(\vec{x}, \vec{u}, \vec{y_{k-1}}) \cdot \cdots \\ & (\exists \vec{x}_1, \vec{i}_1)\, \mathbf{T}_1(\vec{x}, \vec{u}, \vec{y}_1) \cdot \mathbf{A}(\vec{x})))]\end{aligned}$$

Coudert [10] proposed the recursive image computation. Touati [20] computes the image of a set of states by exploiting the property of the generalized cofactor in converting the

Partition Size Limit	Examples											
	2MDLC (L=83)			BDLC* (L=144)			BDLC (L=172)			BIU (L=154)		
	N	$\|T\|$	Time	N	$\|T\|$	Time	N	$\|T\|$	Time	N	$\|T\|$	Time
1	83	3216	588	144	4106	637	172	7042	5336	154	4021	118
100	20	3905	323	47	7089	354	54	12530	2489	52	11208	88
500	11	10220	236	21	13901	315	27	18641	1518	31	24175	83
1000	7	9057	219	14	18281	224	18	23332	1185	25	35447	84
2000	6	16785	279	11	26227	237	13	29904	1147	19	56351	143
5000	4	50947	627	8	43490	303	8	42263	885	16	107626	143
10000	3	55170	629	7	74727	428	6	70002	862	13	143758	163
20000	3	68432	706	6	131602	718	5	85639	878	12	283086	269
10000000	1	*time out*	–	1	*space out*	–	1	1153617	*time out*	1	*space out*	–

N: Number of partitions
L,$|T|$, Time: As in Table 2.

Table 3: Results on space-time trade off in clustering by the BDD size approach.

image computation into range computation given by

$$(\exists\, \vec{x},\ \vec{i})\ \left[\prod_k \mathbf{T}_{k\,A(\vec{x})}(\vec{x}, \vec{i}, \vec{y_k})\right]$$

where $\mathbf{T}_{k\,\mathbf{A}(\vec{x})}$ denotes the generalized cofactor of $\mathbf{T}_k(\vec{x}, \vec{i}, y_k)$ with respect to $\mathbf{A}(\vec{x})$. This range computation is performed using a balanced binary tree – leaves correspond to terms and variables at nodes of the tree that do not appear in the support of nodes elsewhere are existentially quantified. They reported better performance than [10].

Burch [4] criticized this approach on the grounds that generalized co-factor may introduce new variables in the supports of the terms, which delays the ability to quantify out variables. Heuristically, this would lead to larger BDD size of the intermediate product terms.

Note that if $\mathbf{T}_k(\vec{x}, \vec{i}, \vec{y_k})$ is conjoined with the product term obtained so far, it introduces at least $|\vec{y_k}|$ new variables (the corresponding next state variables). Heuristically the number of the variables getting existentially quantified from the product term and the number of additional variables getting introduced in the product term determine the computational efficiency of this operation. Thus the space requirement and the efficiency of image and pre-image computations become dependent on the order in which these clusters are processed. In [4], an ordering scheme of the partitioned transition relation is proposed and is based on the semantics of the underlying model. However, this requires detailed understanding of the semantics of the model and hence is not easily automated.

Geist *et al.* [11] give a simple automated way to order the relations when each relation consists of the next state function of a single latch. The primary criterion used is to choose the relation next in ordering for which maximum number of variables can be quantified out from the new product (unique variables belonging to that partition). In case of a tie, the relation with the maximum support is chosen.

Since, in our approach, clusters do not necessarily consist of a single latch, the ordering criteria should also take into account the number of next state variables introduced, while choosing the next cluster in the order. It was found that the maximum depth in the BDD ordering of any variable in a partition, referred to as the *index* of the variable, also affects the performance. The reasoning behind this is that existentially quantifying a variable from a function becomes computationally less expensive as the depth of the variable in the ordering increases.

3.3.1 Our Heuristic

In our heuristic, four different factors were used to decide the ordering of the partitions. We maintain two sets of clusters P and Q. The set P denotes the set of clusters which have already been ordered. This set is initialized as empty set. The set Q contains the clusters which are not yet ordered. For each cluster C_i in the set Q, we find the parameters as described below. In the following, PS, PI and NS denote the set of present state, primary input and next state variables respectively. A variable is denoted by v, $\mathcal{S}(T)$ represents the set of support variables of T and $\| A \|$ denotes the cardinality of the set A.

1. $v_{C_i} = \| \{ v \mid (v \in \mathcal{S}(T_{C_i})) \wedge (v \in PS \cup PI) \wedge (v \notin \mathcal{S}(T_{C_j})\ C_j \neq C_i, C_j \in Q) \} \|$, i.e. the number of variables which can be existentially quantified when T_{C_i} is multiplied in the product.

2. $w_{C_i} = \| \{ v \mid (v \in PS \cup PI) \wedge (v \in \mathcal{S}(T_{C_i})) \} \|$, i.e. the number of present state and primary input variables in the support T_{C_i}.

3. $x_{C_i} = \| \{ v \mid (v \in PS \cup PI) \wedge (v \in \mathcal{S}(T_{C_j}),\ C_j \in Q) \} \|$, i.e. the number of present state and primary input variables which have not yet been quantified.

4. $y_{C_i} = \| \{ v \mid (v \in \mathcal{S}(T_{C_i})) \wedge (v \in NS) \} \|$, i.e. the number of next state variables that would be introduced in the product by multiplying T_{C_i}.

5. $z_{C_i} = \| \{ v \mid (v \in NS) \wedge (v \in \mathcal{S}(T_{C_j}), C_j \in Q) \} \|$, i.e. the number of next state variables not yet introduced in the product.

6. $m_{C_i} = \max\{\text{index}(v), v \in \mathcal{S}(T_{C_i}) \wedge v \in (PI \cup PS) \}$, i.e., the maximum BDD index of a variable to be quantified in the support of T_{C_i}.

7. $M_{C_i} = \max\{m_{C_j}, C_j \in Q \}$, i.e. the maximum BDD index of a variable to be quantified out in the remaining clusters.

In order to normalize the effect of parameters 1,2,5 and 6, we form the following ratios.

1. $R^1_{C_i} = (v_{C_i}/w_{C_i})$.

2. $R^2_{C_i} = (w_{C_i}/x_{C_i})$.

3. $R^3_{C_i} = (y_{C_i}/z_{C_i})$.

4. $R^4_{C_i} = (m_{C_i}/M_{C_i})$.

Weights of W_1, W_2, W_3, and W_4 are attached to the above four factors respectively. The order of the clusters is obtained by greedily choosing the cluster with the best cost function value at each step. The chosen cluster is moved from set Q to set P and the process is repeated until all the sets are ordered (set Q becomes empty).

In our framework these weights can be interactively varied. We performed a series of experiments to find a good combination of these weights.

3.3.2 Results and Discussion

Table 4 compares the performance (CPU time in seconds) of our ordering heuristic with the heuristic proposed in [11]. Specifically we report the time taken in the reached state computation. The weights chosen after some experimentation in our heuristic were $W_1 = 2, W_2 = 1, W_3 = 1, W_4 = 1$.

The above results indicate that the proposed approach always outperforms that in [11]. Improvements up to 40% were achieved. The balanced binary tree approach to early quantification proposed in [20] does not explicitly give any ordering to the leaves of the tree, i.e. the latch transition relations. We found that using a random ordering for the leaves led to poor performance. However, the ordering schemes described in this section, coupled with [20]'s approach leads to performances that are usually (but not always) comparable to the results reported above.

	Various Heuristics	
Example	**CAV94**	**IWLS95**
BIU	151	84
Every	1958	1855
2MDLC	270	244
BDLC*	385	322
BDLC	2068	1973
Gigamax	4.6	3.7
sbc	93	52

Table 4: Comparison of CPU time (in seconds) for different cluster ordering heuristics.

4 Conclusion and Future Work

We described a series of algorithms for efficient synthesis and verification using BDD's. We argued that the core computation in synthesis and verification is that of forming the image and pre-image of a set of states under the transition relation characterizing the system. To make this step efficient, we addressed BDD variable ordering, use of partitioned transition relations, and use of clustering. The efficacy of these algorithms was demonstrated on a set of seven industrial examples ranging in size from 28 to 172 binary latches.

These algorithms are an integral part of a second generation BDD based tool (HSIS-II) for both logic synthesis and formal design verification using either model checking or language containment. The input is an enhanced version of Verilog which is compiled to a hierarchical netlist [1]. This is determinized and read into a network of latches and gates. The algorithms described in this paper are integrated into the tool which is aimed at users of synthesis and verification as well as developers interested in creating their own applications on top of the efficient core computation routines provided.

Our routines use several parameters (default or user specified). It is likely that no universal choice of settings will yield the best results for all examples. Hence the ability to set parameters at the prompt is provided; further experiments possibly will lead to a general purpose robust script for novice users.

Other BDD based techniques which look promising in-

clude the "exists–cofactor" of [6], and the "implicitly conjoined invariants" of [13]. We plan to experiment with them since it should be relatively easy with the data structure proposed in this paper to implement these methods. We also intend to experiment with the functional and structural approaches for automatic state variable clustering [8]. Certain limitations of BDD based formal design verification can not be solved by the techniques described in this work. For example, the size of the reached set may be large under any variable ordering. Other data structures like GBDDs, XBDDs, ZBDDs [14] might be useful in these cases. There are also a wide class of heuristics for coping with state explosion that are orthogonal to the approaches we have taken, such as property specific reductions [2], abstractions [12], and conservative approximations to reached state sets [7]. We believe these techniques can be conveniently developed in our framework and then tested and compared on realistic examples.

5 Acknowledgements

We would like to thank Thomas R. Shiple and the reviewers for their helpful and constructive comments.

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 the Design Automation Conf.*, pages 454–459, June 1994.

[2] A. Aziz, T. R. Shiple, V. Singhal, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Formula-Dependent Equivalence for Compositional CTL Model Checking. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 324–337. Springer-Verlag, 1994.

[3] A. Aziz, S. Tasiran, and R. K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *Proc. of the Design Automation Conf.*, San Diago, CA, June 1994.

[4] J. R. Burch, E. M. Clarke, and D. E. Long. Representing Circuits More Efficiently in Symbolic Model Checking. In *Proc. of the Design Automation Conf.*, June 1991.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proc. of the Design Automation Conf.*, June 1990.

[6] G. Cabodi and P. Camurati. Exploiting Cofactoring for Efficient FSM Symbolic Traversal Based on the Transition Relation. In *Proc. Intl. Conf. on Computer Design*, pages 299–313, Oct. 1993.

[7] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for Approximate FSM Traversal. In *Proc. of the Design Automation Conf.*, pages 25–30, June 1993.

[8] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. A Structural Approach to State Space Decomposition for Approximate Reachability Analysis. In *Proc. Intl. Conf. on Computer Design*, Oct. 1994.

[9] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, June 1989.

[10] O. Coudert and J. C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 126–129, Nov. 1990.

[11] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1994.

[12] S. Graf. Verification of a Distributed Cache Memory by Using Abstractions. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 207–219. Springer-Verlag, 1994.

[13] A. J. Hu, G. York, and D. L. Dill. New Techniques for Efficient Verification with Implicitly Conjoined BDD's. In *Proc. of the Design Automation Conf.*, pages 276–282, June 1994.

[14] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi. Extended BDD's: Trading off Canonicity for Structure in Verification Algorithms. In *Proc. Intl. Conf. on Computer-Aided Design*, 1991.

[15] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi. Variable Ordering for FSM Traversal. In *Proc. Intl. Conf. on Computer-Aided Design*, 1991.

[16] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[17] C. Pixley. A Computational Theory and Implementation of Sequential Hardware Equivalence. In E. M. Clarke and R. P. Kurshan, editors, *Proc. of the Workshop on Computer-Aided Verification*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 293–320. American Mathematical Society, June 1990.

[18] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient Formal Design Verification: Data Structure + Algorithms. Technical Report UCB/ERL M94, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Oct. 1994.

[19] R. Rudell. Dynamic Variable Ordering for Binary Decision Diagrams. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 42–47, Nov. 1993.

[20] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 130–133, Nov. 1990.