On the Verification of Sequential Equivalence

Jie-Hong R. Jiang, and Robert K. Brayton, *Fellow, IEEE*

***Abstract*— The state explosion problem limits formal verification to small- or medium-sized sequential circuits partly because BDD sizes heavily depend on the number of variables dealt with. In the worst case, a BDD size grows exponentially with the number of variables. Thus reducing this number can possibly increase the verification capacity. In particular, this paper shows how sequential equivalence checking can be done in the *sum state space*.**

Given two finite state machines M_1 and M_2 with numbers of state variables m_1 and m_2 respectively, conventional formal methods verify equivalence by traversing the state space of the product machine, with $m_1 + m_2$ registers. In contrast, this paper introduces a different possibility, based on partitioning the state space defined by a *multiplexed machine*, which can have merely $\max\{m_1, m_2\} + 1$ registers. This substantial reduction in state variables potentially enables the verification of larger instances. Experimental results show the approach can verify benchmarks with up to 312 registers, including all of the control outputs of microprocessor 8085.

I. Introduction

SEQUENTIAL equivalence checking plays a crucial role in VLSI design to ensure functional correctness. It has been greatly advanced since symbolic techniques [4] were used in formal methods based on state space traversal. However, these formal methods cannot be scaled as easily with the increasing complexity of system designs due to the state explosion problem, which says that the state space grows exponentially in the number of state variables. Therefore recent research [3], [11] has focused on reducing the number of state variables by retiming [13], with the hope that verification can be conducted on the reduced circuits. Unlike these circuit-based transformations, this paper reduces the register count in the verification construction. Moreover, the verification itself is *structure-independent*, that is, neither circuit similarities nor register correspondences [6] are assumed.

In this paper, we reason about sequential equivalence based on the fact that two finite state machines (FSMs) are equivalent if and only if their initial states are equivalent. To identify equivalent states of an FSM, binary decision diagrams (BDDs) [2] were used in [16], [14], [8] for symbolic execution. The fixpoint computation in [16], [14] is carried out on a product machine constructed over two identical copies of the FSM. As shown in [7], when the product machine is constructed over two FSMs under comparison, the same computation can be used for sequential equivalence checking. In addition to the approach of [16], [14], the computation in [8] for equivalent state identification is done on the original FSM without constructing a product machine. However, an n-state FSM in [8] is represented by n shared n-terminal BDDs. This representation may be expensive in practice. In contrast, we identify equivalent states by applying BDD-based functional decomposition [12] to keep the computation in the original FSM without any special representation. Since the computation is in a single FSM, we introduce the *multiplexed machine* to combine two FSMs into one. Thereby we can transform the sequential equivalence checking problem to the state equivalence problem of a multiplexed machine.

Our equivalence checking technique avoids state traversal, by partitioning the state space based on equivalence relations among states [10]. Rather than reason about the sequential equivalence in the product state space of two sequential machines under comparison, we achieve this attempt in the *sum state space*. Compared to product machine based verification, the proposed approach almost halves the number of state variables. More precisely, checking the equivalence of two n-input FSMs M_1 and M_2 with m_1 and m_2 state variables respectively, our method can keep the total number of variables to be at most $n + \max\{m_1, m_2\} + 1 + \lceil \log_2(\min\{m_1, m_2\} + 1) \rceil$. Hence, the sizes of BDDs in our verification technique could be much smaller than those in product machine based techniques.

Unlike previous verification techniques of [4] and [7], the efficiency of our approach depends heavily on the encountered number of equivalence classes of states. Since each equivalence class is represented by a BDD node, our approach is limited to instances with less than $\sim 10^6$ equivalence classes per output. Fortunately, it is applicable in most practical applications. On the other hand, because the number of equivalence classes in the reachable state subspace is invariant, our technique tends to be more robust than previous approaches in verifying different implementations of a design. For high-speed designs, registers are mostly added to reduce cycle time not to increase the number of equivalence classes. (For example, backward retiming cannot increase equivalence classes.) In such designs, our proposed technique should be preferable to those of [4] and [7].

The contributions of this paper are as follows. We apply BDD-based functional decomposition to the identification of equivalent states. Two important consequences are the elimination of universal and existential quantifications, and the possible simplification with respect to the reachable state subspace. To extend the above computation for sequential equivalence checking, we introduce the multiplexed machine such that the verification can be done in the sum state space. In addition, several techniques are proposed to enhance the computational robustness; several properties are analyzed to contrast different verification techniques.

The remainder of this paper is organized as follows. Prelimi-

Manuscript received September, 2002; revised January, 2003. This work was supported by the California State Micro Program and their industrial sponsors, Cadence and Synplicity.

J.-H. Jiang and R. Brayton are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

naries and definitions are given in Section II. After introducing the technique for equivalent state identification in Section III, we present our equivalence checking algorithm in Section IV and analyze its properties in Section V. Experimental results are then given in Section VI, and conclusions in Section VII.

II. Definitions, Notations and Preliminaries

A. Equivalence Relations and Partitions

An *equivalence relation* is a binary relation on a set, $\mathcal{S}$, satisfying reflexive, symmetric and transitive laws and induces a unique *partition* π on $\mathcal{S}$. The partition is a set $\pi = \{E_1, E_2, \ldots\}$ of subsets of $\mathcal{S}$ such that

- $E_i \neq \emptyset$ for all i;
- $E_i \cap E_j = \emptyset$ for all $i \neq j$;
- $E_1 \cup E_2 \cup \cdots = \mathcal{S}$.

Each E_i forms an *equivalence class*. Two elements in the same class satisfy the equivalence relation, but elements in different classes do not. For two equivalence relations R_1 and R_2 with partitions π_1 and π_2 respectively, $R_1 \subseteq R_2$ if and only if π_1 is a *refinement* of π_2, denoted as $\pi_1 \leq \pi_2$, i.e. each equivalence class of π_1 is contained in some equivalence class of π_2. On the other hand, the *product* of two arbitrary partitions, π_1 and π_2, denoted $\pi_1 \cdot \pi_2$, is the partition corresponding to the relation $R_1 \cap R_2$, i.e. two elements are in the same equivalence class of $\pi_1 \cdot \pi_2$ if and only if they are both in one equivalence class of π_1 as well as in one of π_2.

An FSM[1] is a six-tuple $(\mathcal{S}, s^i, \Sigma, \Omega, \delta, \omega)$, where $\mathcal{S}$ is the set of states, s^i the initial state, Σ/Ω the set of input/output alphabets, $\delta : \mathcal{S} \times \Sigma \mapsto \mathcal{S}$ the transition function, and ω the output function. For a Moore machine, $\omega : \mathcal{S} \mapsto \Omega$; for a Mealy machine, $\omega : \mathcal{S} \times \Sigma \mapsto \Omega$. Given an FSM M, its output and transition functions define an equivalence relation, denoted $\equiv_M$, and thus induce a partition, denoted π_M, over the state space of M.

In this paper we concentrate on equivalence relations on a set of states. Two states s_1 and s_2 are equivalent, satisfying $s_1 \equiv_M s_2$, if and only if by using them as initial states, no input sequence can result in different output sequences. To approximate state equivalence, we define a *k-equivalence relation*, denoted $\cong_M^k$, and say two states s_1 and s_2 are *k-equivalent*, satisfying $s_1 \cong_M^k s_2$, if and only if they are indistinguishable under all input sequences with length up to k. Also say two states (or FSMs) are *k-distinguishable* if k is the shortest length of input sequences that differentiate them. We denote the partition associated with $\cong_M^k$ as π_M^k. To derive π_M from the approximation, we have $\pi_M = \pi_M^k$ if $\pi_M^k = \pi_M^{k-1}$ for large enough k, that is, a fixpoint has been reached. Similarly, we define a *$\langle k \rangle$-equivalence relation*, denoted as $\cong_M^{\langle k \rangle}$. Two states s_1 and s_2 satisfy $s_1 \cong_M^{\langle k \rangle} s_2$ if and only if, by using them as initial states, the outputs *at the k^{th} step* are equal for any length-k input sequence. The corresponding partition of $\cong_M^{\langle k \rangle}$ is denoted $\pi_M^{\langle k \rangle}$. By definition, we can derive the following lemma.

Lemma 1: For a Moore machine and $k \geq 1$,
$\pi_M^{\langle k \rangle} = \{E_i \mid s_1, s_2 \in E_i$ iff $\delta(s_1, x), \delta(s_2, x) \in E_j \in \pi_M^{\langle k-1 \rangle}$, for any $x \in \Sigma$, for some $j\}$
and
$\pi_M^{\langle 0 \rangle} = \{E_i \mid s_1, s_2 \in E_i$ iff $\omega(s_1) = \omega(s_2)\}$.
For a Mealy machine and $k \geq 2$,
$\pi_M^{\langle k \rangle} = \{E_i \mid s_1, s_2 \in E_i$ iff $\delta(s_1, x)$, $\delta(s_2, x) \in E_j \in \pi_M^{\langle k-1 \rangle}$, for any $x \in \Sigma$, for some $j\}$,
$\pi_M^{\langle 0 \rangle} = \{\mathcal{S}\}$
and
$\pi_M^{\langle 1 \rangle} = \{E_i \mid s_1, s_2 \in E_i$ iff $\omega(s_1, x) = \omega(s_2, x)$, $\forall x \in \Sigma\}$.

Proof: The base cases are direct results of the definition. Now we show the connection between $\pi_M^{\langle k \rangle}$ and $\pi_M^{\langle k-1 \rangle}$. There exists a length-$(k-1)$ input sequence to distinguish two states s_1 and s_2 at the output of the $(k-1)^{st}$ step if and only if s_1 and s_2 are in different equivalence classes of $\pi_M^{\langle k-1 \rangle}$. Therefore two states, say s_3 and s_4, cannot be distinguished at the output of the k^{th} step if and only if their successor states, i.e. $\delta(s_3, x)$ and $\delta(s_4, x)$, are in the same equivalence class of $\pi_M^{\langle k-1 \rangle}$ for any $x \in \Sigma$. ■

The connection between $\pi_M^{\cdot}$ and $\pi_M^{\langle \cdot \rangle}$ is indicated in Proposition 1.

Proposition 1: For an FSM M, two states are in the same equivalence class defined by π_M^k if and only if they are in the same equivalence class of $\pi_M^{\langle 0 \rangle}$, of $\pi_M^{\langle 1 \rangle}$, ..., and of $\pi_M^{\langle k \rangle}$.

B. Functional Decomposition

In this paper we adopt functional decomposition [17] for partitioning the state space to identify equivalent states and to verify sequential equivalence. In functional decomposition, variables of a Boolean function are divided into two disjoint subsets, the *bound set* and the *free set*. In BDD-based functional decomposition [12], bound set variables are ordered above free set ones. A *cutset* $\mathcal{C}$ of the BDD is the set of (downward) edges which cross the boundary defined by the bound set and free set variables. A node is called an *equivalence node* if there exists an edge, $e \in \mathcal{C}$, directed to it.

For a Boolean function $f(\vec{\lambda}, \vec{\mu})$, we can interpret the specification of the bound set variables $\vec{\lambda}$ and free set variables $\vec{\mu}$ as a partition over the space spanned by $\vec{\lambda}$, denoted Λ. That is, the set of all paths from the root of a BDD to an equivalence node forms an equivalence class. Each such set represents a subspace of Λ. Two minterms λ_1 and λ_2 in Λ are equivalent under arbitrary assignments of the free set variables, i.e. $\forall \vec{\mu}\ (f(\lambda_1, \vec{\mu}) = f(\lambda_2, \vec{\mu}))$, if and only if their corresponding paths in the BDD lead to the same equivalence node.

Given a set of Boolean functions $\{f_1, \ldots, f_k\}$, which do not necessarily have common supports, we can always expand these to the same Boolean space spanned by the union of the input variables of all functions. Let the bound set variables be $\vec{\lambda}$. Then the free set variables $\vec{\mu}$ are all variables excluding those in $\vec{\lambda}$. Suppose we want to find the equivalence classes of the minterms in Λ, such that two minterms λ_1 and λ_2 are equivalent under arbitrary assignments of all other variables, i.e. $\forall \vec{\mu}, \forall i\ (f_i(\lambda_1, \vec{\mu}) = f_i(\lambda_2, \vec{\mu}))$, if and only if these two minterms are in the same equivalence class. To represent equivalence classes by a BDD as in the single function case, we can construct a *hyper-function* $\mathcal{F}$ [9] of

[1]This paper only considers completely specified deterministic FSMs.

$\{f_1, \ldots, f_k\}$ by adding $\lceil \log_2 k \rceil$, new free set binary variables, $\vec{\eta}$, to encode these functions. Assume the overall free set variables become $\vec{\mu}'$. Thus two minterms λ_1 and λ_2 in Λ have $\forall \vec{\mu}' \ (\mathcal{F}(\lambda_1, \vec{\mu}') = \mathcal{F}(\lambda_2, \vec{\mu}'))$ if and only if their corresponding paths in the BDD of $\mathcal{F}$ lead to the same equivalence node.

III. Identification of State Equivalence

To find a minimum state FSM, equivalent to a given one, equivalent states are identified. Since each state in an equivalence class (of reachable states) can represent the entire class, the number of states of the minimum state FSM equals the number of the equivalence classes of the original FSM. This section proposes a more direct, in the sense that we deal with equivalence classes instead of equivalence relations, approach than those of [16], [14] to locate equivalent states. Given an FSM, we show that BDD-based functional decomposition can be exploited to extract equivalence classes of states.

Our approach seems conceptually similar to that in [8], where an FSM with n states is represented by n shared n-terminal BDDs. However, functional decomposition does not apply in this representation. As a result, the basic operations are of fundamental difference. Moreover, since our computation operates directly on the output and transition functions, it is representatively more efficient than the previous work.

A. State Equivalence vs. Functional Decomposition

In the base cases, $\pi_M^0 = \pi_M^{\langle 0 \rangle}$ for a Moore machine M and $\pi_M^1 = \pi_M^{\langle 1 \rangle}$ for a Mealy machine, output function ω plays the central role, as indicated in Lemma 1. Examining the case for a Moore machine M, we can see that ω serves directly as the characteristic function for π_M^0. On the other hand, the characteristic function of π_M^1 of a Mealy machine M is not clearly indicated by ω. We relate BDD-based functional decomposition to the computation of this characteristic function. In general ω is composed of a set of binary variables $\{\omega_1, \omega_2, \ldots, \omega_k\}$. According to Section II-B, we have to construct the hyper function $\mathcal{F}$ of $\{\omega_1, \omega_2, \ldots, \omega_k\}$. The supports of $\mathcal{F}$ consist of three parts: state variables $\vec{s}$, primary inputs $\vec{x}$, and new added variables $\vec{\eta}$ for encoding the hyper-function. Let $\vec{s}$ be the bound set variables and the rest be the free set. Accordingly, the equivalence nodes of the BDD of $\mathcal{F}$ represent the equivalence classes of π_M^1. Paths from the root to an equivalence node are states in a corresponding equivalence class. At this point we can ignore the functions represented by these equivalence nodes. That is, we can get rid of the BDD structures below these nodes. By re-encoding these nodes using alphabet Ψ, (introducing $\lceil \log_2 \log_2 |N| \rceil$ binary variables suffices to re-express N equivalence nodes because $\lceil \log_2 \log_2 |N| \rceil$ variables can generate at least N different functions, i.e. N nodes in BDD), we can have a characteristic function ψ for π_M^1 of a Mealy machine M, $\psi : \mathcal{S} \mapsto \Psi$.

Playing a similar trick, we show how to compute the characteristic function of $\pi_M^{\langle k \rangle}$, $k = 1$ or 2 for a Moore or Mealy machine respectively. Assume $\psi : \mathcal{S} \mapsto \Psi$ is the characteristic function derived from the last iteration (for both types of machines). Then the composition function $\psi \circ \delta$, $\psi(\delta(s, x))$, plays exactly the same role as ω in a Mealy machine, from which we have shown how to derive a characteristic function of $\pi_M^{\langle 1 \rangle}$. Consequently, by functional decomposition of the hyper-function of $\psi \circ \delta$, we have a characteristic function of $\pi_M^{\langle 1 \rangle}$ for Moore and $\pi_M^{\langle 2 \rangle}$ for Mealy machine M. The algorithm is summarized in Figure 1. The function call is denoted as *CompNewPartition*. By Lemma 1, we can derive the following theorem.

```
input: $\psi$ — characteristic function of $\pi^{\langle k-1 \rangle}$,
       $\tau$ — function to be composed
output: characteristic function of $\pi^{\langle k \rangle}$
begin
  01  form hyper-function $\mathcal{F}$ of $\psi \circ \tau$
  02  build BDD of $\mathcal{F}$ with state variables above others
  03  re-encode equivalence nodes and simplify BDD
  04  return new characteristic function
end
```

Fig. 1. Algorithm *CompNewPartition*: Compute New Partition

Theorem 1: Given the characteristic function of $\pi_M^{\langle k-1 \rangle}$ and $\delta : \mathcal{S} \times \Sigma \mapsto \mathcal{S}$ as the function to be composed, *CompNewPartition* generates the characteristic function of $\pi_M^{\langle k \rangle}$, where $k \geq 1$ (≥ 2) for Moore (Mealy) machine M.

B. Algorithm for Equivalent State Identification

To identify equivalent states, we have to compute π_M^k until it equals π_M^{k-1}; then $\pi_M = \pi_M^k$. Theorem 2 provides three alternatives to derive π_M. Its proof is supported by Lemma 2, also restated as Lemma 3.

Lemma 2: Consider an FSM with transition function $\delta : \mathcal{S} \times \Sigma \mapsto \mathcal{S}$. Let π_1 and π_2 be two arbitrary partitions on $\mathcal{S}$. For $s_1, s_2 \in \mathcal{S}$,
$\delta(s_1, x)$ and $\delta(s_2, x)$ are in the same equivalence class of π_1 and of π_2 for any $x \in \Sigma$
if and only if
$\delta(s_1, x)$ and $\delta(s_2, x)$ are in the same equivalence class of $\pi_1 \cdot \pi_2$ for any $x \in \Sigma$.

Proof: Let R_1 and R_2 be corresponding equivalence relations of π_1 and π_2 respectively.

($\rightarrow$) The condition we have implies $\{(\delta(s_1, x), \delta(s_2, x))\} \subseteq R_i$, $i = 1, 2$, $\forall x$. Thus $\{(\delta(s_1, x), \delta(s_2, x))\} \subseteq R_1 \cap R_2$, $\forall x$. Since $R_1 \cap R_2$ is the equivalence relation of $\pi_1 \cdot \pi_2$, the proof follows.

($\leftarrow$) From $\{(\delta(s_1, x), \delta(s_2, x))\} \subseteq R_1 \cap R_2$, $\forall x$, we obtain $\{(\delta(s_1, x), \delta(s_2, x))\} \subseteq R_i$, $i = 1, 2$, $\forall x$. That is, $\delta(s_1, x)$ and $\delta(s_2, x)$ are in the same equivalence class of π_1 and of π_2 for any $x \in \Sigma$. ∎

Lemma 3: For an FSM with transition function δ, assume π_1 and π_2 are two partitions over the state space. Let ψ_1, ψ_2 and $\psi_{1 \cdot 2}$ be the characteristic functions of π_1, π_2, and $\pi_1 \cdot \pi_2$ respectively. For characteristic functions $\psi_1' =$ *CompNewPartition*(ψ_1, δ), $\psi_2' =$ *CompNewPartition*(ψ_2, δ), and $\psi_{1 \cdot 2}' =$ *CompNewPartition*$(\psi_{1 \cdot 2}, \delta)$, their corresponding partitions satisfy $\pi_1' \cdot \pi_2' = \pi_{1 \cdot 2}'$.

```
input: an FSM $M = (\mathcal{S}, s^i, \Sigma, \Omega, \delta, \omega)$
output: characteristic function of $\pi_M$
begin
  01  if $M$ is a Moore machine then $\psi_- := \omega$
  02          else $\psi_- :=$ CompNewPartition(identity fn, $\omega$)
  03  $\psi_+ :=$ CompNewPartition($\psi_-$, $\delta$)
  04  while num. equiv. nodes of $\psi_+ \neq$ that of $\psi_-$ do
  05          $\psi_- := \psi_+$
  06          $\psi_+ :=$ CompNewPartition($\psi_+$, $\delta$)
  07          $\psi_+ :=$ combine $\psi_+$ and $\psi_-$
  08  return $\psi_+$
end
```

Fig. 2. Algorithm *IDES1*: Identify Equivalent States (Equation 1)

Theorem 2: Given an FSM M, for a positive integer k,

$$\begin{aligned} \pi_M^k &= \pi_M^{k^-} \cdot \pi_M^{k-1} && (1) \\ &= \begin{cases} \pi_M^{k^-} \cdot \pi_M^0 & \text{if } M \text{ is a Moore machine} \\ \pi_M^{k^-} \cdot \pi_M^1 & \text{if } M \text{ is a Mealy machine} \end{cases} && (2) \\ &= \pi_M^{\langle 0 \rangle} \cdot \pi_M^{\langle 1 \rangle} \cdots \pi_M^{\langle k \rangle} && (3) \end{aligned}$$

where $\pi_M^{k^-} = \{E_i \mid s_1, s_2 \in E_i$ iff $\delta(s_1, x)$ and $\delta(s_2, x)$ are in the same equivalence class of π_M^{k-1} for any $x \in \Sigma\}$.

Proof: We prove these equations by the order 3, 2, 1.

Equation 3: By the definition of π_M^k, states in an equivalence class are indistinguishable under length-k input sequences. According to Proposition 1, no outputs at steps from 0 to k can distinguish two states if and only if the states lie in the same equivalence class of $\pi_M^{\langle 0 \rangle}$, of $\pi_M^{\langle 1 \rangle}$, ..., and of $\pi_M^{\langle k \rangle}$. Thus, by Lemma 2, the states stay in the same equivalence class of $\pi_M^{\langle 0 \rangle} \cdot \pi_M^{\langle 1 \rangle} \cdots \pi_M^{\langle k \rangle}$.

Equation 2: Following the result of Equation 3, we get $\pi_M^{k-1} = \pi_M^{\langle 0 \rangle} \cdot \pi_M^{\langle 1 \rangle} \cdots \pi_M^{\langle k-1 \rangle}$. Suppose we use the characteristic function of π_M^{k-1} and transition function δ as the inputs to *CompNewPartition*. By Theorem 1 and Lemma 3, the output of the algorithm is $\pi_M^{k^-}$, that is, the characteristic function of $\pi_M^{\langle 1 \rangle} \cdot \pi_M^{\langle 2 \rangle} \cdots \pi_M^{\langle k \rangle}$ for a Moore machine or of $\pi_M^{\langle 2 \rangle} \cdot \pi_M^{\langle 3 \rangle} \cdots \pi_M^{\langle k \rangle}$ for a Mealy machine. Make a product partition with the initial partition induced by the outputs. We derive π_M^k. (Note that π_M^0 is redundant for a Mealy machine.)

Equation 1: By expressing $\pi_M^{k^-}$ and π_M^{k-1} in the product forms of $\pi_M^{\langle \cdot \rangle}$'s as in the proof of Equation 2, the equation follows. ■

Based on Equations 1, 2 and 3 to derive π_M, Figures 2, 3 and 4 sketch three algorithms, denoted as *IDES1*, *IDES2* and *IDES3* respectively. In these pseudo codes, "combine" a set of characteristic functions means using the procedure in Figure 1 except $\mathcal{F}$ is the hyper-function of the set of characteristic functions.

These algorithms terminate in a finite number of iterations. *IDES1* and *IDES2* converge because the partitions over finite states are refined continuously and the number of equivalence classes grows monotonically. On the other hand, because $\pi_M^{\langle k \rangle}$ in general is not a refinement of $\pi_M^{\langle k-1 \rangle}$, *IDES3* cannot simply determine the fixpoint by comparing the numbers of equivalence nodes in $\psi_M^{\langle k-1 \rangle}$ and $\psi_M^{\langle k \rangle}$. Therefore, it is more expensive to do fixpoint analysis. In general one should check whether or not new equivalence classes are created over previous partitions.

```
input: an FSM $M = (\mathcal{S}, s^i, \Sigma, \Omega, \delta, \omega)$
output: characteristic function of $\pi_M$
begin
  01  if $M$ is a Moore machine then $\psi^i := \omega$
  02          else $\psi^i :=$ CompNewPartition(identity fn, $\omega$)
  03  $\psi_- := \psi^i$
  04  $\psi_+ :=$ CompNewPartition($\psi_-$, $\delta$)
  05  while num. equiv. nodes of $\psi_+ \neq$ that of $\psi_-$ do
  06          $\psi_- := \psi_+$
  07          $\psi_+ :=$ CompNewPartition($\psi_+$, $\delta$)
  08          $\psi_+ :=$ combine $\psi_+$ and $\psi^i$
  09  return $\psi_+$
end
```

Fig. 3. Algorithm *IDES2*: Identify Equivalent States (Equation 2)

```
input: an FSM $M = (\mathcal{S}, s^i, \Sigma, \Omega, \delta, \omega)$
output: characteristic function of $\pi_M$
begin
  01  $\psi^{\langle 0 \rangle} :=$ identity function
  02  if $M$ is a Moore machine
  03          then $\psi^{\langle 0 \rangle} := \omega$
  04                  $\psi^{\langle 1 \rangle} :=$ CompNewPartition($\psi^{\langle 0 \rangle}$, $\delta$)
  05          else $\psi^{\langle 1 \rangle} :=$ CompNewPartition($\psi^{\langle 0 \rangle}$, $\omega$)
  06  $k := 1$
  07  while fixpoint not reached do
  08          $k := k + 1$
  09          $\psi^{\langle k \rangle} :=$ CompNewPartition($\psi^{\langle k-1 \rangle}$, $\delta$)
  10  return combine $\psi^{\langle i \rangle}$, $i = 0, 1, \ldots, k-1$
end
```

Fig. 4. Algorithm *IDES3*: Identify Equivalent States (Equation 3)

Although Figure 2 and Figure 3 look quite similar, the major difference is in combining two characteristic functions in Line 7 and Line 8 respectively. Despite keeping one more characteristic function, *IDES2* could require less memory than *IDES1* because ψ^i has a simpler BDD representation than ψ_-. On the other hand, although *IDES3* keeps all the characteristic functions along iterations, it has maximal flexibility to arrange the combination of them to reduce peak memory consumption.

C. Robust Equivalent State Identification

The limitations of equivalent state identification using BDD-based functional decomposition result from the explicit representation of equivalence classes and the restricted BDD variable ordering. In this section we propose some possible techniques to reduce BDD sizes.

Using any underestimated unreachable states as the don't care set, we can assign each such unreachable state to any equivalence class of reachable states. This flexibility enables the simplification of characteristic functions. However, because these algorithms use the number of equivalence classes to decide fixpoints, the number of equivalence classes with solely unreachable states should be kept as a constant during the iterations. (Note that if unreachable states are not used as don't cares, there is no such restriction.) Otherwise, we have to complicate the fixpoint condition by testing if an equivalence class is contained in the don't care set. Claim 1 shows BDD *constrain* [5] is a good simplification operator satisfying this requirement. On the contrary, BDD *restrict* [5] violates it. However, a BDD *restrict* followed by a *constrain* is a good operation.

Claim 1: Given a Boolean function $f(\vec{\lambda}, \vec{\mu})$ with bound set and free set variables $\vec{\lambda}$ and $\vec{\mu}$ respectively, assume Λ is the space spanned by $\vec{\lambda}$. Let $c(\vec{\lambda})$ be the characteristic function of the care set of Λ. Then $constrain(f, c)$ eliminates all equivalence nodes whose corresponding equivalence classes are contained in the don't care set, and preserves all other equivalence nodes.

Proof: Since BDD structures below equivalence nodes are irrelevant, we can think of f to be another function $g : \Lambda \mapsto N$, where N is the set of equivalence nodes. As $constrain(g, c)$ has its range equal to the image $\{g(\lambda) \mid c(\lambda) = \text{TRUE}, \forall \lambda \in \Lambda\}$, equivalence nodes not in this image disappear from the range and those in this image remain in the range. (On the other hand, the *restrict* operator could increase c to c', $c \subseteq c'$. Although equivalence nodes in the original image are kept, some with solely unreachable states might exist.) ∎

To reduce the impact of the restricted BDD variable ordering, we can use the following strategy. Within the allowed threshold of BDD size, find the variable ordering such that the lowest state variable is as high as possible. Treat this variable and those above it as bound set variables; all others are the free set. Then compact the BDD such that every node under the cutset is an equivalence node. Work on the new smaller BDD, and apply variable re-ordering to it based on the same strategy, incrementally throwing away unnecessary variables. On the other hand, since this ordering restriction emerges only from functional decomposition, arbitrary ordering can be used in other BDD manipulations. This restricted ordering is needed only when counting the number of equivalence classes and in constraining BDD with respect to the reachable state subspace.

Directly building a single hyper-function of a set of (binary) functions $\{f_1, \ldots, f_k\}$ may be impractical. Fortunately, this can be avoided by computing equivalence classes incrementally. For instance, first perform functional decomposition on f_1. For each resultant equivalence class, use it as the care set and others as the don't care set. Hence there is a greater chance to build a hyper-function for the simplified functions of $f_2, \ldots, f_k$. (If it fails, we can deepen the recursion level to extract more don't cares.) Conducting functional decomposition on it, the equivalence classes in the care set are encoded using new binary functions. In this way, BDD sizes are kept small. This approach trades time for memory.

We can also explore flexibility to reduce a partition before using it to compute a new partition. Given two partitions π_1 and π_2, we say any $\pi_1^\dagger$ ($\neq \pi_1$) satisfying $\pi_1^\dagger \cdot \pi_2 = \pi_1 \cdot \pi_2$ is a *reduced partition* of π_1 with respect to π_2. In particular, a simpler reduced partition, whose characteristic function has a smaller BDD size, is of interest. Theorem 3 states the validity of this flexibility.

Proposition 2: If $\pi_d \leq \pi_c$ holds for two partitions π_c and π_d, there exists a partition π_x such that $\pi_c \cdot \pi_x = \pi_d$.

Lemma 4: Assume partitions π, π' and π_c satisfy $\pi \cdot \pi_c = \pi' \cdot \pi_c$. If any π_d satisfies $\pi_d \leq \pi_c$, then $\pi \cdot \pi_d = \pi' \cdot \pi_d$.

Proof: By $\pi_d \leq \pi_c$ and Proposition 2, assume there exist a partition π_x such that $\pi_c \cdot \pi_x = \pi_d$. For $\pi \cdot \pi_c = \pi' \cdot \pi_c$, we derive $\pi \cdot \pi_c \cdot \pi_x = \pi' \cdot \pi_c \cdot \pi_x$, i.e. $\pi \cdot \pi_d = \pi' \cdot \pi_d$. ∎

Assume after certain iterations of refinement, the overall (product) partition is π_o. Let π_y be a new (not overall) partition after one more iteration, and let $\pi_y^\dagger$ be a reduced partition of π_y with respect to any π_x, such that $\pi_o \leq \pi_x$. (Let $\psi_\diamond$ denote the characteristic function of $\pi_\diamond$ for any subscript $\diamond$.) We have

Theorem 3: For ψ_z = *CompNewPartition*(ψ_y, δ) and ψ_z' = *CompNewPartition*($\psi_y^\dagger$, δ), equality $\pi_o \cdot \pi_y \cdot \pi_z = \pi_o \cdot \pi_y \cdot \pi_z'$ holds.

Proof: Let $\psi_{\triangleleft \cdot \triangleright}$ denote the characteristic function of $\pi_\triangleleft \cdot \pi_\triangleright$, for any subscripts $\triangleleft$ and $\triangleright$. In addition, $(\pi)^*$ is used to denote the partition with characteristic function *CompNewPartition*(ψ, δ), for any partition π with characteristic function ψ.

By the definition of a reduced partition, $\pi_y^\dagger \cdot \pi_x = \pi_y \cdot \pi_x$. Since $\pi_o \leq \pi_x$, equation $\pi_y^\dagger \cdot \pi_o = \pi_y \cdot \pi_o$ holds according to Lemma 4. So $(\pi_y^\dagger \cdot \pi_o)^* = (\pi_y \cdot \pi_o)^*$. From Lemma 3, we get $(\pi_y^\dagger)^* \cdot (\pi_o)^* = (\pi_y)^* \cdot (\pi_o)^*$. Since $\pi_z' = (\pi_y^\dagger)^*$ and $\pi_z = (\pi_y)^*$, then $\pi_z' \cdot (\pi_o)^* = \pi_z \cdot (\pi_o)^*$. Also from Theorem 2, $\pi_o \cdot \pi_y \leq (\pi_o)^*$. Hence by Lemma 4, $\pi_o \cdot \pi_y \cdot \pi_z = \pi_o \cdot \pi_y \cdot \pi_z'$. ∎

In the light of Theorem 3, an algorithm can be implemented by modifying *IDES2* and *IDES3* as follows. Keep a set of characteristic functions to represent the overall partition. Compute new partitions based only on an essential partition, which consists of equivalence classes that refine the previous overall partition. In this manner, the BDD size is kept small and the iterative computation is sped up.

IV. Verification of Sequential Equivalence

The proposed technique can be applied for sequential verification. The following two propositions form the basis of our equivalence checking. The first states a property that two equivalent FSMs must have.

Proposition 3: Given two equivalent FSMs M_1 and M_2 with sets of equivalence classes π_{M_1} and π_{M_2} respectively, assume expunging unreachable states from π_{M_1} and π_{M_2} results in $\pi_{M_1}^\flat$ and $\pi_{M_2}^\flat$ respectively. Then there exists a bijection $f : \pi_{M_1}^\flat \mapsto \pi_{M_2}^\flat$, where f reflects the state isomorphism of M_1 with M_2.

On the other hand, to show the equivalence between two FSMs, Proposition 4 gives necessary and sufficient conditions.

Proposition 4: M_1 and M_2, with initial states s_1^i and s_2^i respectively, are equivalent if and only if there exists a bijection $f : \pi_{M_1}^\flat \mapsto \pi_{M_2}^\flat$ (f reflects the state isomorphism of M_1 with M_2), and $E_2 = f(E_1)$ with $s_1^i \in E_1 \in \pi_{M_1}^\flat$ and $s_2^i \in E_2 \in \pi_{M_2}^\flat$.

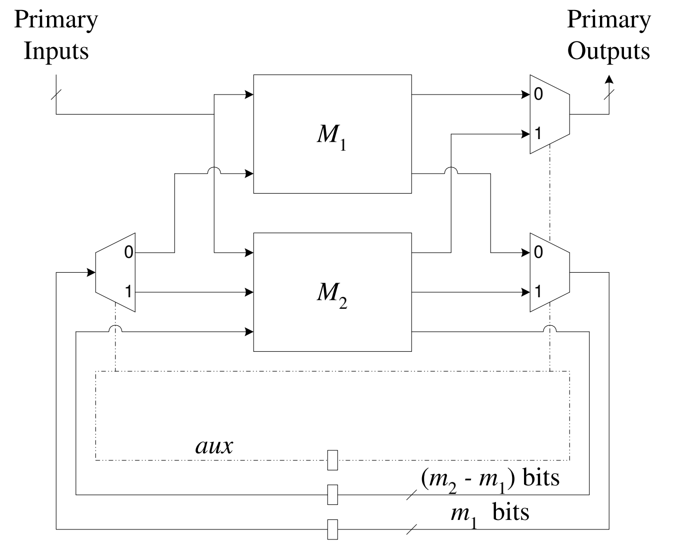

Fig. 5. Multiplexed Machine.

Based on Proposition 4, we can extend the identification of state equivalence to sequential equivalence checking. In order to pose the problem of verification as the identification of state equivalence, the *multiplexed machine* is introduced.

A. The Multiplexed Machine

To check equivalence between two FSMs M_1 and M_2 with m_1 and m_2 registers respectively, assume without loss of generality $m_2 \geq m_1$. Their multiplexed machine, denoted $M_{1\bowtie 2}$, is depicted in Figure 5. The two FSMs share the same primary inputs. Their corresponding outputs are multiplexed as a set of global primary outputs. To minimize the state variables of $M_{1\bowtie 2}$, for every next state variable of M_1, we pair it arbitrarily with one of M_2. This pair is then multiplexed before being fed to a register, whose output is then demultiplexed to recover the current state variables for M_1 and M_2. In addition, one self-looped *auxiliary state variable*, abbreviated *aux*, is added, which controls all multiplexers and demultiplexers as indicated by the dotted lines in Figure 5. The value of *aux* remains the same as its initial value. Let $M_{1\bowtie 2}$ select M_1 and M_2 when *aux* has values 0 and 1 respectively. No matter what the initial value of *aux* is, the multiplexed machine functions the same as M_1 and M_2 if they are equivalent. In the verification, we can imagine that *aux* is in a superposition status, possessing values 0 and 1 simultaneously. (Note that, without changing its functionality, the multiplexed machine can be simplified by omitting the demultiplexers. That is, replacing each demultiplexer, we directly connect its input to outputs. Also it is worth mentioning that choosing any subset of the next state variables of M_1 to be paired is valid. Suppose, in the extreme case, we choose an empty subset. Then *aux* and the multiplexers for outputs are unnecessary. The multiplexed machine, therefore, degenerates into two separate machines. The corresponding verification is discussed in Section IV-E.)

B. Algorithm for Sequential Equivalence Checking

Given two FSMs M_1 and M_2 with initial states[2] s_1^i and s_2^i respectively, without loss of generality assume their multiplexed machine $M_{1\bowtie 2}$ selects M_1 (M_2) while *aux* equals 0 (1). Their equivalence can be verified based on Lemma 5, a consequence of Proposition 4.

Lemma 5: M_1 and M_2 are equivalent if and only if $\pi_{M_{1\bowtie 2}}$ has at least one reachable state with *aux* bit 0 and at least one with 1 in every equivalence class containing any reachable state, and has initial states (s_1^i with *aux* 0 and s_2^i with *aux* 1) in the same equivalence class.

Proof: Assume $f : \pi^\flat_{M_1} \mapsto \pi^\flat_{M_2}$ reflects the state isomorphism between M_1 and M_2. Let $E_2 = f(E_1)$ for $E_1 \in \pi^\flat_{M_1}$ and $E_2 \in \pi^\flat_{M_2}$. Then after adding the *aux* bit, original reachable states (including initial states) $s_1 \in E_1$ and $s_2 \in E_2$ must be within the same equivalence class of $\pi_{M_{1\bowtie 2}}$. Thus every equivalence class of $\pi_{M_{1\bowtie 2}}$ containing any reachable state must have at least one (with *aux* 0) contributed from M_1 and one (with *aux* 1) from M_2. ■

By iterative refinement of the state space as in the identification of state equivalence, equivalence classes of states for $M_{1\bowtie 2}$ can be derived whenever the fixpoint has been reached. According to Lemma 5, both conditions are checked. However, the first condition implies that we need to know reachable states of both M_1 and M_2. Fortunately, the first condition is redundant, i.e. as long as the second condition is satisfied, so is the first. This property is stated in Theorem 4. As a result, reachability analysis can be completely eliminated.

Theorem 4: M_1 and M_2 are equivalent if and only if $\pi_{M_{1\bowtie 2}}$ has initial states, namely s_1^i with *aux* 0 and s_2^i with *aux* 1, within the same equivalence class.

Proof: By contradiction, we show that the first condition in Lemma 5 is redundant. Assume $\pi_{M_{1\bowtie 2}}$ has initial states in the same equivalence class E^i, and there exists an equivalence class E containing reachable states all with *aux* bits 0 (or 1 does not matter). Therefore, $E \neq E^i$. For any reachable state of E, there must be a reachable state, say s, (with *aux* bit 0) that transitions to it. This transition makes s have no equivalent reachable states from M_2. Therefore the equivalence class containing s has all reachable states with *aux* bits 0. Continuing this argument, we conclude that E^i must exclude the state, s_2^i with *aux* 1. Hence a contradiction arises. ■

Further, rather than checking that the condition of Theorem 4 is satisfied in the overall partition of the state space, validity can be verified on the new partition at each iteration. The correctness of this variant is based on Proposition 1. As the BDD representation of the current partition is obtained, it is of linear time complexity in the number of state variables to test if two initial states are within the same equivalence class. Consequently this checking can be done efficiently in each iteration. Figure 6 outlines the overall procedure for sequential equivalence checking.

Remark: In theory k FSMs can be verified simultaneously by introducing $\lceil \log_2 k \rceil$ auxiliary state variables to control

[2]To simplify the discussion, we assume each FSM has a single initial state. This can be straightforwardly generalized to a set of initial states.

```
input: two FSMs under equivalence checking
output: YES if equivalent; NO otherwise
begin
  01  build the multiplexed machine $M$
  02  compute the init. partition $\pi_M^i$
  03  if init. states NOT in an equiv class of $\pi_M^i$
  04          then return NO
  05  while fixpoint not reached do
  06          compute $\pi_M^{\mathrm{new}}$
  07          refine the overall partition and simplify $\pi_M^{\mathrm{new}}$
  08          if init. states NOT in an equiv class of $\pi_M^{\mathrm{new}}$
  09                  then return NO
  10  return YES
end
```

Fig. 6. Algorithm: Verify Sequential Equivalence

the k-to-1 multiplexers of their corresponding multiplexed machine.

C. Robust Sequential Equivalence Checking

To make the verification procedure more robust, the techniques and restrictions listed in Section III-C are also applicable here. Instead of repeating them, this section is concerned with those that are particular to verification.

Verifying each primary output and/or characteristic function separately could substantially reduce the number of encountered equivalence classes. The numbers of equivalence classes induced by individual primary outputs may be exponentially smaller than those induced by all of the primary outputs. The correctness of this separation is inferred from Lemma 2. It is interesting to notice that the cone of inference reduction has been automatically taken care of due to this separation, i.e., irrelevant state variables with respect to the considered primary output disappear.

Although reachability analysis is unnecessary, any underestimation of unreachable states of M_1 and/or M_2 can be used as a don't care set to simplify BDD expressions and to reduce unnecessary state refinements. Theorem 5 shows the correctness of such simplification and the maximal don't care set for the multiplexed machine. However, as mentioned in Section III-C, the fixpoint condition should be preserved to ensure the algorithm terminates.

Theorem 5: The equivalence condition of M_1 and M_2 is invariant under don't care simplification by unreachable states of $M_{1\bowtie 2}$, that is, unreachable states of M_1 with *aux* 0 together with those of M_2 with *aux* 1.

Proof: Because state transition is irrelevant to the simplification of characteristic functions of partitions, the proof of Theorem 4 still holds.

Assume the sets of reachable (unreachable) states of M_1 and M_2 are R_1 (U_1) and R_2 (U_2) respectively. Let α be the auxiliary state variable. Then the reachable states of $M_{1\bowtie 2}$ is $(\neg\alpha\wedge R_1)\vee(\alpha\wedge R_2)$. Its complement is $\neg(\neg\alpha\wedge R_1)\wedge\neg(\alpha\wedge R_2)$ = $(\alpha \vee U_1) \wedge (\neg\alpha \vee U_2)$ = $(\alpha \wedge U_2) \vee (\neg\alpha \wedge U_1) \vee (U_1 \wedge U_2)$ = $(\alpha \wedge U_2) \vee (\neg\alpha \wedge U_1)$. ∎

Besides don't care simplification, the partitioned state space can be reduced further according to the following theorem.

Theorem 6: Let $\pi^k_{M_{1\bowtie 2}}$ be the partition associated with the k-equivalence relation of $M_{1\bowtie 2}$. Then equivalence checking is invariant under the reduction of $\pi^k_{M_{1\bowtie 2}}$ by collapsing the set $\{E \in \pi^k_{M_{1\bowtie 2}} \mid \forall s \in E.$ The *aux* bit of s is $0.\}$ of equivalence classes into one equivalence class and collapsing $\{E \in \pi^k_{M_{1\bowtie 2}} \mid \forall s \in E.$ The *aux* bit of s is $1.\}$ into another.

Proof: It is clear that M_1 and M_2 are equivalent only if the collapsed equivalence classes are unreachable from the initial states. Also, if the condition holds for all $k \geq 0$, then M_1 and M_2 are equivalent.

Since we collapse the equivalence classes of M_1 and of M_2 separately, states from one machine which have transitions to these equivalence classes do not have corresponding equivalent states from the other machine. Besides, as state transition relations are not affected by the collapsing, the equivalence relation among other states, which cannot transition to these equivalence classes, remains intact. Hence the verification is invariant under this reduction. ∎

Corollary 1: For two FSMs M_1 and M_2 with n_1 and n_2 equivalence classes respectively, the number of equivalence classes can be kept at most $\min\{n_1, n_2\} + 1$ in our sequential equivalence checking with the use of the collapsing process in Theorem 6. Hence the number of variables introduced to generate equivalence nodes is at most $\lceil \log_2 \log_2(\min\{n_1, n_2\} + 1)\rceil$. Assume the n-input FSMs M_1 and M_2 have m_1 and m_2 state variables respectively. Then by verifying each output separately, the total number of variables in our verification is at most $(n+\max\{m_1, m_2\}+1+\lceil \log_2 \log_2(\min\{n_1, n_2\}+1)\rceil)$ $\leq (n + \max\{m_1, m_2\} + 1 + \lceil \log_2(\min\{m_1, m_2\} + 1)\rceil)$.

In the construction of the multiplexed machine, a multiplexer, selecting state variables, pairs a state variable from M_1 with any unpaired one from M_2. Since this pairing is arbitrary (and thus can be adaptively changed on-the-fly), an optimization problem is to maximize the BDD sharing between M_1 and M_2, and to simplify the consequent BDD manipulations. Heuristics can be derived based on the cone of inference reduction and functional similarity. The former pairs two state variables which are supports of two similar sets of primary outputs; the latter pairs two state variables with similar transition functionalities. In the extreme case, when comparing two identical copies of an FSM, we can possibly reduce the BDD such that it is as if these is only one machine.

D. Error Tracing and Shortest Distinguishing Sequence

Given two states s_1 and s_2 which are k-distinguishable at an output of an FSM $M = (\mathcal{S}, s^i, \Sigma, \Omega, \delta, \omega)$, this section illustrates how to derive a length-k input sequence differentiating them.

Since s_1 and s_2 are k-distinguishable, their corresponding BDD paths lead to different equivalence nodes in some characteristic function at the k^{th} refinement. Let the functions represented by these two BDD nodes be f_1 and f_2. (Notice that, f_1 and f_2 should be the functions before re-encoding and simplification mentioned in Section III-A.) Then any solution, say x^*, to (f_1 XOR f_2) provides the k^{th} distinguishing input vector. On the other hand, two states $s'_1 = \delta(s_1, x^*)$ and $s'_2 = \delta(s_2, x^*)$ are $(k-1)$-distinguishable. They result in the distinguishability of s_1 and s_2 at the k^{th} refinement. Similarly the $(k-1)^{st}$ distinguishing input vector can be obtained. Repeating this process backward, one can derive a shortest distinguishing sequence to trace an error.

E. State Space Partitioning on Separate Machines

The multiplexed machine is not the only construction that extends state equivalence to machine equivalence. To prove the equivalence of M_1 and M_2, the state variables can be kept disjoint while the inputs are shared. Thereby their state spaces are partitioned separately but simultaneously by maintaining two sets of shared BDDs during functional decomposition. Again, they are equivalent if and only if their initial states lead to the same equivalence node when the fixpoint is reached.

In the case of the multiplexed machine, state variables of M_1 and M_2 are merged by multiplexers. As mentioned in Section IV-C, the register pairing affects the cone of inference and BDD manipulations. By state space partitioning on separate machines, the interference among state variables is removed. However the major drawback is that there is no BDD sharing between M_1 and M_2 above the cutset. Notice that, although the number of state variables in this case is the same as for the product machine, the verification is still in the sum state space.

F. State Space Partitioning on Product Machine

Verification by state space partitioning also works for the product machine as well. It can be done by slight modifications of [16], [14], previously known as the *backward state traversal* [7]. We refer to it as state space partitioning on the product machine.

When compared to state space partitioning on the multiplexed machine, this approach has more flexibility in BDD variable ordering. However, this flexibility prevents simplification by the *restrict* or *constrain* operator with respect to the reachable states because this might corrupt the represented equivalence relation.

V. Analysis

This section consists of two parts. First, some verification properties, independent of the implementation of a design, are analyzed. Second, we discuss circuit implementation related effects on the sequential equivalence checking problem.

A. Implementation-Independent Aspects

Given an FSM taking a total of n iterations in state space partitioning, its *partition structure* is defined as an ordered sequence $\widehat{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, where σ_i denotes the accumulated number of equivalence classes at the i^{th} iteration. Thus $\sigma_i < \sigma_{i+1}$, for $i = 1, \ldots, n-1$, and $\sigma_n = \sigma_{n+1}$.

Theorem 7: Any two equivalent FSMs must have the same partition structure in their reachable state subspace.

Proof: Assume two equivalent FSMs M and M' have sets of equivalence classes π and π' respectively in their reachable state subspace. Therefore, according to Proposition 3, there exists a bijection $f : \pi \mapsto \pi'$.

Suppose M and M' have different partition structures. Since the state space is continuously refined in fixpoint computation, there must exist l- and k-distinguishable state pairs (s_1, s_2) and (s'_1, s'_2) respectively, such that $l > k$, $s_1 \in E_1 \in \pi$, $s'_1 \in f(E_1) \in \pi'$, $s_2 \in E_2 \in \pi$, and $s'_2 \in f(E_2) \in \pi'$. Let δ and δ' be the transition functions of M and M' respectively. Then pairs $\{(s_i, s_j) \mid \exists x(\delta^{-1}(s_i, x), \delta^{-1}(s_j, x)) = (s_1, s_2)\}$ must be at least $(l-1)$-distinguishable and at least one of them is $(l-1)$-distinguishable. Similarly, $\{(s'_i, s'_j) \mid \exists x({\delta'}^{-1}(s'_i, x), {\delta'}^{-1}(s'_j, x)) = (s'_1, s'_2)\}$ are at least $(k-1)$-distinguishable and at least one of them, say (s'_{i^*}, s'_{j^*}), is $(k-1)$-distinguishable. Let x^* be the input such that $(\delta'(s'_1, x^*), \delta'(s'_2, x^*)) = (s'_{i^*}, s'_{j^*})$. Also let $(s_{i^*}, s_{j^*}) = (\delta(s_1, x^*), \delta(s_2, x^*))$. Suppose $s_{i^*} \in E_{i^*} \in \pi$ and $s_{j^*} \in E_{j^*} \in \pi$. Then $s'_{i^*} \in f(E_{i^*}) \in \pi'$ and $s'_{j^*} \in f(E_{j^*}) \in \pi'$. Now since (s_{i^*}, s_{j^*}) is at least $(l-1)$-distinguishable and $(l-1) > (k-1)$, we are ready to have recursive reasoning for (s_{i^*}, s_{j^*}) and (s'_{i^*}, s'_{j^*}). At some point of the recursion, we will reach the situation that (s'_{i^*}, s'_{j^*}) can be differentiated by some output while (s_{i^*}, s_{j^*}) can not. This violates the base cases of Lemma 1. Hence M and M' must have the same partition structure. ■

Therefore, partition structures in reachable state subspace form a signature for equivalent FSMs. This may not be true for the entire state space. However, even without the knowledge of state reachability, the following holds.

Theorem 8: Given two FSMs M_1 and M_2 converging in m and n steps respectively in state space partitioning, their product machine converges in no more than $\min\{m, n\}$ steps in state space partitioning.

Proof: In state space partitioning, the product machine has state "equivalence relation" $\equiv_P$ over (ordered) pairs of states, (s_1, s_2) with $s_1 \in S_1$ and $s_2 \in S_2$, where S_1 and S_2 are the sets of states of M_1 and M_2 respectively. Notice that $\equiv_P$ may not satisfy reflexive and symmetric laws. Nevertheless, the transitive law holds for the ordered pairs of states. Since the transitive law is maintained during the fixpoint computation, it is clear that once one machine converges, so does the product machine. On the other hand, this state partitioning procedure does not refine the state subspace $\{s \mid s \in S_1, \forall s_2 \in S_2, (s, s_2) \notin \equiv_P, \text{ or } s \in S_2, \forall s_1 \in S_1, (s_1, s) \notin \equiv_P\}$. Hence it could converge in less than $\min\{m, n\}$ steps. ■

Theorem 9: Given two FSMs M_1 and M_2 converging in m and n steps respectively in state space partitioning, then their multiplexed machine converges in exactly $\max\{m, n\}$ steps in state space partitioning. With the state space reduced by Theorem 6 in each iteration, the computation converges in the same step as the state space partitioning on their product machine.

Proof: The construction of the multiplexed machine is designed to match corresponding equivalence classes between M_1 and M_2. State space partitioning on the combined machine has no effect on the partition of the state subspace spanned by any individual FSM. Once each subspace of M_1 and M_2

has reached a fixpoint in state partitioning, so has the space of their combined machine. Therefore the combined machine converges in exactly $\max\{m, n\}$ steps.

When the state space is reduced by Theorem 6 in each iteration, the fixpoint computation does not refine the state subspace spanned by the collapsed equivalence classes. The state space is partitioned in the same way as that of the product machine. Hence the multiplexed and product machines converge in the same step in state space partitioning. ■

In contrast, for state traversal of an FSM, although we can similarly define a *traversal structure* to be the sequence of numbers of reached states, we can not use it as a signature. Moreover, even if the traversal depths for two FSMs are known, they merely provide a lower bound on the depth of the product machine. No strong argument like Theorems 8 and 9 is possible.

The following theorem shows the connection between the number of refinements in state partitioning and the depth of state traversal.

Theorem 10: Given two k-distinguishable FSMs M_1 and M_2, both state-traversal and state-partition based approaches differentiate them at the k^{th} step.

Proof: Since state traversal on the product machine of M_1 and M_2 implicitly enumerates all possible transitions, clearly any discrepancy can be observed in the shortest steps.

On the other hand, for state partition, since the initial states from M_1 and M_2 must be k-distinguishable in the combined machine of M_1 and M_2. The theorem follows. ■

As a consequence,

Corollary 2: Given two FSMs M_1 and M_2, let $M_{1\times 2}$ be their product machines. Assume p is the number of refinements in state partitioning on $M_{1\times 2}$, and q is the depth of state traversal on $M_{1\times 2}$. Then $\min\{p, q\}$ is an upper bound on the number of iterations required for equivalence checking.

In other words, following Corollary 2, if $p > q$, we can conclude the equivalence of M_1 and M_2 in q refinements of state partitioning on $M_{1\times 2}$. Similarly, if $p < q$, their equivalence can be confirmed in p steps of state traversal on $M_{1\times 2}$.

Also, it follows immediately from Theorem 8 that,

Corollary 3: Given two FSMs, M_1 and M_2, converging in m and n steps respectively in state space partitioning, their equivalence can be concluded in no more than $\min\{m, n\}$ steps in state partitioning on their multiplexed machine.

B. Implementation-Dependent Aspects

Retiming [13] is an important technique in sequential circuit optimization. There are two types of atomic moves in retiming, namely forward (from inputs to outputs) moves and backward (from outputs to inputs) moves across functional blocks. Here we investigate their effects on the number of equivalence classes in the state space. Suppose an FSM M_b is retimed from another FSM M_f using only backward moves across a functional block with function $f : S_b \mapsto S_f$, where S_b and S_f are the state spaces of M_b and M_f respectively. (Equivalently M_f is retimed from M_b using forward moves across the functional block with function f.)

Proposition 5: Two states s_b and s'_b of M_b are equivalent, i.e. $s_b \equiv_{M_b} s'_b$, if and only if their corresponding states $f(s_b)$ and $f(s'_b)$ of M_f are equivalent, i.e. $f(s_b) \equiv_{M_f} f(s'_b)$.

Proposition 6: If $s_b \equiv_{M_b} s'_b$, then the corresponding states of s_b, s'_b, $f(s_b)$, and $f(s'_b)$ in the multiplexed machine $M_{b\bowtie f}$ of M_b and M_f are in the same equivalence class of $M_{b\bowtie f}$.

Theorem 11: The number of equivalence classes of M_b is not greater than that of M_f.

Proof: Since f is a *total function*, i.e. f is well defined for all states of M_b, the theorem follows from Proposition 5. ■

Theorem 12: The number of equivalence classes of M_f is greater than that of M_b if and only if there exists a state s of M_f such that $f^{-1}(s) = \emptyset$ and $s \not\equiv_{M_f} f(s_b)$, $\forall s_b \in S_b$.

Proof: The theorem follows from Proposition 5. ■

Similar arguments of Theorems 11 and 12 were used in [19] for the discussion of the validity of retiming.

VI. Experimental Results

Using the VIS [1] environment, we compared three equivalence checking techniques, namely,

STPM – state traversal on the product machine,
SPPM – state partitioning on the product machine, and
SPMM– state partitioning on the multiplexed machine.

The experiments were conducted on a Linux machine with a Pentium III XEON 700 MHz CPU and 2 Gb of RAM.

For *STPM* and *SPPM* the VIS sequential verification command is used. Dynamic variable reordering is turned on and the hybrid method [15], considered the state-of-the-art technique for image computation, is used. For *SPMM*, variable reordering is enabled when appropriate.

To demonstrate the relative power of the three techniques, we first compare a set of benchmark circuits against themselves. (Although combinational checking suffices in this circumstance, we are only interested in sequential methods.) In general, combinational equivalence checking should be tried in situations where there is structural similarity. The techniques of this paper aim at situations where there is no such similarity. The self-comparison benchmarks are used to compare the methods on a large set of examples. Care is taken not to exploit similarity by using a method for pairing state variables which considers only the cones of inference of the primary outputs. To further emphasize that no similarity is being exploited, a second set experiments is done comparing circuits against their retimed versions.

An argument why self-comparison is sufficient for the experiments is Proposition 3, which states that two different implementations, M_1 and M_2, must have corresponding equivalence classes in the reachable set of states. Thus the reachable state spaces of $M_{1\bowtie 2}$, $M_{1\bowtie 1}$ and $M_{2\bowtie 2}$ all have the same number of equivalence classes. Also even if M_1 and M_2 have incomparable numbers of equivalence classes in the whole state spaces, by Corollary 1 the number of equivalence classes encountered by *SPMM* is at most $\min\{n_1, n_2\} + 1$, where n_i is the number of equivalence classes of M_i, $i = 1, 2$. Thus conclusions drawn from self-comparison experiments should remain valid for general comparisons.

TABLE I
PROFILES OF BENCHMARK CIRCUITS

Circuit	In	Out	Reg	Reach (Depth)
s1196	14	14	18	2616 (2)
s298	3	6	14	218 (18)
349	9	11	15	2625 (6)
s400/s444	3	6	21	8865 (150)
s420.1	18	1	16	65536 (65535)
s499	1	22	22	22 (21)
s526/s526n	3	6	21	8868 (150)
s641	35	24	19	1544 (6)
s713	35	23	19	1544 (6)
s953	16	23	29	504 (10)
s967	16	23	29	549 (10)
s991	65	17	19	524288 (3)
bigkey	262	197	224	1.17e+67 (2)
clma	382	82	33	158908 (411)
mm4a	7	4	12	832 (3)
mm9a	12	9	27	2.25e+7 (3)
mm9b	12	9	26	2.25e+7 (3)
mult16a	17	1	16	65535 (16)
sbc	40	56	28	154593 (9)
control	33	21	35	119 (6)
IFetchControl2	27	38	59	2.50e+8 (27)
IFetchControl3	27	38	61	1.00e+9 (27)
parsepack	9	65	70	3.70e+19 (9)
parsesys	9	65	312	2.21e+48 (103)
8085	18	27	193	N/A
bpb	9	4	36	6.87e+10 (32)
cbp_16_4	17	17	16	131072 (1)
cbp_32_4	33	33	32	4.29e+9 (1)
key	258	193	228	N/A
minmax5	8	5	15	12032 (3)
minmax10	13	10	30	1.79e+8 (3)
tbk-retime	6	3	49	2048 (3)

Before presenting empirical results in Tables III and IV, we provide the characteristics of the benchmark circuits in Tables I and II. Table I gives the profiles of the selected benchmarks from ISCAS89, LGSYNTH91, TEXAS97, VIS and TEXAS. Columns 2, 3 and 4 indicate the number of inputs, outputs and registers respectively. In addition, the number of reachable states and the corresponding traversal depth are provided in Column 5. (Here we reset uninitialized state variables to zero.)

Also the information of equivalence classes is included in Table II. As mentioned in Section IV-C, we can verify sequential equivalence by examining each primary output separately instead of treating them as a whole. The advantage is that we can reduce the peak memory requirements recording encountered equivalence classes. To provide strong evidence, Table II contains two parts of data. The first part, *Overall Partition*, in Columns 2 and 3 shows the number of equivalence classes induced by all primary outputs. The number in the following parentheses indicates the depth of refinement in the corresponding fixpoint computation. In contrast, the second part, *Worst Partial Partition*, in Columns 4 and 5 lists the largest number of equivalence classes induced by some primary output. The number in the following parentheses indicates the maximum depth of refinement among all outputs. Circuit `s991` is an example where separating verification tasks for each output makes a substantial reduction in the number of encountered equivalence classes. In the extreme case, the number of equivalence classes induced by all outputs can be exponentially (in the number of outputs) larger than those induced by individual outputs. Usually the separation of verification tasks lengthens the required refinement. However, as BDD manipulations could be simplified substantially, the run time can still be reduced in most cases. Further, within each part we compare the number (in the column marked *whole*) of equivalence classes in the whole state space to the number (in the column marked *reach*) of equivalence classes

TABLE II
CHARACTERISTICS OF EQUIVALENCE CLASSES OF BENCHMARK CIRCUITS

	Overall Partition		Worst Partial Partition	
Circuit	whole (rfn)	reach (rfn)	whole (rfn)	reach (rfn)
s1196	82944 (2)	1509 (2)	96 (3)	56 (3)
s298	8061 (16)	135 (12)	249 (24)	118 (20)
s349	18608 (5)	1801 (5)	248 (8)	35 (6)
s400	608448 (93)	8865 (93)	17174 (183)	8597 (183)
s420.1	65536 (32768)			
s444	608448 (93)	8865 (93)	17174 (183)	8597 (183)
s499	4.19e+6 (1)	22 (1)	24 (21)	22 (21)
s526	1.43e+6 (119)	8868 (93)	43068 (199)	8597 (183)
s526n	1.43e+6 (119)	8868 (93)	43068 (199)	8597 (183)
s641	294912 (1)	1480 (1)	24750 (8)	1248 (8)
s713	294912 (1)	1480 (1)	24750 (8)	1248 (8)
s953	N/A	504 (2)	42 (10)	35 (10)
s967	N/A	549 (2)	42 (10)	35 (10)
s991	327680 (1)		10 (2)	
bigkey	N/A		4 (2)	
clma	N/A		N/A	5950 (178)
mm4a	3616 (1)	712 (1)	452 (2)	217 (1)
mm9a	N/A		522244 (2)	260617 (1)
mm9b	N/A		N/A	260617 (1)
mult16a	65536 (16)	65535 (16)	65536 (16)	65535 (16)
sbc	N/A		N/A	23048 (10)
control	N/A	43 (2)	14 (6)	8 (5)
IF'hC'l2	N/A		N/A	9434 (37)
IF'hC'l3	N/A		N/A	8442 (39)
parsepack	N/A		18 (9)	10 (9)
parsesys	N/A		164 (21)	N/A
8085*	N/A		309619 (28)	N/A
bpb	N/A		512 (3)	
cbp_16_4	65536 (1)			
cbp_32_4	4.29e+9 (1)			
key	N/A		64 (7)	N/A
minmax5	30784 (1)	5520 (1)	1924 (2)	965 (2)
minmax10	1.07e+9 (1)	N/A	2.09e+6 (2)	1.04e+6 (1)
tbk-retime	16 (1)		16 (3)	

in the reachable subspace. As can be seen, in most instances this subset is fairly small when compared to the entire space. Since *SPMM* directly benefits from these reductions, it can easily verify some large instances which are unverifiable for *STPM* and *SPPM* as indicated in Tables III and IV, where the results for *SPPM* and *STPM* report the best of verifying combined outputs and verifying each output separately. From experience, *SPPM* has better results in verifying combined outputs for most circuits while *SPMM* has the opposite results. This might be explained by the fact that the performance of *SPPM* is not directly related to the encountered number of equivalence classes while that of *SPMM* is.

From the experiment in Table III we observe that, for *SPMM*, using a monolithic BDD as a characteristic function suffices for all verifiable benchmarks. The only exception is `sbc`, where an array of characteristic functions need to be maintained. Because using multiple characteristic functions usually complicates the fixpoint computation, it is in general more time consuming. Also we find that *SPMM* takes longer time than *STPM* and *SPPM* for circuits, such as `s382`, `s420.1`, etc., with numerous equivalence classes and deep refining processes. It is understandable because *SPMM* enumerates each equivalence class in every refining process.

For circuits like `s420.1`, where the depths of traversal and refinement are both exponential in the size of inputs, none of the three techniques is competent. However, for `s420.1`, since the depth of refinement is half of that of traversal, *SPPM* is about twice as fast as *STPM*. Notice that, as analyzed

TABLE III
SEQUENTIAL EQUIVALENCE CHECKING BETWEEN IDENTICAL CIRCUITS

	STPM		SPPM		SPMM	
Circuit	mem (Mb)	time (sec)	mem (Mb)	time (sec)	mem (Mb)	time (sec)
s1196	28.3	2.3	25.1	1.5	12.4	2.1
s298	7.8	0.2	16.4	1.0	8.7	0.9
s349	12.7	1.5	25.4	1.3	10.8	1.9
s400	12.8	4.9	43.1	4.8	56.6	448.8
s420.1	45.1	669.2	37.9	290.9	62.0	2.98e+5
s444	12.7	4.8	42.2	4.5	55.8	438.9
s499	299	157.1	16.5	1.0	8.6	0.2
s526	22.5	7.1	117.0	293.8	50.4	358.2
s526n	16.6	4.4	82.7	150.9	50.4	357.8
s641	11.9	0.7	27.4	0.6	39.5	3.3
s713	11.8	0.7	27.6	0.6	39.2	6.4
s953	11.3	0.1	27.9	0.8	11.9	1.1
s967	11.4	0.9	27.5	0.8	10.3	0.5
s991	35.4	26.4	64.9	11.6	10.7	0.3
bigkey	>2G	N/A	>2G	N/A	21.4	1.3
clma	142	134.6	>2G	N/A	117	4.30e+4
mm4a	8.6	0.3	7.7	0.1	15.3	0.9
mm9a	82.1	1.24e+5	58.9	16.6	244	4673.7
mm9b	>2G	N/A	>2G	N/A	693	3.12e+4
mult16a	8.5	0.2	8.4	0.1	87.8	126.1
sbc	>2G	N/A	>2G	N/A	537	1.29e+5
control	191	79.4	46.1	7.9	23.3	1.1
IF'hC'l2	>2G	N/A	N/A	>1.0e+6	258	1.37e+4
IF'hC'l3	>2G	N/A	N/A	>1.0e+6	259	1.38e+4
parsepack	>2G	N/A	64.9	110.9	19.0	1.2
parsesys	>2G	N/A	458	2.91e+4	102	45.9
8085*	>2G	N/A	>2G	N/A	793	3.06e+5
bpb	>2G	N/A	51.7	62.9	46.1	17.2
cbp_16_4	18.0	0.3	18.0	0.3	75.2	70.2
cbp_32_4	25.0	0.8	24.7	0.7	>2G	N/A
key	>2G	N/A	>2G	N/A	68.5	15.4
minmax5	27.3	0.8	28.1	0.6	26.0	12.2
minmax10	151	1694.9	47.2	2.3	733	8.75e+4
tbk-retime	>2G	N/A	>2G	N/A	84.2	112.3

in Section V-A, although the product machine has traversal depth 65535 (due to self-comparison), we can conclude the equivalence by traversing states at the 32768^{th} step even before the fixpoint is reached.

For cbp and minmax series of circuits, where depths are shallow, *STPM* and *SPPM* performs much better than *SPMM*, which needs to take care of numerous equivalence classes as listed in Table II. On the other hand, for minmax circuits, as discussed in [7], *SPPM* has a polynomial complexity in input sizes while *STPM* has an exponential one. In comparison, *SPPM* is the best choice for these cases.

Circuits key and bigkey are another extreme, which has a few equivalence classes. *SPMM* verifies them quite easily while both *STPM* and *SPPM* fail. In general for control logic *SPMM* performs much better than the other two. Microprocessor 8085 is an example, where *SPMM* verifies all the outputs except the sixteen for the address bus. (The results of 8085 in Tables II and III exclude these unverifiable outputs.) Other examples are control, IFetchControl2 and IFetch-Control3. On the other hand, due to the large number of outputs in IFetchControl2, IFetchControl3, clma, sbc, etc., *SPMM* takes a long time to verify them because it processes each output once at a time. Fortunately, these tasks can be parallelly verified to minimize the total completion time.

TABLE IV
SEQUENTIAL EQUIVALENCE CHECKING BETWEEN DIFFERENT IMPLEMENTATIONS OF SAME DESIGN

Circuit	Reg	STPM mem (Mb)	STPM time (sec)	SPPM mem (Mb)	SPPM time (sec)	SPMM mem (Mb)	SPMM time (sec)
`s208.1` / `s208.1-retime`	8 / 16	12.4	0.3	11.8	0.5	8.9	2.3
`s298` / `s298-retime`	14 / 34	12.7	0.3	21.8	1.7	9.6	0.7
`s386` / `s386-retime`	6 / 15	12.6	0.2	13.0	0.3	7.3	0.1
`s499` / `s499-retime`	22 / 41	437	196.9	690	401.3	10.7	1.8
`s510` / `s510-retime`	6 / 34	13.6	0.4	19.5	1.8	12.3	0.4
`s526` / `s526-retime`	21 / 58	48.3	24.3	237	2012.2	55.4	552.5
`s526n` / `s526n-retime`	21 / 64	48.4	41.5	204	5238.7	53.2	325.9
`s526-retime` / `s526n-retime`	58 / 64	>2G	N/A	982	1.26e+5	54.5	469.3
`s641` / `s641-retime`	19 / 18	37.5	1.9	41.1	1.9	29.3	9.7
`s991` / `s991-retime`	19 / 42	345	2431.9	139	760.8	74.3	134.6
`mult16a` / `mult16a-retime`	16 / 106	>2G	N/A	>2G	N/A	N/A	>1.0e+6
`tbk` / `tbk-retime`	5 / 49	56.1	10.3	70.1	79.2	46.2	6.6

In Table IV, the equivalence between a circuit and its retimed implementation is checked. Retimed circuits were obtained by using SIS [18], except for TEXAS benchmarks, `s641-retime` and `tbk-retime`. Other circuits, which are included Table III but absent from Table IV, either take too long for SIS to retime, or have incompatible initial states, created by the retiming. Table IV suggests that *SPMM* does not benefit particularly when self-comparison is done. (This is due to the fact that state variables are paired only by cone of inference of outputs. Otherwise, corresponding state variables are avoided to be paired together. Doing so destroys BDD sharing in the experiments of self-comparison.) This supports that the results of Table III are relevant for comparing the three methods. Also observe from Table IV that *SPMM* is relatively stable when moving from self-comparison to comparing against retimed versions. For example, for `s526` and `s526n`, the results in Tables III and IV are similar for *SPMM* but *STPM* and *SPPM* yield substantial variances. The stability of *SPMM* derives from the fact that it depends mainly on the maximum number of registers in the two designs plus the number of equivalence classes encountered.

Another view of Tables III and IV is shown in Table V, where the second and the third columns denote the numbers of wins in terms of smaller memory and time usage respectively, and the last gives the number of examples on which the method failed. This analysis indicates that *SPMM* is on average more efficient and more rugged than the other two methods.

TABLE V
OVERALL STATISTICS

Method	Wins in Memory	Wins in Time	Failed
STPM	11	12	13
SPPM	7	15	10
SPMM	28	21	2

We did not experiment with the equivalence checking between inequivalent circuits. However the expectation is that, according to Theorem 10, all of the three verification techniques can report the non-equivalence in the same iteration, say in the n^{th} iteration. To generate a counterexample on the other hand, both *STPM* and *SPPM* have time complexity $O(n)$ while *SPMM* has $O(n^2)$. This difference results from the fact that, in *SPMM*, the input information of the previous iterations is thrown away when equivalence nodes are re-expressed using newly introduced variables.

To summarize the results, the major limitation of *SPMM* is the encountered number of equivalence classes during verification. In contrast, *STPM* and *SPPM* do not suffer the same limitation because equivalence classes are not explicitly represented in the BDDs. For a circuit with a not-so-deep depth of refinement and a “reasonable” number ($\leq \sim 10^6$) of equivalence classes per output, *SPMM* has a great chance of verifying it. On the other hand, due to the fact that the number of equivalence classes in the reachable state subspace is invariant under different implementations, *SPMM* tends to be the most robust verification technique.

VII. CONCLUSIONS

This paper consists of two parts: the identification of equivalent states and the verification of sequential equivalence. We show that the former can be done efficiently by BDD-based functional decomposition. By introducing the multiplexed machine, we can verify sequential equivalence by means of state partitioning in the sum space, a new possibility to do formal equivalence checking. In high speed designs, a great portion of registers are for timing speed-up rather than increasing the number of equivalence classes of states. In such cases, state space partitioning would become preferable to state space traversal.

A major advantage of the new verification technique is the substantial reduction in the number of state variables.

Compared to product machine based techniques, our approach almost halves the number of state variables. Although there is an intrinsic restriction on the BDD variable ordering, to overcome it and minimize the BDD sizes, several techniques are proposed. These make our algorithm even more promising.

ACKNOWLEDGMENTS

The presentation of this paper was greatly strengthened through anonymous reviewers' comments.

REFERENCES

[1] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa, "VIS: A System for Verification and Synthesis," in *Proc. Int'l Conf. Computer Aided Verification*, pp. 428-432, 1996.
[2] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers*, vol. 35, no. 8, pp. 677-691, Aug. 1986.
[3] G. Cabodi, S. Quer, and F. Somenzi, "Optimizing Sequential Verification by Retiming Transformations," in *Proc. Design Automation Conf.*, pp. 601-606, 2000.
[4] O. Coudert, C. Berthet, and J. C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," in *Proc. Int'l Workshop Automatic Verification Methods for Finite State Systems*, 1989.
[5] O. Coudert and J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits," in *Proc. Int'l Conf. Computer-Aided Design*, pp. 126-129, 1990.
[6] C. A. J. van Eijk, "Sequential Equivalence Checking Based on Structural Similarities," *IEEE Trans. on Computer-Aided Design*, vol. 19, no. 7, pp. 814-819, July 2000.
[7] T. Filkorn, "A Method for Symbolic Verification of Synchronous Circuits," in *Proc. Int'l Symp. Computer Hardware Description Languages and their Applications*, pp. 249-259, 1991.
[8] J. G. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm, "Mona: Monadic Second-Order Logic in Practice," in *Proc. Int'l Workshop Tools and Algorithms for the Construction and Analysis of Systems, TACAS '95, LNCS 1019*, 1996.
[9] J.-H. Jiang, J.-Y. Jou, and J.-D. Huang, "Unified Functional Decomposition via Encoding for FPGA Technology Mapping," *IEEE Trans. on VLSI*, vol. 9, pp. 251-260, April 2001.
[10] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, 1978.
[11] A. Kuehlmann and J. Baumgartner, "Transformation-Based Verification Using Generalized Retiming," in *Proc. Int'l Conf. Computer Aided Verification*, pp. 104-117, 2001.
[12] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula, "BDD Based Decomposition of Logic Functions with Application to FPGA Synthesis," in *Proc. Design Automation Conf.*, pp. 642-647, 1993.
[13] C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems," *Journal of VLSI and Computer Systems*, vol. 1, no. 1, pp. 41-67, 1983.
[14] B. Lin, H. J. Touati, and A. R. Newton, "Don't Care Minimization of Multi-Level Sequential Logic Networks," in *Proc. Int'l Conf. Computer-Aided Design*, pp. 414-417, 1990.
[15] I.-H. Moon, J. H. Kukula, K. Ravi, and F. Somenzi, "To Split or to Conjoin: The Question in Image Computation," in *Proc. Design Automation Conf.*, pp. 23-28, 2000.
[16] C. Pixley, "A Theory and Implementation of Sequential Hardware Equivalence," *IEEE Trans. on Computer-Aided Design*, vol. 11, no. 12, pp. 1469-1478, Dec. 1992.
[17] J. P. Roth and R. M. Karp, "Minimization over Boolean Graphs," *IBM J. Res. Dev.*, pp. 227-238, 1962.
[18] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephen, R. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, Univ. California, Berkeley, 1992.
[19] V. Singhal, C. Pixley, R. L. Rudell, and R. K. Brayton, "The Validity of Retiming Sequential Circuits," in *Proc. Design Automation Conf.*, pp. 316-321, 1995.