Submitted to *Design Automation Conference*, Anaheim CA, June 9 – 13, 1997

Heuristic Symmetry Reduction for Invariant Verification

William N. N. Hung[1] Adnan Aziz[1] Kenneth McMillan[2]

1. ECE Department, The University of Texas At Austin

2. Cadence Berkeley Laboratories, Berkeley

Abstract

We describe techniques that use symmetry to perform efficient invariant checking. We start by developing the theory needed to exploit symmetry for designs specified at the gate level. This is followed by a proof of the inadequacy of BDD based methods for highly symmetric designs; this motivates the use of explicit state enumeration. Exact symmetry reduction is conjectured to be computationally intractable; we propose fast heuristic reduction procedures. Experiments with these routines demonstrate their effectiveness in practise; we also compare running times with a BDD based tool.

Keywords: Formal Verification, Invariant Checking, Symmetry, Netlists, Hierarchy, Lower bounds for BDDs

1 Introduction

In this paper we will be concerned with the problem of *design verification*. Traditionally, designs have been verified by extensive simulation: a model is built (in software or hardware) and large numbers of test inputs are applied. The behavior of the model is compared with that of a reference model, or simply eye-balled. This methodology offers no guarantees of correctness (indeed the the precise definition of what constitutes a correct design is never specified); for large designs, the fraction of the design which can be explored in this fashion becomes exponentially small. There are many examples of designs that passed extensive simulation, but were still found to contain bugs [18, 12, 5]. Indeed verification is a major bottleneck in product development; in many cases the group performing verification is comparable in size to the design team itself.

This sorry state of affairs has led to the proposal of "formal methods" for design verification; the adjective formal refers to the unambiguous specification of the system and the properties being checked, together with the validation step generating a mathematically rigorous proof of correctness.

A formal methodology for verification consists of the following [2]:

1. A set of rules to generate formulas describing *systems*, and semantic functions that assign meaning to the formulas.

2. A set of rules to generate formulas defining the *properties* that are to be checked, and semantic functions that define satisfaction of a property by a system.

3. An *algorithm* for checking the satisfaction relation.

Systems can be described at increasingly detailed levels, incorporating timing functionality, complex data types, arithmetic, etc. Similarly, the class of properties can be made increasingly expressive, starting from Symbolic Trajectory Evaluation, passing through CTL, CTL*, and beyond. Generally, the higher the level of system/property detail, the higher the computational complexity of checking the satisfaction relation [11].

Netlists, which are built out of an interconnection of gates and latches, naturally arise in the course of digital design. These are described in detail in Section 2, and will play the role of systems in this paper. Properties will be subsets of the state space of the design, and will be referred to as invariants. referred to as invariants. A netlist, together with a designated initial state, satisfy the property when all reachable states lie in the invariant, i.e. there is no sequence of inputs which when applied to the design starting at the initial state leads to a state lying outside the invariant. While not being nearly as expressive as formalisms such as CTL, invariants can still be used to specify a very large class of important safety properties, such as mutual exclusion [9]; they can readily be extended to handle deadlock.

A naive approach to checking invariants would be to extract the state transition graph of the design, and check whether all states reachable from the initial state belong to the invariant. However a design with n 1-bit registers has 2^n states; consequently this approach takes prohibitively large computational resources, and is not practical for all but the smallest of designs. An obvious improvement to the naive algorithm is to explore states incrementally, and only store the reachable states; while usually being a tiny fraction of the entire state space, the set of reachable states is still enormous. Part of this complexity is inherent: verifying invariants on any class of compositional designs is typically P-SPACE–hard.

In practise, many designs are well structured, and this can be exploited to devise heuristic procedures which perform well on specific classes of designs.

For example, Binary Decision Diagrams (BDDs) use an implicit representation of the state space of the model where complexity is a function of the "mutual information" between parts of the system being modeled, rather that the number of states of the model [19]. Partial order methods exploit the fact that many events in concurrent systems are independent; that is, the temporal order of two events may be irrelevant to the to the outcome of a computation. By considering executions that are not totally ordered, we can reduce the space of possible executions much more efficiently [13]. In compositional approaches, modules are reduced to only those behaviors that are relevant to the property [7].

Abstraction techniques can be used to decompose the design into simpler components. They are also applied to decompose the specification into sub-properties. In contrast to the approaches described above, design verification using abstraction requires a considerable amount of user intervention and expertise [16].

A large class of designs incorporate a certain degree of symmetry – examples include memories, caches, register files, bus protocols, network protocols, and generally anything involving replicated structure [6]. For certain classes of properties, verification of the invariant at a particular state ensures its correctness at all symmetrically equivalent states. Various researchers have exploited this fact to reduce the complexity of verification [6, 8, 15]. In these approaches, a candidate symmetry relation is specified by the designer, and checked by the tool. When verification is performed, analysis is restricted to representative states drawn from the equivalence classes of symmetrically equivalent states.

Clarke et al. [6] used symmetry in conjunction with BDDs to reduce the size of the BDDs arising in CTL model checking. They built up a beautiful theory which formalized the notions of symmetric states and characterized CTL properties where symmetry reduction is sound. They also proved nontrivial lower bounds on the computational complexity of the canonicalization problem, which is integral to symmetry reduction (cf. Section 5.2). Another contribution was an exponential lower bound on the BDD specifying the equivalence relation induced on the state space for a large class of symmetries. Ip and Dill [15] used symmetry for faster invariant checking when using an explicit representation for states. They restricted their attention to a special class of symmetries, which they refered to as scalarsets – basically, a set of components possess scalarset symmetry when any pair can be interchanged. Recently, Ip has extended his work to reflexive and rotational symmetries. Dill has anecdotally remarked that for the designs that they were verifying (mostly loosely coupled cache

coherency protocols) coupling explicit data structures with symmetry reduction is superior to using BDDs; this provided the initial impetus for our work on symmetry reduction.

We have been strongly influenced by the work of Clarke et al. [6], and Ip and Dill [15]. Our first significant contribution is the proof of an exponential lower bound on the size of BDDs needed to represent the reached state sets of completely symmetric systems; this provides the theoretical justification for using explicit state enumeration. Another important contribution of this paper is the development of heuristic symmetry reduction procedures; the need for such procedures was stressed by Clarke et al [6]. These procedures are fast, and are optimal in many cases. Another contribution is the interpretation of the theory of symmetries for designs specified as netlists; Clarke et al. [6] performed their analysis for Kripke structures, while Ip and Dill [15] used a guarded command language. The extension to netlists is nontrivial; we hope that it will make work in symmetry more accessible to the majority of designers, who are more used to specifying designs as netlists. Furthermore, our check that the designer specified symmetries are sound is BDD based (cf. Section5.2); as such it is complete as opposed to the approach taken by Ip and Dill [15], which consist of a simple syntactic check on the source program. When the design in question has been synthesized, symmetry may be lost at the syntactic level, but since synthesis preserves logical functionality, a BDD based checker will not be affected adversely. It appears that Clarke et al. [6] do not actually check the candidate symmetries for correctness.

We have implemented the procedures described in this paper, and experimented with two classes of examples incorporating various degrees of symmetry. These include a hierarchical tree based mutual exclusion protocol, and randomized routing protocols for a variety of topologies (cube,star,hybrid,etc.), based loosely on those described in [21]. In addition to reporting results on our reduction algorithms on these examples, we have provided what we believe is the first concrete comparison of explicit and BDD based verification.

The remainder of this paper is structured as follows: in Section 2, we give the syntax and semantics for netlists. This is followed in Section 3 by a detailed exposition of the invariant checking problem. In Section 4 we prove an exponential lower bound on the size of the BDD for the characteristic function of the set of permutations. Symmetries and their application to invariant checking on netlists are described in Section 5. The heuristic symmetry reduction algorithms are developed in Section 6. Experiments, together with a discussion of their significance are given in Section 7. We conclude by summarizing our results, and suggesting future work in Section 8.

2 Formal models for hardware

In order to be able to formally reason about hardware, we need to develop mathematical models for digital systems. In this section we develop two formalisms for expressing designs, namely finite state machines and netlists. Finite state machines are more abstract; they correspond to the behavioral specification as given by the designer. Netlists are "structural"; they are closer to the actual implementation.

We first review some useful terminology in the theory of sequences. Recall that in the usual set-theoretic development of the natural numbers $0, 1, 2, \ldots$, each natural number is the union of its predecessors [26]. Thus 0 is $\{\}$, 1 is $\{0\}$, 2 is $\{0, 1\}$, and so on. The set of natural numbers is denoted by ω. A *finite sequence over* a set Σ is a function whose domain is a natural number, and whose range is a subset of Σ. We will denote the sequence $f : n \to \Sigma$ by $\langle f(0), f(1), \ldots, f(n-1)\rangle$.

2.1 Finite state machines

Finite state machines provide a natural way of describing systems in which the output depends not only on the current input, but also on past values of the input, while possessing only a bounded amount of memory. Hopcroft and Ullman described the theoretical aspects of FSMs in [14]; below we develop enough theory to suffice for this paper.

Definition 1 A Finite State Machine (FSM) is a quintuple $(Q, I, O, \lambda, \delta)$ where Q is a finite set referred to as the *states*, I, and O are finite sets referred to as the set of *inputs* and *outputs* respectively, $\delta : Q \times I \to Q$ is the *next-state function*, and $\lambda : Q \times I \to O$ is the *output function.*

An FSM on inputs X and outputs Y can be represented graphically by a directed finite graph, referred to as a *state transition graph*, where the vertices are referred to as *states*, and the edges are referred to as *transitions*. The edges are labeled with input/output value pairs – the input value enables the transition, and the output value is produced. The destination node of the edge represents the next state for that input value. This is illustrated in Figure 1.

Given a state s_0 and a sequence of inputs $\mathbf{i} = \langle i_0, i_1 \ldots, i_{k-1}\rangle$ we will refer to the sequence of states $\sigma = \langle s_0, s_1, \ldots, s_k\rangle$ as being the *run* (sometimes referred to as the *path*) starting at s_0 on input $\mathbf{i}$ iff for all k, $\delta_M(s_k, i_{k+1}) = s_{k+1}$.

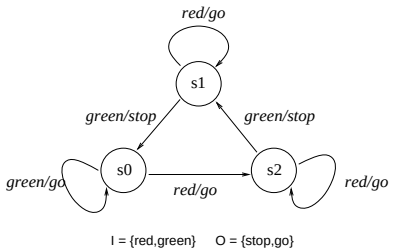


Figure 1: Graphically representing FSMs

2.2 Netlists

A netlist is a representation of a design at the *structural level*. It is closer to the actual implementation of the design than FSMs, which can be viewed as behavioral level descriptions of the design [20].

Definition 2 A *netlist* is a directed graph, where the nodes correspond to elementary circuit elements, and the edges correspond to wires connecting these elements. Each node is labeled with a distinct variable w_i. For simplicity, we will assume that the netlist is *Boolean*, i.e. all variables take values in $2 = \{0, 1\}$; the extension to the multi-valued case is straightforward. The three basic circuit elements are *primary inputs, latches*, and *gates*. Primary input nodes have no fan-ins; latches have a single input. Every gate G has an associated Boolean function $f_G : \{0,1\}^n \to \{0,1\}$, and an ordered list of n nodes referred to as its *inputs*. (When $n = 0$ the gate is a constant). Some nodes are designated as being *primary outputs*.

We require that every cycle in a netlist to include at least one latch (i.e. there are no combinational cycles).

Figure 2 provides a graphical depiction of a netlist.

The definition given above of a simple netlist is essentially the data structure used by many logic synthesis tools to internally represent designs derived from HDL descriptions of gate/RTL level designs.

Given a set of assignments to each input node and a state, one can uniquely compute the values of each node in the netlist by evaluating the functions at gates. In this way, a netlist η on inputs $a_1, a_2, \ldots, a_n$, outputs $b_1, b_2, \ldots, b_m$ and latches $r_1, r_2, \ldots, r_k$ bears a natural correspondence to an FSM M_η on inputs $X_\eta = 2^n$, outputs $Y_\eta = 2^m$, and state space $Q_\eta = 2^k$. The next-state function of M_η is defined by the composed logic gates in the following manner: for each latch r_i we can find a function $f_i : Q_\eta \times I_\eta \to \{0,1\}$ by composing the functions of the gates from the inputs and latch outputs to the input of the latch. We will refer to f_i as the next-state function of the latch i.

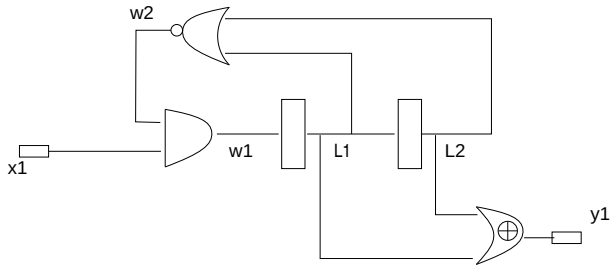


Figure 2: A simple netlist: the node x_1 is a primary input; l_1, l_2 are latches, and w_1, w_2, y_1 are gates. y_1 is designated a primary output.

Then $\delta_{M_\eta}(w_1, w_2, \ldots, w_n, r_1, r_2, \ldots, r_k) : Q_\eta \times X_\eta \to Q_\eta$ is simply the vector of functions $[f_1 f_2 \ldots f_k]$. Similarly, the output function is defined by composing the functions of gates from inputs and latches to output nodes.

An example of this correspondence is given in Figure 3. We will refer to the netlist as being an *implementation* of the finite state machine.

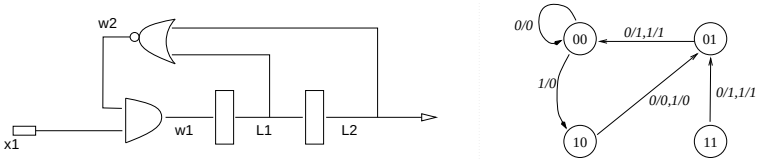


Figure 3: Deriving an FSM from a netlist

3 Design Verification through Invariant Checking

Given a netlist η, and a pair of states (s, t) from S_η, we will say t is *reachable* from s if there is a sequence of inputs $\langle i_o, i_1, \ldots, i_{k-1} \rangle$ and a sequence of states $\langle s_0, s_1, \ldots, s_k \rangle$ so that $s_0 = s$, $s_k = t$, and $(\forall j)\,[\delta_\eta(s_j, i_j) = s_{j+1}]$. A common problem in formal verification of hardware designs is to determine if every state reachable from the reset state lies in a set G of "good states"; this is sometimes referred to as *invariant checking*. Typically, G is presented as the set of states for which a particular output node takes a specified value.

One solution to the invariant checking problem is to determine all states reachable from the reset state, and check that they all lie in G. This can be achieved by performing a traversal of the stage transition graph of the design. The traversal can be depth first or breadth first — successors of a

state can be generated by iterating over all possible inputs, This approach is summarized in Figure 4 – the traversal is depth first. The procedure can readily be modified so that when the invariant fails, it returns a state lying outside the invariant, together with a sequence of inputs that will take the initial state to the bad state.

```
 /* Initially s is the reset state, visited is set to ∅ */
boolean function check_invariant(s, η, G, visited) {
 if (is_member( s, visited) )
    return TRUE;
 if (s ∉ G)
    return FALSE;
 for_each_input (η, i) {
    t = F_η(s, i);   /* call cycle simulator */
    if (check_invariant(t, G, visited))
       add_to_set( visited, t);
    else
       return FALSE;
 }
 return TRUE;
}
```

Figure 4: Basic algorithm for invariant checking.

In a straightforward implementation, the set of visited states would be stored explicitly, e.g in a linked list or a hash table, with the entries being bit arrays. However, the storage requirements can become enormous, since the number of reachable states in a design with n registers can be as large as 2^n. This "explosion of states" is the major bottleneck in formally verifying large sequential designs. To some extent, this complexity is inherent, since the invariant checking problem is P-SPACE–complete. However, a variety of heuristics have been proposed to deal with this complexity. One such heuristic is based on the idea that many designs contain repeated symmetric components, and as a result, much of the search of the state space can be "pruned".

4 BDDs perform poorly on permutations

For designs containing a large amount of symmetry, the set of reached states often is very similar to the set of all permutations of the initial state; this happens for example in randomized routing schemes, where any packet can reach any location on the way to its destination [21].

In this section we show that there is no variable ordering under which a polynomial sized Reduced Ordered Binary Decision Diagram can be built for the characteristic function of the set of permutations. This immediately shows that BDDs are not a good data structure to represent sets of states in which the reached state set can reach permutations of the initial set of states; such designs occur for example in routing protocols. We do this by lower bounding the number of cofactors of the BDD for the function and by looking at the least significant bits of the words.

The characteristic function for permutations

The characteristic function of the set of permutations on $\{0, 1, \ldots, N-1\}$ is the Boolean function $f_N : 2^{N \cdot \log N} \rightarrow 2$ defined as follows: (here $n = \log N$ for convenience)

$$f_N(\alpha_{00}, \alpha_{01}, \ldots, \alpha_{0(n-1)}, \alpha_{10}, \alpha_{11}, \ldots, \alpha_{1(n-1)}, \ldots, \alpha_{(N-1)1}, \alpha_{(N-1)2}, \ldots, \alpha_{(N-1)(n-1)}) = 1$$

if and only if $(\mathbf{a_0}, \mathbf{a_1}, \ldots, \mathbf{a_{N-1}})$ (where a_i is the integer derived by viewing $\alpha_{i0}\alpha_{i1}\cdots\alpha_{i(n-1)})$) as the binary representation of an integer) is a permutation of $\{0, 1, \ldots, N-1\}$, that is for each $p \in \{0, 1, \ldots, N-1\}$, there is an k so that $p = \mathbf{a_k}$.

Lower bounding the BDD size

Theorem 4.1 *Under any variable ordering, the BDD for f_N has at least $2^{N/2}$ nodes.*

Proof: Let f_N be defined over the variables

$$X = \{x_0^0, \ldots, x_{n-1}^0, x_0^1, \ldots, x_{n-1}^1, \ldots, x_0^{N-1}, \ldots, x_{n-1}^{N-1}\}$$

Define the set of variables T to be the variables corresponding to the least significant bits, i.e.

$$T = \{x_0^0, x_0^1, \ldots, x_0^{N-1}\}$$

Let σ be an arbitrary variable ordering for f_N. Now walk down the variable ordering σ starting from the top, until $N/2$ variables from T have been encountered. Denote the set of variables in the ordering up to this point by V. We will show f_N has at least $2^{N/2}$ cofactors with respect to the set of variables in V, proving the theorem.

Denote the set of variables from T in V by I. Now let $q_0, q_1, \ldots q_{N-1}$ be the sequence of indices of the low order bits as they occur in the variable order. That is, let the projection of the variable order on to the low order bits be $x_0^{q_0}, x_0^{q_1}, \ldots, x_0^{q_{N-1}}$.

Assign the value $q_i \bmod (N/2)$ to each set of high order bits $x_1^i, x_2^i, \ldots, x_N^i$. Call this partial truth assignment ϕ; it leaves open the value of the lower order bits in such a way that if we fix the values of the variables in I, then it exactly determines the values of the variables in $T - I$ (which must be exactly the complement of the corresponding variables in I).

Now choose any distinct truth assignments σ and σ' to I. Clearly the truth assignments $\sigma \cup \phi$ and $\sigma' \cup \phi$ when projected onto V must give distinct cofactors of f_N. This is because if we take the unique extension of $\sigma \cup \phi$ satisfying f_N and project onto V, we get a truth assignment which is not consistent with $\sigma' \cup \phi$ projected to V. Hence there are at least $2^{N/2}$ cofactors of f_N, proving the claim of the theorem. ■

We contrast our result with that of Clarke et al. [6] who showed that for a fairly general class of symmetries (including fully permutational symmetries), the BDD representing the equivalence relation on the states resulting from these symmetries was exponential. (This equivalence relation is described in detail in the next section.) This in itself does not show that the BDD for the set of permutations is exponential, since instantiating variables decreases the size of the BDD. The converse is true, that is our result implies that the BDD for the equivalence relation on the states which are permutations of one another must be exponential (since the BDD for the set of permutations results on instantiating one set of variables from the equivalence relation to a permutation).

It should also be pointed out that even when there is little node sharing, BDDs are still asymptotically as efficient a way of representing sets as any explicit data structure. However, in real life, constants count; consider a design with 128 1-bit storage elements. Representing a state as a bit vector takes 16 bytes [22]; the BDD for the same state contains 128 bdd nodes, each of which takes approximately 20 bytes of storage, resulting in a total of 2560 bytes.

5 Exploiting Symmetry

5.1 Symmetries

We first develop the bare minimum amount of results/terminology related to symmetries; Armstrong [3] is a good general reference to symmetries.

Let J_p be the set of integers $\{1, 2, 3, \ldots, p\}$. A bijective map $\sigma : J_p \to J_p$ is called a *permutation* of the integers from 1 to n. The set of all bijective map on J_p wil be denoted by S_p. Since all bijective maps have inverses, we can safely refer to the inverse of $\sigma \in S_p$ by σ^{-1}. Given a permutation $\sigma \in S_n$ we will use $\mathtt{Supp}(\sigma)$ to denote the set $\{k : \sigma(k) \neq k\}$. The *identity permutation* defined by $f(k) = k$ is typically denoted by e.

For σ_1 and σ_2 from S_p, define the function $\sigma_1 \circ \sigma_2$ from J_p to J_p to be the composite mapping $i \mapsto \sigma_1(\sigma_2(i))$. It is readily checked that $\sigma_1 \circ \sigma_2$ is also bijective, and hence a member of S_p.

A subset Π of S_p is said to be *closed under inversion* if whenever $\sigma \in \Pi$ then $\sigma^{-1} \in \Pi$. The subset Π is *closed under composition* if whenever $\sigma_1, \sigma_2 \in \Pi$ then $\sigma_1 \circ \sigma_2 \in \Pi$. If Π is closed under both inversion and composition, it is referred to as a *subgroup* of S_p.

Given a subset $T = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ of S_p, define the subgroup *generated* by T to be the smallest subgroup of S_p containing T. The existence and uniqueness of such a subgroup follows from the observations that S_p contains T and is closed under inversion and composition, and that given any subgroups T_1, T_2 of S_p the set $T_1 \cap T_2$ is also a subgroup. We will denote the closure of T by $[\![T]\!]$ the elements of T will be referred to as the *generators* of $[\![T]\!]$.

For any $\sigma \in S_p$, consider the permutations $\{\sigma, \sigma^2, \sigma^3, \ldots\}$. Since S_p is finite, for some m, n with $m < n$ we must have $\sigma^m = \sigma^n$; then $\sigma^{n-m-1} \circ \sigma = e$, so $\sigma^{n-m-1} = \sigma^{-1}$. Thus every element of $[\![T]\!]$ can be written as a product of elements of T.

Given an n-dimensional vector $\vec{v} = (v_1, v_2, \ldots, v_p)$, and a permutation $\sigma \in S_p$ we will write $\sigma(\vec{v})$ to denote the vector $(v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(p)})$; the vector $\sigma(\vec{v})$ will sometimes be referred to as the *action* of σ on $\vec{v}$. Given a set of vectors $V = \{\vec{v_1}, \vec{v_2}, \ldots, \vec{v_l}\}$, and a subset $T = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ of S_p, the set $T \cdot V$ is defined to be the set of all vectors $\vec{v}$ for which there are $\sigma_i \in T$ and $\vec{v'} \in V$ so that $\vec{v} = \sigma_i(\vec{v'})$; we will stretch our definition of action, and refer to $T \cdot V$ as the *action* of T on V.

A subgroup Π of S_p gives rise to a natural equivalence relation $\mathcal{E}_\Pi$ on bit-strings of length p, i.e. on elements of $\{0,1\}^p$. The equivalence is defined as follows — $(\alpha, \beta) \in \mathcal{E}_\Pi$ precisely when there exists $\sigma \in S_p$ such that $\beta = \sigma(\alpha)$. It is routine to verify that $\mathcal{E}_\Pi$ is an equivalence relation.

The equivalence classes of $\mathcal{E}_\Pi$ are referred to as its *orbits*. The set $\{0,1\}^p$ can be totally ordered

by the *lexicographic* order relation; $\alpha \prec_{\texttt{lex}} \beta$ precisely when $\Sigma_{i=1}^{p}\alpha_i \cdot 2^i < \Sigma_{i=1}^{p}\beta_i \cdot 2^i$. In this way, given an orbit, we can refer to its *canonical representative*, which is simply the largest element of the orbit under the relation $\prec_{\texttt{lex}}$. Since $\mathcal{E}_\Pi$ is an equivalence relation, the orbits constitute a partition of $\{0,1\}^p$; given an element α of $\{0,1\}^p$, we will use $\hat{\alpha}_\Pi$ to denote the canonical representative of the orbit containing α; when the subgroup Π in question is clear, we will simply write $\hat{\alpha}$.

Given a subset A of $\{0,1\}^p$ and a subgroup Π of S_p, we will say A is *invariant* under Π when $\Pi \cdot A = A$; note that since Π is a subgroup, it must contain the identity, and so $\Pi \cdot A$ trivially contains A.

The following lemma is easily proved:

Lemma 5.1 *When Π is generated from the set $\Pi = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$, A is invariant under $[\![\Pi]\!]$, iff A is invariant under $\{\sigma_i\}$ for every $\sigma_i \in [\![\Pi]\!]$.*

The power of this lemma is that we can check if a given set of states is invariant under $[\![\Pi]\!]$ simply by computing the action of each $\sigma \in \Pi$ on A; the latter can be done using BDDs by reordering the present state variables.

5.2 State space reduction

We are now ready to describe the use of symmetries for reducing the complexity of checking invariants.

Let η be a netlist with n latches and k inputs; let $x_1, x_2, \ldots, x_n$ and $u_1, u_2, \ldots, u_k$ be the variables corresponding to the latches and inputs respectively. Let the next state functions of the latches be $f_1, f_2, \ldots, f_n$. We will find it convenient to use $\vec{F}_\eta$, $\vec{x}$ and $\vec{u}$ to denote the vectors $(f_1, f_2, \ldots, f_n)$, $(x_1, x_2, \ldots, x_n)$ and $(u_1, u_2, \ldots, u_k)$. In this way, given a state $\vec{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in 2^n$ and an input $\vec{\iota} = (\iota_1, \iota_2, \ldots, \iota_k) \in 2^k$, the next state of the design will be $\vec{F}_\eta(\vec{\alpha}, \vec{\iota})$.

Let $G \subset S_\eta = 2^n$ be an invariant to be checked. Let Q be a subset of $S_n \times S_k$, i.e. a set of ordered pairs where the first component is a permutation from S_n and the second from S_k. Let T be the projection of Q to the first component, i.e. the set of all permutations σ in S_n so that there exists some τ in S_k such that $(\sigma, \tau) \in Q$; similarly, let R be the projection of Q to the second component.

Suppose G is invariant under $[\![T]\!]$. Furthermore, suppose every (σ, τ) in Q satisfies the following:

$$\sigma(\vec{F}(\vec{x}, \vec{u})) \quad = \quad \vec{F}(\sigma(\vec{x}), \tau(\vec{u})) \tag{1}$$

Then the following lemma holds:

Lemma 5.2 Let s be a state and u an input; take $t = \vec{F}_\eta(s,u)$. Then for every s' in $[\![T]\!] \cdot \{s\}$ there is an input u' so that $t' = \vec{F}_\eta(s',u')$ is in $[\![T]\!] \cdot \{t\}$.

Proof: Take any $s' \in [\![T]\!] \cdot \{s\}$; then by definition, there is a permutation σ_a in $[\![T]\!]$ so that $s' = \sigma_a s$. Let $\sigma_a = \sigma_{a_1} \circ \sigma_{a_2} \circ \cdots \circ \sigma_{a_j}$, where each $\sigma_{a_i} \in T$. Define the permutation $\sigma_{b_i} \in S_k$ to be such that $(\sigma_{a_i}, \sigma_{b_i}) \in Q$. Take the permutation $\sigma_b = \sigma_{b_1} \circ \sigma_{b_2} \circ \cdots \circ \sigma_{b_j}$ and let $u' = \sigma_b(u)$.

Observe:

$$\begin{aligned}
F(s',u') &= F(\sigma_a(s), \sigma_b(u)) \\
&= F(\sigma_{a_1} \circ \sigma_{a_2} \circ \cdots \circ \sigma_{a_j}(s), \sigma_{b_1} \circ \sigma_{b_2} \circ \cdots \circ \sigma_{b_j}(u)) \\
&= F(\sigma_{a_1}(\sigma_{a_2} \circ \cdots \circ \sigma_{a_j}(s)), \sigma_{b_1}(\sigma_{b_2} \circ \cdots \circ \sigma_{b_j}(u))) \\
&= \sigma_{a_1}(F((\sigma_{a_2} \circ \cdots \circ \sigma_{a_j}(s)), (\sigma_{b_2} \circ \cdots \circ \sigma_{b_j}(u)))) \quad \text{(by Equation 1)} \\
&\vdots \quad \text{repeatedly applying Equation 1} \\
&= \sigma_a(t)
\end{aligned}$$

But $\sigma_a(t)$ lies in the orbit of t by $[\![T]\!]$; thus the input u' gives the desired result. ■

Since G is invariant under $[\![T]\!]$ it has the property that if it contains a state s, it contains every state in the orbit of s under $[\![T]\!]$. Coupling this fact with Lemma 5.2, we can immediately infer the following:

Corollary 5.3 Suppose G is invariant under $[\![T]\!]$; then a state s can reach a state outside G if and only if its canonical representative $\hat{s}$ can reach a state outside G.

5.3 Symmetry Reduction

The result of Corollary 5.3 suggests the following strategy for reducing the complexity of invariant checking.

1. Have the designer suggest permutations for $Q \subset S_n \times S_k$.
2. Check that the permutations in Π satisfy the condition of Equation 1.
3. Traverse the STG of the design, while "canonicalizing" states i.e. mapping states to the canonical representatives elements of their orbits.

The advantage of this approach is that it is not necessary to store all states — only canonical representatives are stored. Additionally, many states may be avoided all together, since they may not be reached by passing directly through canonical representatives. An example of this approach is shown in Figure 5.

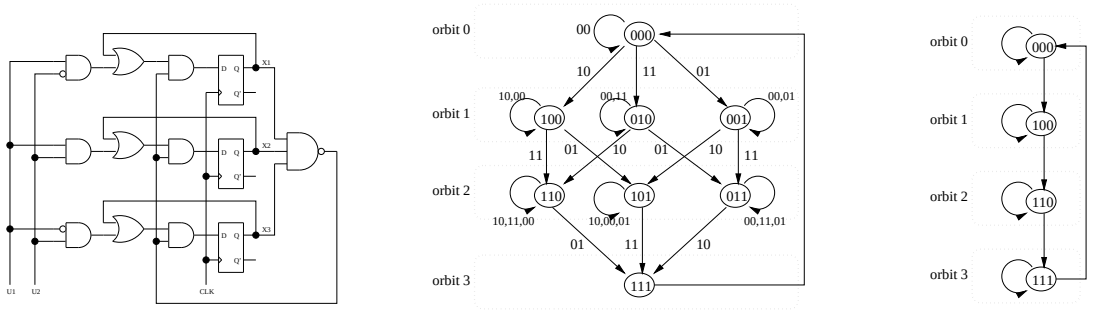


Figure 5: A symmetric netlist, and its STG; the orbits are the circled sets of states. The STG on the canonical representatives is on the right.

Pseudo-code for an algorithm for invariant checking with symmetry reduction is given in Figure 6, where the function $R_\Pi : \{0,1\}^n \to \{0,1\}^n$ maps states to their canonical representatives.

5.3.1 Implementation issues

With a little experience, it is relatively straightforward for the designer to suggest permutations for Π. For many designs, BDDs for the next-state functions of individual latches are small, and can be built easily [25]. The check that the permutations satisfy Equation 1 is simple to implement using BDDs: it amounts to re-ordering [23], and the resultant BDDs are exactly the same size as the original BDDs (when the symmetries do satisfy Equation 1).

The problem with the strategy described above is that there is no known efficient method for taking a set of permutations T and a state s, and computing the canonical representative of s under $[\![T]\!]$. Indeed, Clarke et al. [6] have shown that it is extremely unlikely that such a method exists:

Theorem 5.4 *Given states s and t and a set of permutations T, determining if s and t lie in the same orbit is as hard as the graph isomorphism problem.*

The graph isomorphism problem has been intensely studied by researchers in the field of computational

```
 /* Initially s is the reset state, visited is set to ∅ */
boolean function check_invariant_with_symm_reduction (s, η, G, visited, R_Π) {
  if (is_member( s, visited) )
    return TRUE;
  if (s ∉ G)
    return FALSE;
  for_each_input (η, i) {
    t = F_η(s, i);   /* Computed using cycle simulation */
    t̂ = R_Π(t);
    if (check_invariant_with_symm_reduction(t̂, η, G, visited), R_Π)
      add_to_set( visited, t̂);
    else
      return FALSE;
  }
  return TRUE;
}
```

Figure 6: Invariant checking with symmetry reduction.

complexity theory [10]. It is conjectured that there is no polytime algorithm for it; a consequence would be that there is no polytime procedure for canonicalization.

6 Heuristics for canonicalization

Even though Theorem 5.4 is strong evidence that the canonicalization problem is intractable in the general case, all is not lost. We will discuss in Section 6.1 classes of permutations for which canonicalization can be performed efficiently.

Additionally, it is not necessary to find the lexicographically largest equivalent state. In fact, any function $C : S_\eta \to S_\eta$ which has the property that $C(s) \in \Pi \cdot \{s\}$ can be used in place of the canonicalization function R_Π for the algorithm in Figure 6; the algorithm will continue to be correct, though it will traverse more states. We will refer to such functions as *reduction functions*.

In Section 6.2 we go about finding reduction functions which are fast to compute and come "close" to the representative; heuristically, this will result in greater reduction.

6.1 Exact Reduction Functions

There are several important sets of generators for which canonicalization can be performed in polynomial time. For example, for a set $\Pi \subset S_N$ of permutations consisting solely of *transpositions* (i.e. permutations which interchange a pair of numbers and leave the remainder unchanged) and a vector $\vec{v} \in \{0,1\}^N$, canonicalization of $\vec{v}$ can be performed by "bubble sorting" $\vec{v}$ with respect to the transpositions in Π [15].

Similarly, if Π consists of a single permutation (which includes the special case of *rotational symmetry*), $[\![\Pi]\!]$ will have at most N elements; hence $[\![\Pi]\!] \cdot \{\vec{v}\}$ can be exhaustively searched.

It is common for large designs to be *hierarchical* i.e. to consist of an set of interconnected components, possibly with some surrounding "glue logic". In such designs, there may be symmetry internal to the individual components as well as between the components themselves.

When performing verification of hierarchical designs, we flatten the hierarchy and treat it as a single netlist; the invariant is checked on the STG extracted from this netlist. Thus the notion of symmetric states and the application of canonicalization continues to hold.

Consider a hierarchical design at two levels, where the lower level has a number of replicated components. When there are symmetries on both the individual components, and at the higher level, between the components themselves, a global state can be canonicalized by first individually

canonicalizing the local state of the components, and the canonicalizing the global state keeping the relative order local states unchanged. More technically, suppose that there are n components, each with m local states bits. Then the following lemma holds:

Lemma 6.1 *Let $A = B \cup C_1 \cup C_2 \cup \cdots \cup C_n$ be a set of permutations from $S_{n \cdot m}$ so that every permutation σ in B preserves the relative ordering of the state bits local to each component, every permutation C_i acts only on the state bits local to component i, and for every i, C_i is a shifted version of C_1.*

Then the canonical representative of a vector s from $2^{n \cdot m}$ under $[\![A]\!]$ is the same as the state $\breve{s}$ derived iteratively as follows:

$$\begin{aligned} s_1 &= \widehat{s}_{[\![C_1]\!]} \\ s_2 &= \widehat{s_1}_{[\![C_2]\!]} \\ &\vdots \\ s_n &= \widehat{s_{n-1}}_{[\![C_n]\!]} \\ \breve{s} &= \widehat{s_n}_{[\![B]\!]} \end{aligned}$$

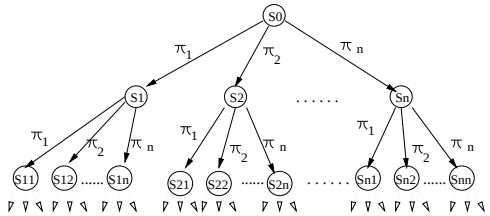
The complete proof is in the technical report; intuitively, this result follows from the observation that permutations from B do not change the relative order of the state bits local to the components, and this relative order is optimum at the first stage. Interestingly, it is necessary to canonicalize the individual components first and then the global state.

6.2 Greedy Algorithms for Reduction

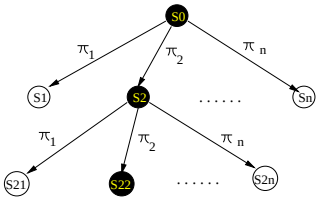
In this section we describe greedy reduction algorithms. To motivate these, we first consider the simple procedure for finding the orbit of a state shown in Figure 8. This procedure generates iteratively generates all the states derivable from the specified states by successive applications of permutations in Π. i.e. in a breadth-first manner. This is depicted graphically in Figure 7(a).

Given an orbit, one can easily determine the lexicographically maximal state in it. However, the size of the orbit can be very large. Still, the procedure in Figure 8 can easily be modified so as to greedily explore a subset of the orbit; the exploration is biased towards "large" states.

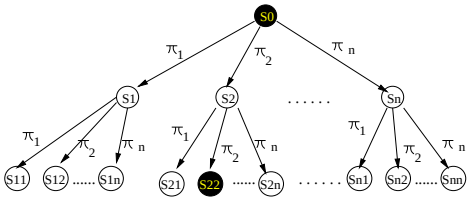
More specifically, consider the algorithm in Figure 9. Here at each step, we continue exploring only from the lexicographically largest state seen at the current iteration. This is graphically depicted



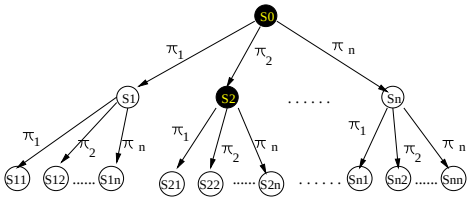
(a) Enumerating the entire orbit from s0



(b) Greedy orbit traversal - lookahead 1



(c) Greedy orbit traversal - lookahead 2



(d) Greedy orbit traversal - lookahead 2, step 1

Figure 7: Exploring orbits.

in Figure 7(b), under the caption *greedy, look-ahead-1.* Note that it is not necessary to store all the visited states, since only the lexicographically maximal state visited is desired. The termination of the algorithm follows from the fact that the **while** loop continues only as long as $s_{\texttt{max}}$, the best state seen so far, increases. Since the state space is finite, this can not go on forever. We conjecture that n^2, where n is the dimension of the state vector is an upper bound on the number of iterations.

The main problem with the algorithm in Figure 9 is that being greedy it can get trapped in local minima. One way of overcoming this is to add more "lookahead". e.g. look ahead 2, step 1. For example, we could apply pairs of permutations to the current state, and in this way determine the best state obtainable by applying two permutations from Π.

A greedy technique that has performed well in practise on a number of combinatorial optimization problems uses a lookahead of 2, but a "step" of only 1 [20]. Specifically, suppose the lexicographically maximal state reachable from the state s in two steps is achieved by applying permutations π_1 followed by π_2; then the search continues from the state $\pi_1(s)$ rather than $\pi_2(\pi_1(s))$. Pseudo-code for this algorithm is given in Figure 10. It is pictorially represented in Figure 7(d).

Both the greedy procedures outlined in Figure 9 and Figure 10 are susceptible to getting trapped in local minima. For example for the state $s = (100111)$ and set Π containing just one permutation π, where $\pi(1) = 6, \pi(2) = 1, \pi(3) = 2, \pi(4) = 3, \pi(5) = 4, \pi(6) = 5$, $\pi(s) = (001111), \pi(\pi(100111)) =$

```
  /* Generate [[Π]] · {s_0} by BFS. */
StateSet_t function BFS_generate_orbit (s_0, Π) {

  StateSet_t visited := ∅;
  StateSet_t active := {s_0};
  while ( !set_equal( active, ∅) ) {
    visited := set_union( active, visited );
    foreach_state (active, t ) {
      foreach_perm (Π, π) {
        t' := π(t);
        if ( !is_member(t', visited) )
          add_to_set( active, t' );
      }
    }
  }
  return visited;
}
```

Figure 8: Generating an orbit for a state using BFS.

```
  /* Look-ahead of 1, iterative */
State_t function greedy[1]_iter (s, Π) {
  s_max := s;
  t := s_max;
  do {
    s_max := t;
    foreach_perm (Π, π) {
      t := MAX( t, π(s_max));
    }
  } while(s_max ≺_lex t);
  return s_max;
}
```

Figure 9: Greedy algorithm with lookahead of 1

```
  /* Look-ahead of 2, step 1 */
State_t function greedy[2]_keep[1]_iter (s, Π) {
  s_max := s;
  t := s_max;
  σ_best := e;
  do {
     s_max := σ_best(s_max);
     t := s_max;
     foreach_perm (Π, π) {
        foreach_perm (Π, τ) {
           t := MAX( t, τ(π(s_max)));
           σ_best := π;
        }
     }
  } while(s_max ≺_lex t);
  return s_max;
}
```

Figure 10: Iterative, greedy algorithm with lookahead of 2, step 1.

(011110). Consequently both procedures return (100111), whereas $\pi(\pi(\pi(100111))) = (111100)$ which is optimal. In practise, this happens mostly when presented with a single permutation which represents a rotational symmetry; it can be overcome by pre-processing the set Π by adding all powers π^k for $\pi \in \Pi$.

7 Experimental Results

We experimented with our algorithms on examples which incorporate symmetry. The `4Tree` and `8Tree` designs are implementations of a tree-structured mutual exclusion protocol; they are hierarchical and afford substantial reduction. `3Cube` and `4Cube` are hypercube topologies incorporating a simple randomized routing protocol; `Dist` and `Star` are hybrid interconnection of processors, also incorporating a simple randomized routing protocol [21]. `DetGame` is from the examples of a university verification tool suite. All these examples are essentially asynchronous, and have very few inputs (corresponding to nondeterministic scheduling and routing).

Benchmark	**Description**	*Latches*
4Tree	Tree based arbiter with four cells	4-valued × 4 + 2-valued × 9
8Tree	Tree based arbiter with eight cells	4-valued × 8 + 2-valued × 21
3Cube	Routing on a 3-D hypercube	2-valued × 24
4Cube	Routing on a 4-D hypercube	4-valued × 16
Dist	Distributed network of processors	4-values × 13
Star	Star network	8-valued × 9
DetGame	Example with a university tool	9-valued × 9

Table 1: Benchmarks.

We report results on reachability analysis for these examples (this information could be used for deadlock detection, violation of mutual exclusion, etc.) in Table 2. The results indicate that explicit state enumeration coupled with symmetry reduction is usually superior to BDD based analysis for these examples — sometimes dramatically so. These experiments were conducted on a DEC Alpha with 1 GByte of main memory.

Somewhat to our surprise, on most examples the greedy reduction procedure with a lookahead of 1 performed as well as the lookahead of 2 with a step of 1 in terms of reduction; in terms of running time, it was always substantially faster. We conjecture this was because the netlists had a simple

Benchmark	*BDD implicit*		*explicit*		*lookAhead1*		*lookAhead2step1*	
	States	Time	States	Time	States	Time	States	Time
4Tree	8584	9	8584	38	1073	7	1073	9
8Tree	222570	71	222570	1705	6381	89	6381	131
3Cube	40320	144	40320	95	2400	25	1680	241
4Cube	254475	100	254475	1149	13081	210	6462	1565
Dist	63349	8	63349	230	1255	10	1255	60
Star	60480	306	60480	396	7560	97	7560	414
DetGame	181440	447	181440	35	45360	14	45360	27

Table 2: Results on Reachability Analysis.

topology; the most complex symmetries are those on the hypercube and it was on these examples that *lookahead2Step1* was able to find more reductions than *lookAhead1*. Still, increased number of search moves makes *lookAhead2Step1* inferior; thus we propose *lookAhead1* as the method of choice for symmetry reduction.

In Table 3 we provide a comparison of the running times and memory consumption for BDD based and explicit enumeration. Note that we do not pack states into bit-arrays [15], and so there is substantial scope for reducing memory usage in the explicit column. Instead of using compiled code execution model, we use fast cycle simulation techniques based on [4] and [17]. Additionally, we report the largest BDD for state sets encountered during reachability; in fact the peak BDD node consumption is significantly higher than the reported figure, since it does not reflect BDDs built during image computation.

Benchmark	*Best Explicit*		*BDDs*	
	Memory usage	Time	Memory usage	Time
4Tree	13949	7	30222	9
8Tree	95715	89	153792	71
3Cube	40320	25	183840	144
4Cube	103392	210	281568	100
dist	16315	10	24888	8
star	68040	97	175224	306
detGame	408240	14	1474512	447

Table 3: Memory (bytes)/Time (sec) Comparison with BDDs.

8 Summary

To summarize, we have made theoretical and practical contributions towards the use of symmetry in invariant verification. We developed a theory for exploiting symmetry for designs specified at the gate level. We gave theoretical justification for the use of explicit data structures instead of BDDs. We suggested heuristic procedures for symmetry reduction, and presented experimental results on a number of examples; the results are extremely promising. In addition, these experiments underlined the superiority of explicit methods coupled with symmetry to BDDs.

One problem with symmetry reduction is that it is better suited to asynchronous designs; when presented with a design with a large number of inputs, simply iterating though the possible inputs becomes infeasible. We plan to study methods for dealing with this. Computational group theory [24] has a long and rich history; we intend to study the literature of this field, and see if there are any ideas we can borrow for more efficient symmetry reduction. Many real-time protocols [1] also exhibit symmetry; we are developing a theory of symmetry for such systems. Finally, we also plan to offer a tool based on the ideas in this paper to the developers of a BDD based verification tool for formal verification; we believe this will make the capabilities of their tool more robust.

References

[1] R. Alur and D. L. Dill. Automata for Modelling Real Time Systems. In *International Colloquium on Automata, Languages and Programming*, 1990.

[2] Rajeev Alur. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Stanford University, Department of Computer Sciences, Stanford University, August 1991.

[3] M. A. Armstrong. *Groups and Symmetry.* Springer-Verlag, 1989.

[4] P. Ashar and S. Malik. Fast Functional Simulation Using Branching Programs. In *Proc. Intl. Conf. on Computer-Aided Design*, November 1995.

[5] 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*, pages 132–136, March 1994.

[6] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Logic Model Checking. In *Proc. of the Computer Aided Verification Conf.*, 1995.

[7] E. M. Clarke, K. McMillan, and D. Long. Compositional Model Checking. In *Proc. IEEE Symposium on Logic in Computer Science*, June 1989.

[8] A. Emerson and E. Sistla. Symmetry in Model Checking. In *Proc. of the Computer Aided Verification Conf.*, 1993.

[9] 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.

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

[11] A. Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in System Design*, 1:151–238, October 1993.

[12] C. Harkness and E. Wolf. Verifying the Summit Bus Converter Protocols with Symbolic Model Checking. *Formal Methods in System Design*, 4:83–99, February 1994.

[13] G. Holzmann and D. Peled. The State of SPIN. In *Proc. of the Computer Aided Verification Conf.*, August 1996.

[14] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[15] N. Ip and D. Dill. Better Verification Through Symmetry. In *Proc. Intl. Symp. on Computer Hardware Description Languages.*, 1993.

[16] David E. Long. *Model Checking, Abstraction and Compositional Verification* . PhD thesis, Carnegie Mellon University, Pittsburgh, PA, July 1993.

[17] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia. Fast Discrete Function Evaluation. In *Proc. Intl. Conf. on Computer-Aided Design*, November 1995.

[18] K. L. McMillan and J. Schwalbe. Formal Verification of the Encore Gigamax Cache Consistency Protocols. In *International Symposium on Shared Memory Multiprocessors*, April 1991.

[19] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[20] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw Hill, 1994.

[21] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[22] R. Ranjan, J. Sanghavi, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. High Performance BDD Package Based on Exploiting Memory Hierarchy. In *Proc. of the Design Automation Conf.*, Las Vegas, NV, June 1996.

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

[24] Charles C. Sims. *Computation with Finitely Presented Groups*, volume 48 of *Encyclopedia of mathematics and its applications.* Cambridge University Press, New York, 1994.

[25] 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, November 1990.

[26] R. Vaught. *Set Theory.* Birkhauser, 1995.