Functional Dependency for Verification Reduction*

Jie-Hong R. Jiang and Robert K. Brayton

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Abstract. The existence of functional dependency among the state variables of a state transition system was identified as a common cause of inefficient BDD representation in formal verification. Eliminating such dependency from the system compacts the state space and may significantly reduce the verification cost. Despite the importance, how to detect functional dependency *without or before knowing the reachable state set* remains a challenge. This paper tackles this problem by unifying two closely related, but scattered, studies — detecting signal correspondence and exploiting functional dependency. The prior work on either subject turns out to be a special case of our formulation. Unlike previous approaches, we detect dependency directly from transition functions rather than from reached state sets. Thus, reachability analysis is not a necessity for exploiting dependency. In addition, our procedure can be integrated into reachability analysis as an on-the-fly reduction. Preliminary experiments demonstrate promising results of extracting functional dependency without reachability analysis. Dependencies that were underivable before, due to the limitation of reachability analysis on large transition systems, can now be computed efficiently. For the application to verification, reachability analysis is shown to have substantial reduction in both memory and time consumptions.

1 Introduction

Reduction [12] is an important technique in extending the capacity of formal verification. This paper is concerned with property-preserving reduction [7], where the reduced model satisfies a property if and only if the original model does. In particular, we focus on reachability-preserving reduction for safety property verification using functional dependency.

The existence of dependency among state variables frequently occurs in state transition systems at both high-level specifications and gate-level implementations [17]. As identified in [11], such dependency may cause inefficient BDD [6] representation in formal verification. Moreover, it can be used in logic minimization [13, 17]. Its detection, thus, has potential impact on formal verification and logic synthesis, and has attracted extensive research in both domains (e.g., see

* This work was supported in part by NSF grant CCR-0312676, California Micro program, and our industrial sponsors, Fujitsu, Intel and Synplicity.

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 268–280, 2004.

[2, 13, 11, 8, 17]). The essence of all prior efforts [2, 13, 8, 17] can be traced back to *functional deduction* [5], where variable dependency is drawn from a single characteristic function. Thus, as a common path, the variable dependency was derived from the characteristic function of a reached state set. However, state transition systems of practical applications are often too complex to compute their reachable states, even though these systems might be substantially reduced only after variable dependency is known. An improvement was proposed in [8] to exploit the dependency from the currently reached state set in every iteration of a reachability analysis. The computation, however, may still be too expensive and may simplify subsequent iterations very little.

To avoid such difficulty, we take a different path to exploit the dependency. The observation is that the dependency among state variables originates from the dependency among transition functions[1]. In consideration of efficiency, some variable dependency can better be concluded directly from the transition functions rather than from the characteristic function of a reached state set. Therefore, the computation requires only *local* image computation. As the derived dependency is an invariant, it can be used by any BDD- or SAT-based model checking procedure to reduce the verification complexity. Since not all dependency can be discovered this way due to the imperfect information about state reachability, this method itself is an approximative approach. To complete the approximative computation, our procedure can be embedded into reachability analysis as an on-the-fly detection. Reachability analysis is thus conducted on a reduced model in each iteration. Our formulation leads to a unification of two closely related, but scattered, studies on detecting signal correspondence [10, 9] and exploiting functional dependency [11, 8].

2 Preliminaries and Notations

As a notational convention, a vector (or, an ordered set) $\boldsymbol{v} = \langle v_1, \ldots, v_n \rangle$ is specified in a bold-faced letter while its unordered version is written as $\{\boldsymbol{v}\} = \{v_1, \ldots, v_n\}$. In this case, n is the cardinality (size) of both $\boldsymbol{v}$ and $\{\boldsymbol{v}\}$, i.e., $|\boldsymbol{v}| = |\{\boldsymbol{v}\}| = n$. Also, when a vector $\boldsymbol{v}$ is partitioned into k sub-vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$, the convention $\langle \boldsymbol{v}_1; \ldots; \boldsymbol{v}_k \rangle$ denotes that $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ are combined into one vector with a proper reordering of elements to recover the ordering of $\boldsymbol{v}$.

This paper assumes, without loss of generality, that multi-valued functions are replaced with vectors of Boolean functions. The image of a Boolean functional vector $\boldsymbol{\psi}$ over a subset C of its domain is denoted as $\mathit{Image}(\boldsymbol{\psi}, C)$; the range of $\boldsymbol{\psi}$ is denoted as $\mathit{Range}(\boldsymbol{\psi})$. Let $\psi : \mathbb{B}^n \rightarrow \mathbb{B}$ be a Boolean function over variables $x_1, \ldots, x_n$. The **support set** of ψ is $\mathit{Supp}(\psi) = \{x_i \mid (\psi|_{x_i=0} \text{ XOR } \psi|_{x_i=1}) \neq \text{FALSE}\}$. For a characteristic function $F(\boldsymbol{x})$ over the set $\{\boldsymbol{x}\}$ of Boolean

[1] As state transition systems are often compactly representable in transition functions but not in transition relations, this paper assumes that transition functions are the underlying representation of state transition systems. Consequently, our formulation is not directly applicable to nondeterministic transition systems. The corresponding extension can apply the MOCB technique proposed in [11].

variables, its **projection** on $\{\boldsymbol{y}\} \subseteq \{\boldsymbol{x}\}$ is defined as $F[\{\boldsymbol{y}\}/\{\boldsymbol{x}\}] = \exists x_i \in \{\boldsymbol{x}\}\backslash\{\boldsymbol{y}\}.F(\boldsymbol{x})$. Also, we denote the identity function and its complement as $\Im$ and $\Im^\dagger$, respectively.

A state transition system $\mathcal{M}$ is a six-tuple $(S, I, \Sigma, \Omega, \boldsymbol{\delta}, \boldsymbol{\lambda})$, where S is a finite set of states, $I \subseteq S$ is the set of initial states, Σ and Ω are the sets of input and output alphabets, respectively, and $\boldsymbol{\delta} : \Sigma \times S \to S$ (resp. $\boldsymbol{\lambda} : \Sigma \times S \to \Omega$) is the transition function (resp. output function). As symbols and functions are in binary representations in this paper, $\mathcal{M}$ will be specified, instead, with a five-tuple $(I, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{\delta}, \boldsymbol{\lambda})$, where $\boldsymbol{r}$ (resp. $\boldsymbol{s}$) is the vector of Boolean variables that encodes the input alphabets (resp. states).

3 Functional Dependency

We formulate functional dependency for state transition systems in two steps. First, **combinational dependency** among a collection of functions is defined. Second, the formulation is extended to **sequential dependency**.

3.1 Combinational Dependency

Given two Boolean functional vectors $\boldsymbol{\phi} : \mathbb{B}^l \to \mathbb{B}^m$ and $\boldsymbol{\varphi} : \mathbb{B}^l \to \mathbb{B}^n$ over the same domain, we are interested in rewriting $\boldsymbol{\phi}$ in terms of a function of $\boldsymbol{\varphi}$. The condition when such a rewrite is feasible can be captured by a refinement relation, $\sqsubseteq \subseteq (\mathbb{B}^l \to \mathbb{B}^m) \times (\mathbb{B}^l \to \mathbb{B}^n)$, defined as follows.

Definition 1. *Given two Boolean functional vectors $\boldsymbol{\phi} : \mathbb{B}^l \to \mathbb{B}^m$ and $\boldsymbol{\varphi} : \mathbb{B}^l \to \mathbb{B}^n$, $\boldsymbol{\varphi}$* **refines** *$\boldsymbol{\phi}$ in $C \subseteq \mathbb{B}^l$, denoted as $\boldsymbol{\phi} \sqsubseteq_C \boldsymbol{\varphi}$, if $\boldsymbol{\phi}(a) \neq \boldsymbol{\phi}(b)$ implies $\boldsymbol{\varphi}(a) \neq \boldsymbol{\varphi}(b)$ for all $a, b \in C$.*

In other words, $\boldsymbol{\varphi}$ refines $\boldsymbol{\phi}$ in C if and only if $\boldsymbol{\varphi}$ is more distinguishing than $\boldsymbol{\phi}$ in C. (As the orderings within $\boldsymbol{\phi}$ and $\boldsymbol{\varphi}$ are not a prerequisite, our definition of refinement relation applies to two unordered sets of functions as well.) In the sequel, the subscription C will be omitted from the refinement relation $\sqsubseteq$ when C is the universe of the domain. Based on the above definition, the following proposition forms the foundation of our later development.

Proposition 1. *Given $\boldsymbol{\phi} : \mathbb{B}^l \to \mathbb{B}^m$ and $\boldsymbol{\varphi} : \mathbb{B}^l \to \mathbb{B}^n$, there exists a functional vector $\boldsymbol{\theta} : \mathbb{B}^n \to \mathbb{B}^m$ such that $\boldsymbol{\phi} = \boldsymbol{\theta} \circ \boldsymbol{\varphi} = \boldsymbol{\theta}(\boldsymbol{\varphi}(\cdot))$ over $C \subseteq \mathbb{B}^l$ if and only if $\boldsymbol{\phi} \sqsubseteq_C \boldsymbol{\varphi}$. Moreover, $\boldsymbol{\theta}$ is unique when restricting its domain to the range of $\boldsymbol{\varphi}$.*

For $\boldsymbol{\phi} = \boldsymbol{\theta} \circ \boldsymbol{\varphi}$, we call $\phi_1, \ldots, \phi_m \in \boldsymbol{\phi}$ the **functional dependents** (or, briefly, dependents), $\varphi_1, \ldots, \varphi_n \in \boldsymbol{\varphi}$ the **functional independents** (or, briefly, independents), and $\theta_1, \ldots, \theta_n \in \boldsymbol{\theta}$ the **dependency functions**.

Problem Formulation. The problem of detecting (combinational) functional dependency can be formulated as follows. Given a collection of Boolean functions $\boldsymbol{\psi}$, we are asked to partition $\boldsymbol{\psi}$ into two parts $\boldsymbol{\phi}$ and $\boldsymbol{\varphi}$ such that $\boldsymbol{\phi} = \boldsymbol{\theta}(\boldsymbol{\varphi})$. Hence,

the triple $(\boldsymbol{\phi}, \boldsymbol{\varphi}, \boldsymbol{\theta})$ characterizes the functional dependency of $\boldsymbol{\psi}$. We call such a triple a **dependency triplet**. Suppose $\boldsymbol{\varphi}$ cannot be further reduced in $(\boldsymbol{\phi}, \boldsymbol{\varphi}, \boldsymbol{\theta})$ by recognizing more functional dependents from $\boldsymbol{\varphi}$ with all possible modifications of $\boldsymbol{\theta}$. That is, $|\boldsymbol{\varphi}|$ is minimized; equivalently, $|\boldsymbol{\phi}|$ is maximized. Then the triplet maximally characterizes the functional dependency of $\boldsymbol{\psi}$. In this paper, we are interested in computing maximal functional dependency. (Although finding a maximum dependency might be helpful, it is computationally much harder than finding a maximal one as it is the supremum over the set of maximal ones.)

The Computation. In the discussion below, when we mention Boolean functional vectors $\boldsymbol{\phi}(\boldsymbol{x})$ and $\boldsymbol{\varphi}(\boldsymbol{x})$, we shall assume that $\boldsymbol{\phi} : \mathbb{B}^l \rightarrow \mathbb{B}^m$ and $\boldsymbol{\varphi} : \mathbb{B}^l \rightarrow \mathbb{B}^n$ with variable vector $\boldsymbol{x} : \mathbb{B}^l$. Notice that $Supp(\boldsymbol{\phi})$ and $Supp(\boldsymbol{\varphi})$ are subsets of $\{\boldsymbol{x}\}$. The following properties are useful in computing dependency.

Theorem 1. *Given functional vectors $\boldsymbol{\phi}$ and $\boldsymbol{\varphi}$, $\boldsymbol{\phi} \sqsubseteq \boldsymbol{\varphi}$ only if $Supp(\boldsymbol{\phi}) \subseteq Supp(\boldsymbol{\varphi})$.*

Corollary 1. *Given a collection of Boolean functions $\psi_1(\boldsymbol{x}), \ldots, \psi_k(\boldsymbol{x})$, if, for any $x_i \in \{\boldsymbol{x}\}$, ψ_j is the only function such that $x_i \in Supp(\psi_j)$, then ψ_j is a functional independent.*

With the support set information, Theorem 1 and Corollary 1 can be used as a fast screening in finding combinational dependency.

Theorem 2. *Given functional vectors $\boldsymbol{\phi}$ and $\boldsymbol{\varphi}$, $\boldsymbol{\phi} \sqsubseteq \boldsymbol{\varphi}$ if and only if $|Range(\boldsymbol{\varphi})| = |Range(\langle \boldsymbol{\phi}, \boldsymbol{\varphi} \rangle)|$.*

Theorem 3. *Let $\theta_i \in \boldsymbol{\theta}$ be the corresponding dependency function of a dependent $\phi_i \in \boldsymbol{\phi}$. Let $\Theta_i^0 = \{\boldsymbol{\varphi}(\boldsymbol{x}) | \phi_i(\boldsymbol{x}) = 0\}$ and $\Theta_i^1 = \{\boldsymbol{\varphi}(\boldsymbol{x}) | \phi_i(\boldsymbol{x}) = 1\}$. Then $\phi_i \sqsubseteq \boldsymbol{\varphi}$ if and only if $\Theta_i^0 \cap \Theta_i^1 = \emptyset$. Also, θ_i has Θ_i^0, Θ_i^1, and $\mathbb{B}^n \backslash \{\Theta_i^0 \cup \Theta_i^1\}$ as its off-set, on-set, and don't-care set, respectively. That is, $\theta_i(\boldsymbol{\varphi}(\boldsymbol{x})) = \phi_i(\boldsymbol{x})$ for all valuations of $\boldsymbol{x}$.*

From Theorem 2, we know that the set $\{\boldsymbol{\varphi}\}$ of functional independents is as distinguishing as the entire set $\{\boldsymbol{\phi}\} \cup \{\boldsymbol{\varphi}\}$ of functions. Theorem 3, on the other hand, shows a way of computing dependency functions.

Given a collection $\{\boldsymbol{\psi}\}$ of Boolean functions, its maximal dependency can be computed with the procedure outlined in Figure 1. First, by Theorem 2, for each function $\psi_i \in \{\boldsymbol{\psi}\}$ we obtain the minimal subsets of $\{\boldsymbol{\psi}\}$ which refine ψ_i. Let the minimal refining subsets for ψ_i be $E_i^1, \ldots, E_i^k$. (Notice that $k \geq 1$ since ψ_i refines itself and, thus, $\{\psi_i\}$ is one of the subsets.) The calculation can be done with *local* image computation because by Theorem 1 and Corollary 1 we only need to consider subsets of functions in $\{\boldsymbol{\psi}\}$ which overlap with ψ_i in support sets. Second, we heuristically derive a minimal set of functional independents that refines all the functions of $\{\boldsymbol{\psi}\}$. Equivalently, for each ψ_i, some $E_i^{j_i}$ is selected such that the cardinality of $\bigcup_{i=1}^{|\boldsymbol{\psi}|} E_i^{j_i}$ is minimized. This union set forms the basis of representing all other functions. That is, functions in the union set are the functional independents; others are the functional dependents. Finally, by Theorem 3, dependency functions are obtained with respect to the selected basis.

CombinationalDependency
input: a collection $\{\psi\}$ of Boolean functions
output: a dependency triplet (ϕ, φ, θ)
begin
01 for each $\psi_i \in \{\psi\}$
02 derive minimal refining sets $E_1^i, \ldots, E_k^i$
03 select a minimal basis $\{\varphi\}$ that refines all $\psi_i \in \{\psi\}$
04 compute the dependency functions $\{\theta\}$ for $\{\phi\} = \{\psi\} \backslash \{\varphi\}$
05 **return** (ϕ, φ, θ)
end

Fig. 1. Algorithm: CombinationalDependency.

A Digression. There were other variant definitions of dependency (see [15] for more examples). The functional dependency defined in [5] (Section 6.9), which follows [15], is too weak to be applicable in our application. We, thus, resort to a stronger definition. As noted below, our definition turns out to be consistent with *functional deduction* (see [5], Chapter 8), which is concerned with the variable dependency in a single characteristic function.

We relate our formulation to functional deduction as follows. In functional deduction, variable dependency is drawn from a single characteristic function. Thus, to exploit the dependency among a collection of functions $\boldsymbol{\psi}(\boldsymbol{x})$, a single relation $\Psi(\boldsymbol{x}, \boldsymbol{y}) = \bigwedge_i (y_i \equiv \psi_i(\boldsymbol{x}))$ should be built, where y_i's are newly introduced Boolean variables. In addition, to derive dependency solely among $\{\boldsymbol{y}\}$, input variables $\{\boldsymbol{x}\}$ should be enforced in the *eliminable subset* [5]. With the foregoing transformation, variable dependency in functional deduction coincides with our defined functional dependency. A similar result of Theorem 3 was known in the context of functional deduction. Compared to the relational-oriented functional deduction, our formulation can be understood as more functional-oriented, which is computationally more practical.

3.2 Sequential Dependency

Given a state transition system $\mathcal{M} = (I, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{\delta}, \boldsymbol{\lambda})$, we consider the detection of functional dependency among the set $\{\boldsymbol{\delta}\}$ of transition functions. More precisely, detecting the sequential dependency of $\mathcal{M}$ is equivalent to finding $\boldsymbol{\theta}$ such that $\boldsymbol{\delta}$ is partitioned into two vectors: the dependents $\boldsymbol{\delta}_{\boldsymbol{\phi}}$, and the independents $\boldsymbol{\delta}_{\boldsymbol{\varphi}}$. Let $\{\boldsymbol{s}\} = \{\boldsymbol{s}_{\boldsymbol{\phi}}\} \cup \{\boldsymbol{s}_{\boldsymbol{\varphi}}\}$ be such that the valuations of $\boldsymbol{s}_{\boldsymbol{\phi}}$ and $\boldsymbol{s}_{\boldsymbol{\varphi}}$ are updated by $\boldsymbol{\delta}_{\boldsymbol{\phi}}$ and $\boldsymbol{\delta}_{\boldsymbol{\varphi}}$, respectively. Then $\boldsymbol{\theta}$ specifies the dependency of $\mathcal{M}$ by $\boldsymbol{s}_{\boldsymbol{\phi}} = \boldsymbol{\theta}(\boldsymbol{s}_{\boldsymbol{\varphi}})$ and $\boldsymbol{\delta}_{\boldsymbol{\phi}} = \boldsymbol{\theta}(\boldsymbol{\delta}_{\boldsymbol{\varphi}})$, i.e., $\boldsymbol{\delta}_{\boldsymbol{\phi}}(\boldsymbol{r}, \langle \boldsymbol{\theta}(\boldsymbol{s}_{\boldsymbol{\varphi}}); \boldsymbol{s}_{\boldsymbol{\varphi}} \rangle) = \boldsymbol{\theta} \circ \boldsymbol{\delta}_{\boldsymbol{\varphi}}(\boldsymbol{r}, \langle \boldsymbol{\theta}(\boldsymbol{s}_{\boldsymbol{\varphi}}); \boldsymbol{s}_{\boldsymbol{\varphi}} \rangle)$.

Sequential dependency is more relaxed than its combinational counterpart because of the reachability nature of $\mathcal{M}$. The derivation of $\boldsymbol{\theta}$ shall involve a fixed-point computation, and can be obtained in two different ways, the greatest fixed-point (gfp) and the least fixed-point (lfp) approaches, with different optimality and complexity. Our discussions start from the easier gfp computation, and continue with the more complicated lfp one. The optimality, on the other hand, is usually improved from the gfp to the lfp computation.

Remark 1. We mention a technicality regarding the set I of initial states. In general, the combinational dependency among transition functions may not hold for the states in I because I may contain dangling states. (A state is called **dangling** if it has no predecessor states. Otherwise, it is **non-dangling**.) To overcome this difficulty, a new set I' of initial states is defined. Let I' be the set of states which are one-step reachable from I. Now, since all states in I' are non-dangling, the calculated dependency holds for I'. On the other hand, the set of reachable states from I is identical to that from I' except for some states in I. In the verification of safety properties, such a substitution is legitimate as long as states in I satisfy the underlying property to be verified. In our discussion, unless otherwise noted, we shall assume that the set of initial states consists of only non-dangling states.

The Greatest Fixed-Point Calculation. In the gfp calculation, state variables are treated functionally independent of each other initially. Their dependency is then discovered iteratively. Combinational dependency among transition functions is computed in each iteration. The resultant dependency functions are substituted backward in the subsequent iteration for the state variables of their corresponding functional dependents. Thereby, the transition functions and previously derived dependency functions are updated. More precisely, let $\boldsymbol{\theta}^{(i)}$ be the set of derived dependency functions for $\boldsymbol{\delta}^{(i)}$ at the ith iteration. For j from $i-1$ to 1, the set $\boldsymbol{\theta}^{(j)}(\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i-1)})$ of dependency functions is updated in order with $\boldsymbol{\theta}^{(j)}(\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)}) = \boldsymbol{\theta}^{(j)}(\langle \boldsymbol{\theta}^{(j+1)}(\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)}); \ldots; \boldsymbol{\theta}^{(i)}(\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)}); \boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)} \rangle)$. After the updates of $\boldsymbol{\theta}^{(j)}$'s, $\boldsymbol{\delta}^{(i+1)}$ is set to be $\boldsymbol{\delta}_{\boldsymbol{\varphi}}^{(i)}(\boldsymbol{r}, \langle \boldsymbol{\theta}^{(1)}(\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)}); \ldots; \boldsymbol{\theta}^{(i)}(\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)}); \boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)} \rangle)$, where $\{\boldsymbol{\delta}_{\boldsymbol{\varphi}}^{(i)}\} \subseteq \{\boldsymbol{\delta}\}$ corresponds to the functional independents of $\boldsymbol{\delta}^{(i)}$. At the $(i+1)$st iteration, the combinational dependency among $\boldsymbol{\delta}^{(i+1)}$ is computed. The iteration terminates when the size of the set of functional independents cannot be reduced further. The termination is guaranteed since $|\boldsymbol{\delta}^{(i)}|$ decreases monotonically. In the end of the computation, the final $\boldsymbol{\theta}$ is simply the collection of $\boldsymbol{\theta}^{(i)}$'s, and the final set of functional independents is $\boldsymbol{\delta}_{\boldsymbol{\varphi}}^{(k)}$, where k is the last iteration. The computation is summarized in Figure 2, where the procedure *CombinationalDependencyRestore* is similar to *CombinationalDependency* with a slight difference. It computes the dependency among the set of functions given in the first argument in the same way as *CombinationalDependency*. However, the returned functional dependents and independents are the corresponding functions given in the second argument instead of those in the first argument.

Notice that the final result of the gfp calculation may not be unique since, in each iteration, there are several possible choices of maximal functional dependency. As one choice has been made, it fixes the dependency functions for state variables that are declared as dependents. Thereafter, the dependency becomes an invariant throughout the computation since the derivation is valid for the entire set of non-dangling states. For the same reason, the gfp calculation may be too conservative. Moreover, the optimality of the gfp calculation is limited because the state variables are initially treated functionally independent of each other. This limitation becomes apparent especially when the dependency to be

SequentialDependencyGfp
input: a state transition system $\mathcal{M} = (I, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{\delta}, \boldsymbol{\lambda})$
output: a dependency triplet $(\boldsymbol{\delta}_\phi, \boldsymbol{\delta}_\varphi, \boldsymbol{\theta})$ for $\boldsymbol{\delta}$
begin
01 $i := 0$; $\boldsymbol{\delta}^{(1)} := \boldsymbol{\delta}$
02 **repeat**
03 **if** $i \geq 2$
04 **for** j from $i-1$ to 1
05 $\boldsymbol{\theta}^{(j)}(\boldsymbol{s}_\varphi^{(i)}) := \boldsymbol{\theta}^{(j)}(\langle \boldsymbol{\theta}^{(j+1)}(\boldsymbol{s}_\varphi^{(i)}); \ldots; \boldsymbol{\theta}^{(i)}(\boldsymbol{s}_\varphi^{(i)}); \boldsymbol{s}_\varphi^{(i)} \rangle)$
06 **if** $i \geq 1$
07 $\boldsymbol{\delta}^{(i+1)}(\boldsymbol{r}, \boldsymbol{s}_\varphi^{(i)}) := \boldsymbol{\delta}_\varphi^{(i)}(\boldsymbol{r}, \langle \boldsymbol{\theta}^{(1)}(\boldsymbol{s}_\varphi^{(i)}); \ldots; \boldsymbol{\theta}^{(i)}(\boldsymbol{s}_\varphi^{(i)}); \boldsymbol{s}_\varphi^{(i)} \rangle)$
08 $i := i + 1$
09 $(\boldsymbol{\delta}_\phi^{(i)}, \boldsymbol{\delta}_\varphi^{(i)}, \boldsymbol{\theta}^{(i)}) :=$ *CombinationalDependencyRestore*$(\boldsymbol{\delta}^{(i)}, \boldsymbol{\delta})$
10 **until** $|\boldsymbol{\delta}^{(i)}| = |\boldsymbol{\delta}_\varphi^{(i)}|$
11 **return** $(\langle \boldsymbol{\delta}_\phi^{(1)}; \ldots; \boldsymbol{\delta}_\phi^{(i-1)} \rangle, \boldsymbol{\delta}_\varphi^{(i-1)}, \langle \boldsymbol{\theta}^{(1)}; \ldots; \boldsymbol{\theta}^{(i-1)} \rangle)$
end

Fig. 2. Algorithm: SequentialDependencyGfp.

discovered is between two state transition systems (e.g., in equivalence checking). To discover more dependency, we need to adopt a least fixed-point strategy and refine the dependency iteratively.

The Least Fixed-Point Calculation. In the lfp calculation, unlike the gfp one, the initial dependency among state variables is exploited maximally based on the set of initial states. The dependency is then strengthened iteratively until a fixed point has been reached. The set of functional independents tend to increase during the iterations, in contrast to the decrease in the gfp calculation.

Consider the computation of initial dependency. For the simplest case, when $|I| = 1$, any state variable s_φ can be selected as the basis. Any other variable is replaced with either $\Im(s_\varphi)$ or $\Im^\dagger(s_\varphi)$, depending on whether its initial value equals that of s_φ or not. For arbitrary I, the initial variable dependency can be derived using functional deduction on the characteristic function of I. (As noted in Remark 1, excluding dangling states from I reveals more dependency.)

For the iterative computation, transition functions are updated in every iteration by eliminating dependent state variables with the latest dependency functions. Combinational dependency is then obtained for the new set of transition functions. Unlike the gfp iterations, the obtained functional dependency in the ith iteration may not be an invariant for the following iterations because the derived dependency may be valid only in the state subspace spanned by $\{\boldsymbol{s}_\varphi^{(i-1)}\}$. As the state subspace changes over the iterations due to different selections of independent state variables, the dependency may need to be rectified. Notice that the set of functional independents may not increase monotonically during the iterations. This non-convergent phenomenon is due to the existence of the don't-care choices of $\boldsymbol{\theta}^{(i)}$ in addition to the imperfect information about the currently reachable state set. Therefore, additional requirements need to be

SequentialDependencyLfp
input: a state transition system $\mathcal{M} = (I, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{\delta}, \boldsymbol{\lambda})$
output: a dependency triplet $(\boldsymbol{\delta_\phi}, \boldsymbol{\delta_\varphi}, \boldsymbol{\theta})$ for $\boldsymbol{\delta}$
begin
01 $i := 0$; $(\boldsymbol{s}_{\boldsymbol{\phi}}^{(0)}, \boldsymbol{s}_{\boldsymbol{\varphi}}^{(0)}, \boldsymbol{\theta}^{(0)}) := \mathit{InitialDependency}(I)$
02 **repeat**
03 $i := i + 1$
04 $\boldsymbol{\delta}^{(i)} := \boldsymbol{\delta}(\boldsymbol{r}, \langle \boldsymbol{\theta}^{(i-1)}(\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i-1)}); \boldsymbol{s}_{\boldsymbol{\varphi}}^{(i-1)} \rangle)$
05 $(\boldsymbol{\delta}_{\boldsymbol{\phi}}^{(i)}, \boldsymbol{\delta}_{\boldsymbol{\varphi}}^{(i)}, \boldsymbol{\theta}^{(i)}) := \mathit{CombinationalDependencyReuse}(\boldsymbol{\delta}^{(i)}, \boldsymbol{\theta}^{(i-1)})$
06 **until** $\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)}$
07 **return** $(\boldsymbol{\delta}_{\boldsymbol{\phi}}^{(i)}, \boldsymbol{\delta}_{\boldsymbol{\varphi}}^{(i)}, \boldsymbol{\theta}^{(i)})$
end

Fig. 3. Algorithm: SequentialDependencyLfp.

imposed to guarantee termination. Here we request that, after a certain number of iterations, the set of independent state variables increase monotonically until $\boldsymbol{\theta}^{(i)}$ can be reused in the next iteration, that is, the fixed point is reached. The algorithm is outlined in Figure 3. To simplify the presentation, it contains only the iterations where $\{\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)}\}$ increases monotonically. Procedure *CombinationalDependencyReuse* is the same as *CombinationalDependency* except that it tries to maximally reuse the dependency functions provided in its second argument.

In theory, the optimality of the lfp calculation lies somewhere between that of the gfp calculation and that of the most general computation with reachability analysis. Since not all dependency in $\mathcal{M}$ can be detected by the lfp procedure due to the imperfect information about the reachable states, the algorithm is incomplete in detecting dependency. To make it complete, reachability analysis should be incorporated. We postpone this integration to the next section and phrase it in the context of verification reduction.

Remark 2. Notice that when $\boldsymbol{\theta}^{(i)}$'s are restricted to consisting of only identity functions and/or complementary identity ones, refinement relation $\sqsubseteq$ among transition functions reduces to an equivalence relation; the lfp calculation of sequential dependency reduces to the detection of equivalent state variables. Hence, detecting signal correspondence [9] is a special case of our formulation.

4 Verification Reduction

Here we focus on the reduction for safety property verification, where reachability analysis is the core computation. The verification problem asks if a state transition system $\mathcal{M} = (I, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{\delta}, \boldsymbol{\lambda})$ satisfies a safety property P, denoted as $\mathcal{M} \models P$, for all of its reachable states.

Suppose that $(\boldsymbol{\delta_\phi}, \boldsymbol{\delta_\varphi}, \boldsymbol{\theta})$ is a dependency triplet of $\boldsymbol{\delta}$; let $\boldsymbol{s_\phi}$ and $\boldsymbol{s_\varphi}$ be the corresponding state variables of $\boldsymbol{\delta_\phi}$ and $\boldsymbol{\delta_\varphi}$, respectively. To represent the reachable state set, either $\boldsymbol{s}$ or $\boldsymbol{s_\varphi}$ can be selected as the basis. Essentially, $R(\boldsymbol{s}) = \mathit{Expand}(R^{\perp}(\boldsymbol{s_\varphi}), (\boldsymbol{s_\phi}, \boldsymbol{s_\varphi}, \boldsymbol{\theta})) = R^{\perp}(\boldsymbol{s_\varphi}) \wedge \bigwedge_i (s_{\phi i} \equiv \theta_i(\boldsymbol{s_\varphi}))$, where R and $R^{\perp}$ are the characteristic functions representing the reachable state sets in

ComputeReachWithDependencyReduction
input: a state transition system $\mathcal{M} = (I, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{\delta}, \boldsymbol{\lambda})$
output: the set R of reachable states of $\mathcal{M}$
begin
01 $i := 0$; $(\boldsymbol{s}_{\boldsymbol{\phi}}^{(0)}, \boldsymbol{s}_{\boldsymbol{\varphi}}^{(0)}, \boldsymbol{\theta}^{(0)}) := \mathit{InitialDependency}(I)$
02 $I^{\perp 0} := I[\{\boldsymbol{s}_{\boldsymbol{\varphi}}^{(0)}\}/\{\boldsymbol{s}\}]$
03 $R^{\perp 0} := I^{\perp 0}$; $F^{\perp 0} := I^{\perp 0}$
04 **repeat**
05 $i := i + 1$
06 $\boldsymbol{\delta}^{(i)} := \boldsymbol{\delta}(\boldsymbol{r}, \langle \boldsymbol{\theta}^{(i-1)}(\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i-1)}); \boldsymbol{s}_{\boldsymbol{\varphi}}^{(i-1)} \rangle)$
07 $(\boldsymbol{\delta}_{\boldsymbol{\phi}}^{(i)}, \boldsymbol{\delta}_{\boldsymbol{\varphi}}^{(i)}, \boldsymbol{\theta}^{(i)}) := \mathit{CombinationalDependencyReach}(\boldsymbol{\delta}^{(i)}, \boldsymbol{\theta}^{(i-1)}, R^{\perp i-1})$
08 $T^{\perp i} := \mathit{Image}(\boldsymbol{\delta}_{\boldsymbol{\varphi}}^{(i)}, F^{\perp i-1})$
09 $\boldsymbol{s}_{\boldsymbol{\nu}} := \boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)} \backslash \boldsymbol{s}_{\boldsymbol{\varphi}}^{(i-1)}$; $\boldsymbol{\theta}_{\boldsymbol{\nu}} := \boldsymbol{s}_{\boldsymbol{\nu}}$'s corresponding functions in $\boldsymbol{\theta}^{(i-1)}$
10 $R^{\perp i-1} := \mathit{Expand}(R^{\perp i-1}, (\boldsymbol{s}_{\boldsymbol{\nu}}, \boldsymbol{s}_{\boldsymbol{\varphi}}^{(i-1)}, \boldsymbol{\theta}_{\boldsymbol{\nu}}))$
11 $R^{\perp i-1} := R^{\perp i-1}[\{\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)}\}/\{\boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)} \cup \boldsymbol{s}_{\boldsymbol{\varphi}}^{(i-1)}\}]$
12 $F^{\perp i} :=$ simplify $T^{\perp i}$ with $R^{\perp i-1}$ as don't care
13 $R^{\perp i} := R^{\perp i-1} \cup T^{\perp i}$
14 **until** $R^{\perp i} = R^{\perp i-1}$
15 **return** $\mathit{Expand}(R^{\perp i}, (\boldsymbol{s}_{\boldsymbol{\phi}}^{(i)}, \boldsymbol{s}_{\boldsymbol{\varphi}}^{(i)}, \boldsymbol{\theta}^{(i)}))$
end

Fig. 4. Algorithm: ComputeReachWithDependencyReduction.

the total space and, respectively, in the reduced space spanned by $\boldsymbol{s}_{\boldsymbol{\varphi}}$. Let $P(\boldsymbol{s})$ denote the states that satisfy P. Checking whether $R(\boldsymbol{s}) \Rightarrow P(\boldsymbol{s})$ is equivalent to checking whether $R^{\perp}(\boldsymbol{s}_{\boldsymbol{\varphi}}) \Rightarrow P^{\perp}(\boldsymbol{s}_{\boldsymbol{\varphi}})$, where $P^{\perp}(\boldsymbol{s}_{\boldsymbol{\varphi}}) = P(\langle \boldsymbol{\theta}(\boldsymbol{s}_{\boldsymbol{\varphi}}); \boldsymbol{s}_{\boldsymbol{\varphi}} \rangle)$. Hence, the verification problem can be carried out solely over the reduced space. As noted in Remark 1, the set I of initial states might require special handling.

For given dependency, reachability analysis can be carried out solely upon the reduced basis. The validity of the given dependency can be tested in every iteration of the reachability analysis as was done in [11]. Below we concentrate on the cases where dependency is not given. We show how the detection of functional dependency can be embedded into and simplify the reachability analysis.

To analyze the reachability of a transition system with unknown dependency, two approaches can be taken. One is to find the sequential dependency with the forementioned gfp and/or lfp calculation, and then perform reachability analysis on the reduced state space based on the obtained dependency. The other is to embed the dependency detection into the reachability analysis as an on-the-fly reduction. Since the former is straightforward, we only detail the latter. Figure 4 sketches the algorithm. Procedure *CombinationalDependencyReach* is similar to *CombinationalDependencyReuse* with two exceptions: First, the derived dependency is with respect to the reached state set provided in the third argument. Second, the set of independent state variables needs not increase monotonically since the termination condition has been taken care of by the reached state sets. In each iteration of the state traversal, the previously reached state set R is adjusted (by the expansion and projection operations) to a new basis according to the derived dependency triplet.

5 Experimental Results

The forementioned algorithms have been implemented in the VIS [4] environment. Experiments were conducted on a Sun machine with a 900-MHz CPU and 2-Gb memory. Three sets of experiments have results shown in Tables 1, 2, and 3, respectively. Table 1 demonstrates the relative power of exploiting dependency by the detection of signal correspondence, the gfp, and lfp calculations of sequential dependency. Table 2 compares their applicabilities in the equivalence checking problem. Finally, Table 3 shows how reachability analysis can benefit from our computation of functional dependency. In the experiments, all the approaches under comparison use the same BDD ordering. In addition, no reordering is invoked.

Table 1. Comparisons of Capabilities of Discovering Dependency.

		Signal Corr. [9]				Seq. Dep. Gfp				Seq. Dep. Lfp			
Circuit	State Var.	indp.	iter.	mem. (Mb)	time (sec)	indp.	iter.	mem. (Mb)	time (sec)	indp.	iter.	mem. (Mb)	time (sec)
s298-rt	34 (14)	31	5	10	0.3	**23**	2	23	1.6	24	10	41	6.2
s499-rt	41 (22)	41	21	13	1.6	**29**	1	23	11.6	**29**	22	23	8.2
s510-rt	34 (6)	32	4	13	0.4	**21**	2	51	17.5	23	6	58	81.1
s526n-rt	64 (21)	55	4	13	1.0	**37**	2	60	104.2	40	14	58	26.8
s635-rt	51 (32)	50	16	13	0.6	**34**	2	13	2.8	**34**	33	21	7.4
s838.1-rt	73 (32)	48	20	13	1.5	**33**	1	22	3.7	**33**	46	21	18.3
s991-rt	42 (19)	24	2	13	0.5	21	2	21	1.4	**20**	2	21	1.4
mult16a-rt	106 (16)	66	6	13	0.9	75	2	13	1.0	**61**	8	13	4.6
tbk-rt	49 (5)	49	2	49	6.8	**13**	4	62	264.1	21	3	59	48.4
s3271	116	**114**	6	29	2.1	116	0	29	3.0	**114**	6	45	12.6
s4863	104	81	3	47	4.7	81	1	69	178.7	**75**	3	47	14.5
s5378	179	163	12	37	6.5	155	2	51	15.9	**154**	14	51	43.1
s9234.1	211	188	18	99	79.5	189	2	97	250.2	**184**	38	99	967.6
s13207	669	303	16	138	95.6	460	5	111	384.6	**263**	37	100	836.0
s15850	597	431	24	142	221.7	569	3	134	1487.1	**315**	32	142	1441.0
s35932	1728	**1472**	31	281	599.8	1728	0	146	34091.5	–	–	–	$> 10^5$
s38584	1452	869	17	303	525.5	1440	1	155	4103.3	**849**	25	303	22001.1
8085	193	91	15	65	28.9	193	0	70	42.4	**79**	17	63	64.3

Compared in Table 1 are three approaches: the computation of signal correspondence [9], the gfp, and lfp calculations of sequential dependency. The first two columns list the benchmark circuits and their sizes in state variables. The original sizes of retimed circuits (for timing optimization) are listed in the following parentheses. For each compared approach, four columns in order list the sizes of the computed independent state variables, the required numbers of iterations, memory usage, and CPU time. Among these three approaches, the minimum sizes of independent variables are highlighted in bold. It is evident from Table 1 that the lfp calculation of sequential dependency subsumes the detection of signal correspondence in both generality and optimality. On the other hand, the powers of the lfp and gfp calculations are incomparable in practice. They have different directions of approximating reachable state sets. For the gfp calculation, the unreachable state set is gradually pruned each time dependency functions are substituted backward. For the lfp one, the reachable state set grows with the iterative computation. It turns out that the gfp computation is very effective

in exploiting dependency for retimed circuits. For instance, in circuit `tbk-rt`, 13 variables are identified as independents by the gfp calculation, compared to 24 by the lfp one. In general, the gfp computation uses much fewer iterations than the other two approaches. In contrast, the lfp calculation outperforms the other two approaches in circuits not retimed. The table also reveals that all the approaches do not suffer from memory explosion. Rather, the time consumption may be a concern in the gfp and lfp calculations of sequential dependency. This is understandable because testing the refinement relation is more general and complicated than testing the equivalence relation used in the detection of signal correspondence. Fortunately, the tradeoff between quality and time can be easily controlled, for example, by imposing k-substitutability, which uses up to k functions to substitute a dependent function. With our formulation, dependencies that were underivable before, due to the limitation of reachability analysis on large transition systems, can now be computed efficiently.

Table 2. Comparisons of Capabilities of Checking Equivalence.

		Signal Corr. [9]				Seq. Dep. Gfp				Seq. Dep. Lfp			
Circuit	State Var.	indp.	iter.	mem. (Mb)	time (sec)	indp.	iter.	mem. (Mb)	time (sec)	indp.	iter.	mem. (Mb)	time (sec)
`s298`	14+34	39	5	10	0.5	37	2	21	1.5	**30**	13	31	4.4
`s499`	22+41	63	21	14	3.1	43	2	38	7.3	**42**	22	45	23.6
`s510`	6+34	38	4	13	0.6	**27**	2	50	25.9	29	5	36	39.8
`s526n`	21+64	69	8	13	2.4	58	2	59	121.9	**50**	12	58	31.8
`s635`	32+51	66	31	13	7.8	66	1	21	1.4	**51**	33	25	9.1
`s838.1`	32+73	78	31	25	16.8	65	2	48	4.2	**59**	47	37	22.5
`s991`	19+42	42	2	22	1.5	40	2	38	2.5	**39**	3	41	5.4
`mult16a`	16+106	82	6	14	4.6	91	2	14	1.7	**77**	8	26	5.1
`tbk`	5+49	54	2	44	5.5	**17**	4	61	175.6	25	3	59	86.4

With similar layout to Table 1, Table 2 compares the applicabilities of these three approaches to the equivalence checking problem. Here a product machine is built upon a circuit and its retimed version. As noted earlier, the gfp calculation itself cannot prove the equivalence between two systems. It, essentially, computes the dependency inside each individual system, but not the interdependency between them. On the other hand, the detection of signal correspondence can rarely prove equivalence unless the two systems under comparison are almost functionally identical. In contrast, the lfp calculation of sequential dependency can easily prove the equivalence between two systems where one is forwardly retimed from the other, and vice versa. Arbitrary retiming, however, may cause a failure, although in principle there always exists a lfp calculation that can conclude the equivalence. In Table 2, since the retiming operations on the retimed circuits involve both forward and backward moves, none of the approaches can directly conclude the equivalences. However, as can be seen, the lfp calculation can compactly condense the product machines.

Although detecting dependency can reduce state space, it is not clear if the BDD sizes for the dependency functions and the rewritten transition functions are small enough to benefit reachability analysis. In Table 3, we justify that it indeed can improve the analysis. Some hard instances for state traversal are

Table 3. Comparisons of Capabilities of Analyzing Reachability.

		R.A. w/o Dep. Reduction				R.A. w Dep. Reduction			
Circuit	Iter.	peak (bdd nodes)	reached (bdd nodes)	mem. (Mb)	time (sec)	peak (bdd nodes)	reached (bdd nodes)	mem. (Mb)	time (sec)
s3271	4	28819301	16158242	620	2784.1	18843837	10746053	415	1082.6
s4863	2	18527781	248885	365	404.8	549006	8772	67	13.1
s5378	2	–	–	> 2000	–	1151439	113522	70	21.5
s15850	15	29842889	9961945	653	21337.4	17667076	6356714	463	8175.0
8085	50	16663749	1701604	390	24280.2	7830602	1338322	212	4640.1

studied. We compare reachability analyses without and with on-the-fly reduction using functional dependency. In the comparison, both analyses have the same implementation except switching off and on the reduction option. The second column of Table 3 shows the steps for (partial) state traversal. For each reachability analysis, four columns in order shows the peak number of live BDD nodes, the size of the BDD representing the final reached state set, memory usage, and CPU time. It is apparent that, with the help of functional dependency, the reachability analysis yields substantial savings in both memory and time consumptions, compared to the analysis without reduction.

6 Comparisons with Related Work

Among previous studies [11, 8] on exploiting functional dependency, the one closest to ours is [8] while functional dependency in [11] is assumed to be given. The method proposed in [8] is similar to our reachability analysis with on-the-fly reduction. However, several differences need to be addressed. First, previous dependency was drawn entirely from the currently reached state set (using functional deduction) rather than from the transition functions. Thus, in each iteration of their reachability analysis, image computation need to be done before the detection of new functional dependency. The image computation rarely benefits from functional dependency. In contrast, our approach is more effective because the dependency is discovered before the image computation, which is performed on the reduced basis. Second, as previous dependency was obtained from a currently reached state set, not from transition functions, it is not as robust as ours to remain valid through the following iterations. Third, the prior method cannot compute functional dependency without reachability analysis while our formulation can be used as a stand-alone technique. Also, we identify a new initial set of non-dangling states. It uncovers more dependency to be exploited.

For related work specific to sequential equivalence checking, we mention [16, 1, 18, 9]. Among them, the work of [9] is the most relevant to ours; it is a special case of our lfp calculation as noted in Remark 2. While these prior efforts focus on equivalence checking, ours is more general for safety property checking.

7 Conclusions and Future Work

We formulate the dependency among a collection of functions based on a refinement relation. When applied to state transition systems, it allows the detection of functional dependency without knowing reached state sets. With an integration

into a reachability analysis, it can be used as a complete verification procedure with the power of on-the-fly reduction. Our formulation unifies the work of [9] and [8] in the verification framework. In application to the equivalence checking problem, our method bridges the complexity gap between combinational and sequential equivalence checking. Preliminary experiments show promising results in detecting dependency and verification reduction.

As a future research direction, our results might be reformulated in a SAT-solving framework. A path similar to that of [3], where van Eijk's algorithm was adjusted, could be taken to prove safety properties by strengthened induction. Because our approach may impose more invariants than just signal correspondence, we believe that SAT-based verification can benefit from our results.

References

1. P. Ashar, A. Gupta, and S. Malik. Using complete-1-distinguishability for FSM equivalence checking. In *Proc. ICCAD*, pages 346–353, 1996.
2. C. Berthet, O. Coudert, and J.-C. Madre. New ideas on symbolic manipulations of finite state machines. In *Proc. ICCD*, pages 224–227, 1990.
3. P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Proc. FMCAD*, pages 372–389, 2000.
4. R. K. Brayton, et al. VIS: a system for verification and synthesis. In *Proc. CAV*, pages 428–432, 1996.
5. F. M. Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Dover Publications, 2003.
6. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, pages 677–691, August 1986.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
8. C. A. J. van Eijk and J. A. G. Jess. Exploiting functional dependencies in finite state machine verification. In *Proc. European Design & Test Conf.*, pages 9–14, 1996.
9. C. A. J. van Eijk. Sequential equivalence checking based on structural similarities. *IEEE Trans. Computer-Aided Design*, pages 814–819, July 2000.
10. T. Filkorn. Symbolische methoden für die verifikation endlicher zustandssysteme. Ph.D. thesis. Institut für Informatik der Technischen Universität München, 1992.
11. A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In *Proc. DAC*, pages 266–271, 1993.
12. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
13. B. Lin and A. R. Newton. Exact redundant state registers removal based on binary decision diagrams. In *Proc. Int'l Conf. VLSI*, pages 277–286, 1991.
14. C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. *J. VLSI Computer Syst.*, vol. 1, no. 1, pp. 41–67, 1983.
15. E. Marczewski. Independence in algebras of sets and Boolean algebra. *Fundamenta Mathematicae*, vol. 48, pages 135–145, 1960.
16. S. Quer, G. Cabodi, P. Camurati, L. Lavagno, and R. Brayton. Verification of similar FSMs by mixing incremental re-encoding, reachability analysis, and combinational check. *Formal Methods in System Design*, vol. 17, pages 107–134, 2000.
17. E. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proc. ICCAD*, pages 428–435, 1996.
18. D. Stoffel and W. Kunz. Record & play: A structural fixed point iteration for sequential circuit verification. In *Proc. ICCAD*, pages 394–399, 1997.