Copyright © 1994, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

EFFICIENT FORMAL DESIGN VERIFICATION:
DATA STRUCTURE + ALGORITHMS

by

Rajeev K. Ranjan, Adnan Aziz, Robert K. Brayton,
Bernard Plessier, and Carl Pixley

Memorandum No. UCB/ERL M94/100

15 October 1994

EFFICIENT FORMAL DESIGN VERIFICATION: DATA STRUCTURE + ALGORITHMS

by

Rajeev K. Ranjan, Adnan Aziz, Robert K. Brayton,
Bernard Plessier, and Carl Pixley

Memorandum No. UCB/ERL M94/100

15 October 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

EFFICIENT FORMAL DESIGN VERIFICATION: DATA STRUCTURE + ALGORITHMS

by

Rajeev K. Ranjan, Adnan Aziz, Robert K. Brayton,
Bernard Plessier, and Carl Pixley

Memorandum No. UCB/ERL M94/100

15 October 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Efficient Formal Design Verification: Data Structure + Algorithms

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 data structure and a set of BDD based algorithms for efficient formal design verification. We argue that hardware designs should be translated into an intermediate hierarchical netlist of combinational tables and sequential elements, and internally represented by a flattened network of gates and latches, akin to that in SIS [32]. We establish that the core computation in BDD based formal design verification is forming the image and pre-image of sets of states under the transition relation characterizing the design. To make this step efficient, we address BDD variable ordering, use of partitioned transition relations, use of clustering, use of don't cares, and redundant latch removal. Many of these techniques have been studied in the past. We provide a complete integrated set of modified 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.

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

1 Introduction

In the design of digital systems, there are two different levels of verification, corresponding to two major phases of design. In the first phase an initial specification in a high-level description language, like VHDL or Verilog is used. *Design verification* is concerned with the question "Is what I specified what I wanted?". The second phase is synthesizing the initial specification into a circuit which can be implemented. *Implementation verification* is concerned with the question "Is what I synthesized what I specified?".

The traditional approach to design verification is simulation, which is well-understood and has been applied widely in the design community. Designers are comfortable with simulation because thinking in terms of input patterns and expected output patterns is intuitive. However, exhaustive simulation is not feasible for even moderately sized systems. Thus, simulation has a serious drawback; it cannot show the absence of errors, only the presence of errors.

Formal design verification, is the process of mathematically proving that a system possesses a set of properties. The theory behind this approach has been investigated over the last three decades, but only in the last five years have practical tools begun to emerge. Though it overcomes many of the drawbacks of simulation [1], it is currently limited to relatively small designs. We seek to improve the efficiency of verification so that larger designs can be verified.

A detailed survey of the various verification methods can be found in [22]. The first to be used in design verification was *theorem proving*, which usually requires extensive interaction with human experts. *Language containment* (LC) and *model checking* (MC) are two recent *automated* approaches for verifying properties of designs described by state transition systems. In LC, the system and the properties are both specified as ω-automata. The verification problem is equivalent to verifying that the language of the system is contained in the language of the properties (COSPAN [23]). In MC, the properties are specified using temporal logic, and *model checking* is applied to the system specification to verify these properties [17]. Verification tools that manipulate state-based systems explicitly are limited by the size of the state space (Murϕ [16]). However, most real designs consist of a set of interacting components leading to the problem of state explosion. In the context of implementation verification, Coudert and Madre [15] pioneered the use of BDDs to implicitly manipulate the product state space. Since then, the use of BDDs has been extended to manipulate transition systems in the area of design verification (SMV [29]).

In this paper we describe a data structure and a set of algorithms for efficient BDD based formal design verification. We argue that hardware designs can (and should) be mapped into netlists of deterministic (possibly multi-valued) gates and latches. At early stages of the design process, descriptions often contain non-determinism which can be equivalently viewed as coming from unconstrained external inputs. These netlists can be internally represented by a SIS [32] like data structure, on which a large body of BDD techniques developed in SIS, for operations like sequential equivalence and optimization e.g. [10, 26, 30, 33], can be applied. This data structure also provides a common and somewhat familiar software development environment for developers wanting to write their own applications.

We argue that the core computation in BDD based formal design verification is that of forming the image and pre-image of a set of states under the transition relation characterizing the system. In order to make this step as efficient as possible, we address the following:

- Variable ordering techniques.
- Use of partitioned transition relations (keeping the components separate to avoid the large BDD size of the monolithic transition relation for the entire system).
- Use of clustering (grouping together some parts of the design to reduce the number of iterations required for each image and inverse image computation).
- Ordering of the clustered transition relation for efficient image and pre-image computation.

- Use of don't cares in minimizing BDD's.
- Removal of redundant latches.

Many of these approaches have been studied in the past. References and comparisons with previous work are given with the details of our techniques. One salient feature is that all our algorithms are completely automatic. The methods are interdependent and can be used together in various combinations. We report on a subset of these combinations and experimental results on a benchmark set consisting of seven relatively large industrial designs.

The algorithms described in this paper 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. However, an advanced user can exploit the core computation routines for writing new applications possibly using different parameter settings.

The paper is structured as follows: in Section 2 the CTL model checking and language containment paradigms are described. We also introduce the state explosion problem, which motivates the need for symbolic techniques for state enumeration.

Section 3 presents our data structure used to represent hardware designs, which is well suited for formal design verification.

Efficient BDD methods for image and inverse image computations are described in Section 4. We conclude in Section 5 by commenting on some implications of these results and indicating future directions.

2 Preliminaries

2.1 Image and Inverse Image Computations

B represents the boolean set $\{0,1\}$. The following definitions pertain to a finite state system M with n state bits and m inputs.

Definition 1 *$T(\vec{x},\vec{i},\vec{y}) : B^n \times B^m \times B^n \rightarrow B$ is the* transition relation *of a system with n state bits and m inputs. $T(\vec{x},\vec{i},\vec{y}) = 1$ implies that in state $\vec{x}$ there exists a transition to state $\vec{y}$ on input $\vec{i}$.*

Definition 2 *Let $T(\vec{x},\vec{i},\vec{y}) : B^n \times B^m \times B^n \rightarrow B$ be a transition relation and P a subset of B^n. The* image *of P under the transition relation T is the set Q, such that,*

$$\vec{y} \in Q \quad \Leftrightarrow \quad \exists \vec{x} \in P,\ \exists \vec{i} \ s.t.\ T(\vec{x},\vec{i},\vec{y}) = 1 \tag{1}$$

Definition 3 *Let $T(\vec{x},\vec{i},\vec{y}) : B^n \times B^m \times B^n \rightarrow B$ be a transition relation and P a subset of B^n. The* inverse image *(also called* pre-image*) of P under the transition relation T is the set Q, such that,*

$$\vec{x} \in Q \quad \Leftrightarrow \quad \exists \vec{y} \in P, \exists \vec{i} \ s.t.\ T(\vec{x},\vec{i},\vec{y}) = 1 \tag{2}$$

Definition 4 *Let $T(\vec{x},\vec{i},\vec{y}) : B^n \times B^m \times B^n \rightarrow B$ be a transition relation and $I(\vec{x})$ be the set of initial states of the system. The set of reachable states of the system, $R(\vec{y})$ is the least fixed point of*

$$\begin{aligned} R_0(\vec{y}) &= I(\vec{y}) \\ R_{k+1}(\vec{y}) &= R_k(\vec{y}) \cup \exists \vec{x}, \vec{i}\, [\, R_k(\vec{x}) \wedge T(\vec{x},\vec{i},\vec{y})\,] \end{aligned} \tag{3}$$

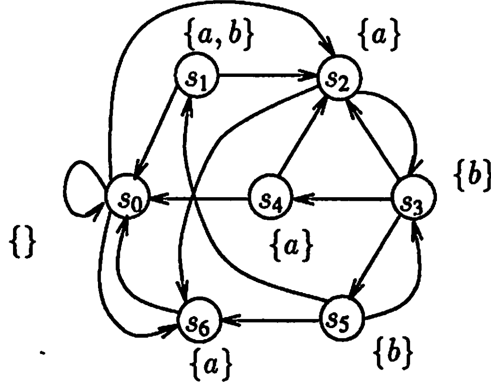

Figure 1: An example illustrating a Kripke structure. $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$, $AP = \{a, b\}$. An edge from state s_i to s_j indicates that $(s_i, s_j) \in T$. States are labelled with the subset of APs true at the state. A path through K is a sequence of states $\sigma_1, \sigma_2, \cdots$ such that $\forall\, i\, (\sigma_i, \sigma_{i+1}) \in T$.

2.2 Formal Design Verification

In this section we informally describe the CTL model checking and language containment approaches to formal design verification. In both cases the underlying design is characterized by a *Kripke structure* [14].

Definition 5 A *Kripke structure* $\mathcal{K}$ is a triple $(S, T, \mathcal{L})$, where S is a finite set of *states*, $T \subset S \times S$ is the *transition relation*, and $\mathcal{L} : AP \rightarrow 2^S$ is the labelling function mapping *atomic propositions (AP)* to sets of states. A pictorial representation of a Kripke structure is given in Figure 1.

In the CTL model checking paradigm [17], properties are expressed as formulas from an inductively defined syntax. Truth of the formulae is interpreted over states in Kripke structures; determining the truth value of a formula over a state in the structure is referred to as model checking and can be algorithmically performed using fixed point calculations. Precise syntax and semantics are given in [17]. As an example, state s_0 in the Kripke structure of Figure 1 models the formula $EF\,(a \wedge b)$ ("there exists a path to a state where both a and b hold"). This is because s_1 is labelled by a and b, and there is a path from s_0 to s_1, namely $s_0 \rightarrow s_2 \rightarrow s_3 \rightarrow s_5 \rightarrow s_1$. This result can be mathematically obtained by finding the least fixed point of

$$\begin{aligned} R_0(\vec{x}) &= p \\ R_{k+1}(\vec{x}) &= R_k(\vec{x}) \cup EX\, R_k(\vec{x}) \end{aligned} \tag{4}$$

where p denotes the set of states which satisfy the formula $(a \wedge b)$, i.e. set of states labelled with "a" and "b". Note that the set of states satisfying $EX\ R(\vec{x})$, i.e. the set of states that can reach some states in $R(\vec{x})$ in one step, can be found by computing the *inverse image* of $R(\vec{x})$, with respect to the transition relation. Similarly, for some other CTL formula we need to perform *image* computations.

In the language containment paradigm, the design is identified by the set of generated output traces $\mathcal{L}_D$, and a property is given by a set of acceptable traces $\mathcal{L}_P$. Verification consists of checking whether all design behavior is acceptable i.e. checking $\mathcal{L}_D \subset \mathcal{L}_P$, which in turn is equivalent to checking that $\mathcal{L}_D \cap \overline{\mathcal{L}_P}$ is empty. Kurshan [23] observed that for certain classes of properties (namely deterministic L-automata) the set $\overline{\mathcal{L}_P}$ is efficiently computable. In simple terms, verification consists of finding a path in a Kripke structure which starts at the initial state and leads to a fair cycle i.e. a cycle which includes at least one state from a designated subset of *fair* states $\mathcal{F}$ [18]. Conceptually,

this check may be performed by first finding the set of states $\mathcal{F}^*$ which given reach a fair cycle. Thus the property fails if and only if the initial state lies in $\mathcal{F}^*$ (since we want $\mathcal{L}_D \cap \overline{\mathcal{L}_P} = \phi$, i.e. no fair cycles).

Suppose $R_\infty(\vec{x})$ represents the set of reachable states. Limit the transition relation to the set of reachable states by $T(\vec{x}, \vec{y}) = T(\vec{x}, \vec{y}) R_\infty(\vec{x})$.

The algorithm to find set of states $\mathcal{F}^*$ is as follows:

1. Initialize $F_0(\vec{y}) = \mathcal{F}(\vec{y})$.

2. Compute A_∞ using following fixed point computation:

$$\begin{aligned} A_0(\vec{y}) &= F_0(\vec{y}) \\ A_{k+1}(y) &= A_k(\vec{y}) \ \cap \ \exists x\,[A_k(\vec{x}) \wedge T(\vec{x}, \vec{y})] \end{aligned} \quad (5)$$

   $A_\infty(\vec{y})$ gives the set of states which can be reached by some states of $F_o(\vec{y})$ which lie on a cycle.

3. Compute B_∞ using following fixed point computation:

$$\begin{aligned} B_0(\vec{x}) &= A_\infty(\vec{x}) \\ B_{k+1}(x) &= B_k(\vec{x}) \ \cap \ \exists \vec{y}\,[B_k(\vec{y}) \wedge T(\vec{x}, \vec{y})] \end{aligned} \quad (6)$$

   $B_\infty(\vec{x})$ gives the set of states which can reach some states of $A_\infty(\vec{x})$ which lie on a cycle.

4. Compute $F_0(\vec{x}) = A_\infty(\vec{x}) \cap B_\infty(\vec{x})$.

5. Repeat (2-4) until convergence.

6. $\mathcal{F}^*$ is given by the least fixed point of

$$\begin{aligned} C_0(\vec{x}) &= F_0(\vec{x}) \\ C_{k+1}(\vec{x}) &= C_k(\vec{x}) \ \cup \ \exists \vec{y}\,[T(\vec{x}, \vec{y}) \wedge C_k(\vec{y})] \end{aligned} \quad (7)$$

It is apparent from Equations [4,5,6,7] that the core computation in verification is that of taking the image or inverse image of sets of states under the transition relation.

2.3 State Explosion

Often designs are constructed by linking components together; unspecified inputs are assumed to take any value on each clock cycle. The synchronous product of components defines a single Kripke structure (also referred to as the *product machine*), the state space of which is the product of the components' state spaces. Hence algorithms that directly manipulate states will have time and space complexity that is exponential in the size of the system description. Indeed, the computational complexity of this problem is known to be PSPACE–complete [3]. The complexity introduced by concurrent interaction is popularly referred to as the "state explosion problem". The quest for heuristic solutions to this problem constitutes the forefront of research in formal verification [2, 9, 13, 15, 21].

Binary Decision Diagrams (BDDs) [7] are canonical representations of Boolean functions on which Boolean operations can be performed efficiently. Furthermore, they can compactly represent a wide variety of commonly encountered functions. Transition relations and sets of states can be represented using BDDs of their characteristic functions, which can be used for efficient fixed-point computations [9, 15, 30]. BDDs are now extensively used for both design and implementation verification of hardware systems and many non-trivial design examples have been verified using BDDs [11, 29]. Still, there are many instances of medium sized circuits that cannot be verified using existing BDD techniques. In Section 4 we provide a partial survey of state-of-the-art BDD techniques and present our contributions.

3 Data Structure

Designers typically specify systems in a high level language which supports constructs like integer arithmetic, multi-valued variables, array structures etc. (e.g. SMV [29] or enhanced Verilog [12]). The description of the system often contains non-determinism typically introduced when some part of the design is abstracted by hiding details. Non-determinism also comes into play while modeling the behavior of the environment. Our approach to *non-determinism* is to add new unconstrained inputs. We convert an arbitrary system description into a deterministic netlist of gates and latches. For example, consider the following *Verilog* description of an *xor* gate with arbitrary delay. The output of the gate is made non-deterministic to model the an arbitrary delay.

```
module xor( clk, inp, out )
input clk, inp;
input ready;
output out;
reg out;

initial out = 0;
always (@posedge clk) begin
       if (ready)
          begin
               out <= [[out, out^inp]];/* next state of out is non-deterministically */
          end                          /* either out or out ⊕ inp*/
       else
          out <= 0;
end
endmodule
```

We use a new, unconstrained binary valued variable `_$nd0` to convert this description into *deterministic* netlist of gates and a latch as given in Figure 2.

It can be shown that the "determinized" design yields a Kripke structure that is bisimilar [6] to that of the original design, and can be safely used in place of the original.

This "determinization" can be done automatically. Suppose $\vec{y} = (y_1\, y_2 \ldots y_m)$ is a non-deterministic function of $\vec{x} = (x_1\, x_2 \ldots x_n)$, where all component variables take binary values. Thus $\vec{y}$ can be expressed by a non-deterministic multi-output table, as illustrated in the following example:

x1	x2	→	y1	y2
-	0		0	1
1	-		1	0
0	1		0	0

Thus, for example, for $\vec{x} = 10$, $\vec{y}$ can be either 01 or 10. We add an unconstrained input d to this table. The corresponding deterministic table is given as follows:

d	x1	x2	→	y1	y2
0	-	0		0	1
1	1	-		1	0
-	0	0		0	1
-	1	1		1	0
-	0	1		0	0

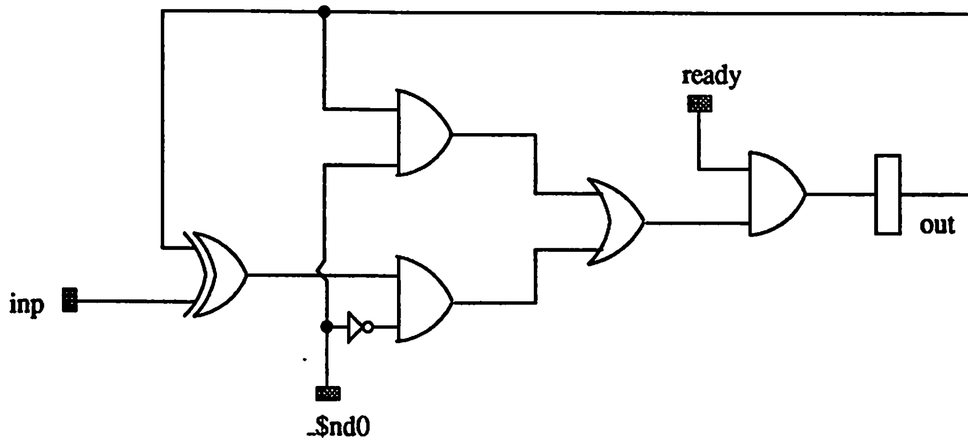

Figure 2: A deterministic netlist of gates and latches that represents the functionality of the non-deterministic Verilog module `xor`. The variable `_$nd0` is a new primary input to the system.

Determinizing with the least number of additional external inputs is equivalent to solving the minimum coloring problem for undirected graphs.

Lemma 3.1 It is NP-complete to decide if *K* binary variables suffice to "determinize" a table.

Proof: Membership in NP: Note that checking a table for determinism is easy: form the pairwise intersection of the cubes in the input space – any pair with non empty intersection should agree on the corresponding outputs. Hence the *K* new variables along with the assignment can be guessed; the resultant table can be checked in polynomial time.

NP-hardness We use a reduction from graph coloring [19]. Let $G = (V = \{v_0, v_1, \ldots, v_{n-1}\}, E)$ be a graph and $K \leq n$ be a positive integer. Suppose $k = \lceil \log_2 n \rceil$. Define a non-deterministic function ∇ on input space $(x_1, x_2, \ldots, x_k)$ and output space $(y_1, y_2, \ldots, y_k)$. The corresponding table is defined on the minterms of the input space by the equation $\nabla(x) = \{y \mid (v_{\|x\|}, v_{\|y\|}) \in E\}$, where $\| x \|$ gives the decimal value of the binary number represented by the vector $(x_1, x_2, \ldots, x_k)$, similarly $\| y \|$. For a non-trivial graph with at least one vertex with degree more than one, the corresponding table is non-deterministic. ∇ can be determinized by adding a single multi-valued variable D to the input space; identify a valid $\mid D \mid$ coloring of the graph by the values taken by D at the corresponding input minterm, i.e. all nodes whose corresponding minterm have the same D value are given the same distinct color.

In fact, since deciding if a 4 coloring exists is already complete, finding the least number of variables to determinize a table with as few as two binary outputs is already intractable. ∎

A similar argument, run in reverse, demonstrates that finding the minimum number of binary variables needed to determinize a table easily reduces to the graph coloring problem for which good heuristics exist, e.g. complementing the graph and covering it by cliques etc.

After determinization, the design is translated into a hierarchical netlist of deterministic, single output gates and latches. Internally, we read the netlist into a flattened graph representation. The nodes of the graph indicate combinational logic blocks and sequential elements, i.e. latches. Nodes may be annotated with information about the original hierarchy that certain partitioning/clustering applications might find useful. Those familiar with the sequential synthesis tool SIS will immediately see the similarity with the SIS or BLIF representation of sequential logic circuits.

We chose such a data structure because it allows easy access to the wide range of BDD applications written for SIS, such as variable ordering and reachability analysis [25, 26, 33]. The wide acceptance of SIS as a framework for developing new applications suggests that our data structure may have similar applicability.

This approach has the advantage over the one used in the initial version of HSIS [1] in that the number of variables initially needed to build the BDDs representing the system is drastically reduced; only the present state, next state, and primary input variables are needed. Also functional composition (like in SIS) can be used to derive the next state and output functions, whereas currently in HSIS relational composition is needed because internal variables can be non-deterministic. Furthermore, the next state function of each latch can be specified separately, since the correlation between next states is carried by the newly created inputs. Thus, the expression for the core computation of finding the image of a set of states $A(\vec{x})$ (Equation 1) can be given by,

$$Image(A(\vec{x}))(\vec{y}) \quad = \quad \exists \vec{x}, \vec{u} \, [A(\vec{x}) \cdot \prod_{i=1}^{n} (y_i \equiv f_i(\vec{x}, \vec{u}))]$$

where y_i is the *next state variable* of latch i, $f_i(\vec{x}, \vec{u})$ is the *next state function*, $\vec{x}$ is the *present state vector* and $\vec{u}$ is the *primary input vector*. The transition relation $T_i(\vec{x}, \vec{u}, y_i)$ of the i-th latch, by definition is the relation $(y_i \equiv f_i(\vec{x}, \vec{u}))$. In Section 4 we describe how to evaluate this expression efficiently using BDDs by choosing an appropriate variable ordering, forming the product of $A(x)$ incrementally with a heuristically chosen permutation of the T_i's, and clustering the T_i's into sets C_k to form a smaller set of transition relations T_{C_k}. Although we do not pursue it in this paper, the fact that we have a next state function instead of a relation also allows experimentation with the method proposed in [15].

4 Algorithms

In this section we present various BDD based algorithms to efficiently perform the core verification computations. In Section 4.1, techniques are discussed to achieve good BDD variable orderings. Section 4.2 presents the use of clustered transition relations. The approach used for ordering the clusters is detailed in Section 4.3. In Section 4.5, we describe a fast technique for removing some redundant latches.

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* (Equation 3) 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 440MBytes memory. A limit of 10000 seconds of CPU time and 400MBytes of data size were used while running the experiments.

4.1 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 [4, 26, 33].

In the *dynamic reordering* scheme [31], 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

Example	# Latches	# Gates	Description
sbc	28	927	ISCAS'89 sequential benchmark (a snooping bus controller).
Gigamax	45	994	Cache coherency protocol description for hardware implementation of Gigamax distributed multiprocessor [29].
BDLC*	144	4775	Abstracted Byte Data Link Controller (BDLC); Manages the transmit-receive protocol between microprocessor and a serial bus. Contains the abstract description of BIT module. Part of a commercial chip.
BDLC	172	6639	Unabstracted version of the previous example.
2MDLC	83	2596	Two BIT modules interacting via a serial bus using BDLC protocol.
BIU	154	3018	Abstracted version of a Bus Interface Unit from a commercial microprocessor.
Every	63	838	Cache flush controller module of a commercial microprocessor.

Table 1: Benchmark examples used in this work.

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.

4.1.1 Results and Discussion

In our framework, we provide options for using ordering heuristics given in [4] and [33]. For our experiments we chose the heuristic in [4] 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 3 indicate that using a previously generated ordering can achieve up to 10x speed improvement.

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.

4.2 Use of Clustered Transition Functions

The two most common methods of representing the transition relation of a system are the following:

Example	L	Different Cases								
		Case A			Case B			Case C		
		$\|T\|$	$\|R\|$	Time	$\|T\|$	$\|R\|$	Time	$\|T\|$	$\|R\|$	Time
2MDLC	83	3894	76158	204	21841	17350	1436	3894	13919	1552
BDLC*	144	52757	*S.O.*	-	36577	7069	1167	27000	5607	1495
BDLC	172	79308	*S.O.*	-	39935	*S.O.*	-	25250	86080	6970
BIU	154	37989	*S.O.*	-	31829	4614	3912	36626	2930	1635

Case A: Only static ordering performed.
Case B: Dynamic ordering performed with a random static ordering.
Case C: Both static and dynamic ordering performed.
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.
S.O.: Space out.

Table 2: Results for various ordering heuristics.

Example	Case C	Case D
2MDLC	1552	180
BDLC*	1495	114
BDLC	6970	2048
sbc	127	107
BIU	1635	270

Case C: Time to perform reachability without using saved ordering file (in secs). Same as case C in Table 2.
Case D: Time to perform reachability with saved ordering file (in secs)

Table 3: Results showing effectiveness of using saved ordering files.

Monolithic Transition Relation: The transition relation of the system is represented by a single BDD [9] which is the conjunction of the transition relations of the individual latches. As the circuit complexity grows, the size of the transition relation usually explodes. Hence this approach becomes infeasible for large, complex circuits.

Partitioned Transition Relation: A vector of transition relations is used [15, 33]; each element of the vector represents the next state relation for a latch. Coudert [15] proposed reducing image computations to range computations by exploiting the property of the constrain operator; the range computation is performed by recursive co-factoring. Efficiency comes from caching intermediate results and exploiting disjoint support. Touati [33] 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 transition relations for individual latches. Reasoning heuristically, as the number of latches in the system grows, the computation time increases.

A simple extension of these two approaches overcomes some of their shortcomings.

We represent the transition relation of the system by a vector of *clustered transition relations*. First, the next state relation of each latch is computed. Next, a group of transition relations are clustered together to form a vector of clustered transition relations. The idea is illustrated below.

Suppose the original vector of transition relations corresponding to latches is given by $T_i = T_i(\vec{x}, \vec{u}, y_i)$ for $i = 1, 2, \ldots n$. Then the image of $A(\vec{x})$ is given by,

$$Image(A(\vec{x})) = \exists \vec{x}, \vec{u} [A(\vec{x}) \prod_i T_i(\vec{x}, \vec{u}, y_i)] \tag{8}$$

While forming clusters of latches, we take the product of the corresponding 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,

$$Image(A(\vec{x})) = \exists \vec{x}, \vec{u} [A(\vec{x}) \prod_{i=1}^{i=K} T_{C_i}] \tag{9}$$

where $T_{C_i} = \prod_{j \in C_i} T_j(\vec{x}, \vec{u}, y_j)$.

In [8], 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.

4.2.1 Proposed Clustering Technique

In our approach the user specifies a limit on the BDD size of individual clusters (Partition Size Limit). The next state relations of latches are ordered using one of the heuristics given in Section 4.3. Then the next state 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.

4.2.2 Results and Discussion

Table 4 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 [9], we observe that the image computation is performed by taking the product of transition relation of clusters 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 total CPU time being a convex function of partition size limit. Intuitively 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 monolithic transition relation is not very big (e.g. 2MDLC).

4.3 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

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	2744	728	144	5114	432	172	7042	9248	154	3943	564
100	16	2943	348	49	8454	235	57	13259	3905	48	9950	430
1000	5	3434	203	14	19613	125	20	24966	2014	18	30511	266
2000	3	3238	171	11	27762	115	14	34774	1662	15	35746	245
5000	2	6612	167	8	44033	106	8	46994	1443	10	76563	228
10000	1	6853	142	6	56410	106	6	61704	1243	9	180914	227
20000	–	–	–	5	88984	116	5	90179	1121	7	217395	171
30000	–	–	–	4	111867	129	4	99020	1185	7	284450	198

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

Table 4: Results on space-time trade off in clustering by the BDD size approach.

the set of states, and $T_i(\vec{x}, \vec{u}, \vec{y_i})$ represents the transition relation of the ith cluster; then the image of $A(\vec{x})$ under the set of transition relations is mathematically given by,

$$Image(A(\vec{x})) = \exists\, \vec{x}, \vec{u}\ \left[\, A(\vec{x})\ \wedge\ T_1(\vec{x}, \vec{u}, \vec{y_1})\ \wedge\ T_2(\vec{x}, \vec{u}, \vec{y_2})\ \wedge\ \ldots\ \wedge\ T_k(\vec{x}, \vec{u}, \vec{y_k}) \right]$$

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,

$$Image(A(\vec{x})) \;=\; \exists\, \vec{x}_k, \vec{u}_k\ (\, T_k(\vec{x}, \vec{u}, \vec{y_k}) \wedge (\exists \vec{x}_{k-1}, \vec{u}_{k-1}\ T_{k-1}(\vec{x}, \vec{u}, \vec{y}_{k-1}) \wedge \cdots \wedge (\exists \vec{x}_1, \vec{u}_1\ T_1(\vec{x}, \vec{u}, \vec{y_1}) \wedge A(\vec{x}))))$$

Coudert [15] proposed the recursive image computation. Touati [33] computes the image of a set of states by exploiting the property of the generalized cofactor in converting the image computation into range computation given by

$$\exists\, \vec{x},\ \vec{u}\ \left[\prod_{i=1}^{i=k} T_{i\,A(\vec{x})}(\vec{x}, \vec{u}, \vec{y_i})\right]$$

where $T_{i\,A(\vec{x})}$ denotes the generalized cofactor of $T_i(\vec{x}, \vec{u}, y_i)$ with respect to $A(\vec{x})$. 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 [15].

Burch [8] 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 $T_i(\vec{x}, \vec{u}, \vec{y_i})$ is conjoined with the product term obtained so far, it introduces $|\vec{y_i}|$ new variables (the corresponding next state variables). We heuristically argue that the number of the variables getting existentially quantified from the product term and the number of 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 [8], 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.* [20] 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.

4.3.1 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 new 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 out 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 qunatified 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 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.

4.3.2 Results and Discussion

Table 5 compares the performance (CPU time in seconds) of our ordering heuristic with the heuristics proposed in [20, 33]. Specifically we report the time taken in the reached state computation. The weights chosen after some experimentation in our heuristic were $W_1 = 2, W_2 = 1, W_3 = 1, W_4 = 1$.

Example	Various Heuristics		
	[33]	[20]	Proposed
BIU	305	326	315
Every	6087	5857	5788
2MDLC	176	244	179
BDLC*	140	191	144
BDLC	*space out*	3023	2231
Gigamax	4.8	7.4	4.8
sbc	116	135	118

Table 5: Comparison of CPU time (in seconds) for different cluster ordering heuristics.

The above results indicate that the proposed approach always outperforms that in [20]. Improvements up to 25% were achieved.

Although in some examples (BIU, BDLC*, 2MDLC, sbc) Touati's heuristic [33] performs marginally better than ours, on BDLC, Touati's approach ran out of memory.

4.4 Don't Cares

Don't care points arise naturally in the context of BDD based formal verification. For example, consider the following fixed point computation for finding the set of reached states:

$$\begin{aligned} A_0(x) &= \text{Init}(x) \\ A_{k+1}(y) &= A_k(y) \cup \exists \vec{x}\, [T(x,y) \wedge A_k(x)] \end{aligned} \tag{10}$$

Define the frontier states at iteration i to be $A_i \setminus A_{i-1}$ where $A_{-1} = \{\}$. Coudert [15] observed that any set of states between A_i and $A_i \setminus A_{i-1}$ can be used in place of A_i in Equation 10 while preserving the result for A_{i+1}. Thus A_{i-1} can be used as a don't care set to minimize the BDD size of the frontier states. Similarly, the transition relations can be simplified with respect to the set of states whose image is being computed. In our framework we provide the option of using don't cares arising in this manner.

Conservative approximations to the unreached states also yield don't cares. Given a set of clusters $\{C_1, C_2, \ldots, C_k\}$ we can compute an upper bound on the projection of reachable states in the product space to a component C_i. Assignments to the latches in component C_i not corresponding to the above states can never be attained in any environment [13].

For components that are too small, this approach fails to give any improvement since typically all states are reached in the component; for components that are too large, the fixed point computation fails. We provide a routine which computes A^i_∞ for a given C^i. We found that for medium sized clusters (approx. 15 latches), A^i_∞ would be considerably smaller than the full state space of the cluster. However, somewhat to our surprise, using the corresponding unreached states to minimize the transition relation for the component usually led to larger BDDs; we are conducting further experiments.

4.5 Removing Redundant Latches

The basic motivation behind this approach is to simplify BDD's for transition relations and reached state sets by removing variables.

A latch is *redundant* if it can be replaced by a wire without changing the functionality of the circuit. Replacing a latch by a wire, reduces the number of BDD variables by two, and heuristically speacking would also reduce the size of the BDDs containing these variables.

We briefly describe two methods to find redundant latches and remove them.

Constant Propagation: Sometimes latch inputs are tied to either *VDD* or *GND*. In our algorithm we detect such latches and propagate their constant values to their fanouts (hence the term "constant propagation"). An example is shown in Figure 3.

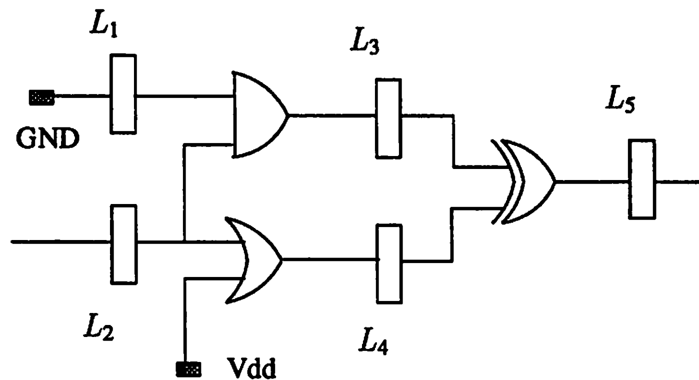

Figure 3: Propagation of Constants through Latches: L_1, L_3, L_4 and L_5 are redundant.

A recursive algorithm for removing redundant latches is the following:

To each latch attach a value parameter - "constant" or "variable" and a status parameter - "processed" or "unprocessed".

1. Mark all latches as "unprocessed".
2. While there exists unprocessed latches, pick an "unprocessed" latch L.
3. Call the function *find_redundant*(L).
4. return

find_redundant(L):

1. If L is processed, then return its value ("constant" or "variable").

2. Mark L as "processed"
3. If L is tied to *VDD* or *GND* assign value "constant" and return.
4. Assign value "variable" to L.
5. For each variable v in the support of the next state function of the latch,
   (a) If v corresponds to a primary input, mark the latch as "variable" and return.
   (b) If v represents a wire with constant value "0" or "1", continue.
   (c) Find latch L_i for which v is the present state variable, and call *find_redundant*(L_i).
6. Modify the next state function of the latch by propagating the constant value of fan-ins.
7. If the next state function becomes a constant then change the value of L to "constant".
8. return value.

At the end of the algorithm, the next state functions of latches do not contain present state variables corresponding to redundant latches. These variables are not considered for further BDD manipulations.

Latch Removal by Retiming: Retiming rearranges the storage elements in a circuit to reduce its cycle time or to reduce the number of storage elements, without changing its functionality [28]. We use retiming to reduce the number of storage elements. A simple example to demonstrate this is shown in Figure 4(a). Note that, the

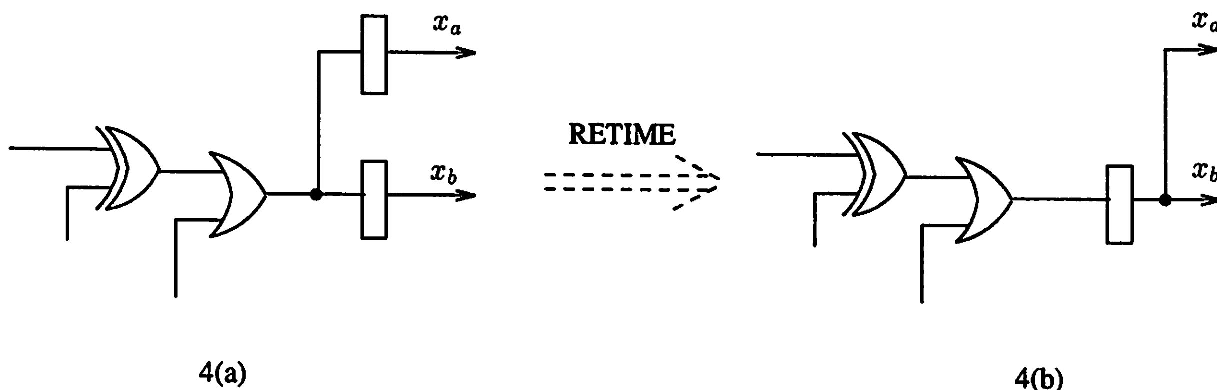

Figure 4: Removing latch by re-timing.

inputs of these 2 latches are fed by the same combinational logic. Hence the next state values of these latches will always be same. The new circuit with the redundant latch removed is presented in Figure 4(b).

An algorithm for removing redundant latch by retiming is following:

1. Sort all the latches in increasing order of the support size of the corresponding next state functions.
2. For each pair of latches L_i and L_j, with the equal support size do the following:
   (a) Suppose x_i, x_j denote the corresponding present state variables (outputs of latches) and F_i, F_j are the corresponding next state functions.
   (b) Find the co-factors $F_{i_{x_i}}$, $F_{i_{\bar{x}_i}}$, $F_{j_{x_j}}$, and $F_{j_{\bar{x}_j}}$.
   (c) If $F_{i_{x_i}} \equiv F_{j_{x_j}}$ and $F_{i_{\bar{x}_i}} \equiv F_{j_{\bar{x}_j}}$ then remove L_j from the circuit and replace it by a "wire" instead (as illustrated in Figure 4).

4.5.1 Results and Discussion

The results of redundant latch removal techniques on various examples are shown in Table 6. We observe up to

examples	A	B	C	D	E	F		G	
						$\|T\|$	$\|R\|$	$\|T\|$	$\|R\|$
gigamax	45	9	0	0	9	2018	402	1389	301
BDLC*	144	1	0	5	6	24275	12208	23441	9984
BIU	154	6	2	26	34	30834	25276	20088	20956

A: Total # of latches
B: # of constant latches removed without constant propagation
C: # of latches removed after constant propagation
D: # of latches removed by re-timing
E: Total # of latches removed
F: Redundant latches not removed
G: Redundant latches removed
$|T|$, $|R|$: As in Table 2.

Table 6: Effects of redundant latch removal on BDD sizes.

30% reduction in the BDD size of the transition relation. Also a reduction of up to 25% was obtained for the BDD size of the reached set.

In the above analysis the reset values of the latches were ignored, i.e. we did not check for the consistency of the reset values. However these optimization techniques can be applied even if the reset values of the latches are taken into account. In the constant propagation approach, the reset value of the latch must match the constant next state value it takes, for it to be made redundant. In the retiming approach, the reset values of the latches must be identical for either of them to be removed. This analysis can be done very easily.

A similar approach was proposed by Lin [27] who describes an algorithm to remove a maximal set of state variables without affecting the uniqueness of reachable states. The problem with this approach is that we need to pre-compute the set of reachable states. In many big designs computing the reachable states becomes infeasible due to the size of the BDD; in our technique redundant latches are removed once the next state relations are calculated. Hence the size of the transition relation and the reached state set is reduced before we need to compute it.

However, our approach is orthogonal to Lin's. After minimizing the transition relation using this approach, we can still apply Lin's method to possibly remove more latches and get a further reduction in BDD size after computing the set of reachable states.

Beer *et al.* [5] mentioned a "constant-elimination" technique to reduce the number of inputs and memory elements; however no algorithm is proposed to detect such cases.

5 Conclusion and Future Work

We described a data structure and a series of algorithms for efficient formal design verification using BDD's. The data structure represents a non-deterministic system as a deterministic netlist of gates and latches which allows for efficient manipulation of hardware designs. We argued that the core computation in BDD based formal 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, use of

clustering, use of don't cares and removal of redundant latches. The efficacy of these algorithms was demonstrated on a set of seven industrial examples ranging in size from 28 to 172 binary latches.

This is part of a second generation BDD based tool (HSIS) 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 formal design verification as well as developers interested in creating their own applications on top of the efficient core computation routines provided.

Other BDD based techniques which look promising include the "exists–cofactor" of [10], and the "implicitly conjoined invariants" of [24]. We plan to experiment with them since it should be relatively easy with the data structure proposed in this paper to implement these methods. 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 [25] 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 [21], and conservative approximations to reached state sets [13]. We believe these techniques can be conveniently developed in our framework and then tested and compared on realistic examples.

6 Acknowledgements

We gratefully acknowledge the supported provided by grants from Motorola and SRC.

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, V. Singhal, and R. K. Brayton. Verifying Interacting Finite State Machines: Complexity Issues. Technical Report UCB/ERL M93/68, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Sept. 1993.

[4] 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.

[5] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–193. Springer-Verlag, 1994.

[6] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing Kripke Structures in Temporal Logic. Technical Report CMU-CS-87-104, Department of Computer Science, Carnegie Mellon University, 1987.

[7] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, C-35:677–691, Aug. 1986.

[8] 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.

[9] 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.

[10] 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.

[11] B. Chen, M. Yamazaki, and M. Fujita. Bug Identification of a Real Chip Design by Symbolic Model Checking. In *Proc. European Conf. on Design Automation*, Paris, France, Feb. 1994.

[12] S. T. Cheng. Compiling Verilog into Automata. Technical Report UCB/ERL M94/37, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1994.

[13] 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.

[14] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, 1986.

[15] 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.

[16] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *Proc. Intl. Conf. on Computer Design*, pages 522–525, Oct. 1992.

[17] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science, 1990.

[18] E. A. Emerson and C. L. Lei. Modalities for Model Checking: Branching Time Strikes Back. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 84–96, 1985.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Co., 1979.

[20] 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.

[21] 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.

[22] A. Gupta. Formal Hardware Verification Methods: A Survey. In *Formal Methods in System Design*, pages 151–238. Kluwer Academic Publishers, New York, 1992.

[23] Z. Har'El and R. P. Kurshan. Software for Analytical Development of Communication Protocols. *AT&T Technical Journal*, pages 45–59, Jan. 1990.

[24] 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.

[25] 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.

[26] 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.

[27] B. Lin. *Synthesis of VLSI Design with Symbolic Techniques*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, 1991.

[28] B. Lockyear and C. Ebeling. Optimal Retiming of Level-Clocked Circuits Using Symmetric Clock Schedules. In *IEEE Trans. Comput.-Aided Design Integrated Circuits*, pages 1097–1109, Sept. 1994.

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

[30] 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.

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

[32] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proc. Intl. Conf. on Computer Design*, pages 328–333, Oct. 1992.

[33] 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.