An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis

Chandan Karfa, Dipankar Sarkar, Chittaranjan Mandal, and Pramod Kumar

***Abstract*—A formal method for checking equivalence between a given behavioral specification prior to scheduling and the one produced by the scheduler is described. Finite state machine with data path (FSMD) models have been used to represent both the behaviors. The method consists of introducing cutpoints in one FSMD, visualizing its computations as concatenation of paths from cutpoints to cutpoints, and identifying equivalent finite path segments in the other FSMD; the process is then repeated with the FSMDs interchanged. Unlike many other reported techniques, this method is strong enough to work when path segments in the original behavior are merged, a common feature of scheduling. It is also capable of verifying several arithmetic transformations and many code-motion techniques employed during scheduling. Correctness and complexity of the method have been dealt with. Experimental results for several high-level synthesis benchmarks demonstrate the effectiveness of the method.**

***Index Terms*—Equivalence checking, finite state machine with data path (FSMD) models, formal verification, high-level synthesis (HLS), scheduling.**

I. Introduction

HIGH-LEVEL synthesis (HLS) is the process of generating the register-transfer-level (RTL) design from the behavioral description. The synthesis process consists of several interdependent subtasks such as specification, compilation, scheduling, allocation and binding [14]. These tasks are handled in various phases. Numerous synthesis tools exist that handle various subsets of tasks in different ways. The complexity of present-day VLSI systems is very high. The specification is given at a high level of abstraction compared to that of the output. In addition, different types of optimizations are performed in each phase of HLS. Phasewise verification techniques, with flexibility to apply appropriate verification methods in various phases are, therefore, desirable for HLS verification.

The input behavior to the scheduler is modified in several ways in order to use a minimum number of time steps to schedule the operations. For example, the control structure of the input behavior may be modified by the path-based scheduler [7], [38] as it tries to merge some consecutive path segments of the input behavior. Also, incorporation of several code-motion techniques [19] in the scheduling process leads to moving operations across the basic-block (BB) boundaries. Consequently, the results of scheduling do not have a one-to-one correspondence with the input. This aspect makes scheduler verification the most challenging of the HLS verification tasks. The objective of this paper is to check that the design descriptions before and after scheduling are computationally equivalent.

Methods to verify the HLS results against the original behavioral description are still evolving [25]. Synthesis verification can be classified into three categories [27]: *presynthesis verification* where software verification methods are used; *formal synthesis verification* where the synthesis result is formally derived using some logical calculus; and *postsynthesis verification* where the synthesized results are verified against the input behavioral descriptions. Chapman *et al.* [8] proposed a presynthesis verification technique and employed it in the BEDROC HLS tool. Software verification, however, poses much difficulty because of the large size of the synthesis tools. Therefore, it is not possible to apply the presynthesis verification for all the phases of the HLS flow; these techniques are better suited for compilation. The works reported in [4] and [33] for HLS verification are based on formal synthesis. However, the weak relationship between the formal transformations and their underlying hardware concepts forces these systems to be manually driven by the designer who has to understand the mathematical formalism. The last category of postsynthesis verification has the advantage that the correspondence between outputs of various design steps in the synthesis flow is established and that it is independent of the synthesis procedure. In fact, the postsynthesis verification is the most frequently used approach today [27]. Our technique belongs to this category.

The end-to-end verification of HLS is proposed in [13], [32], and [37]. These techniques, however, are not only tough but also inadequate in locating exact scheduling errors. Simulation-based verification of HLS is proposed in [3] and [9]. As the complexity of digital systems increases significantly, these techniques are becoming impractical. The postsynthesis methods like in [2] and [5] are not well suited for scheduling verification; rather, they are applicable to the allocation and binding phase of HLS.

A theorem-proving approach for verifying the *As Soon As Possible (ASAP) scheduling* algorithm is reported in [1]. Verification of force-directed list scheduling algorithm for resource-constrained scheduling in HLS systems is presented in [34]. In [36], scheduling results are verified using a sequence of some correctness-preserving RTL transformations called the register transfer splits. The work reported in [32] assumes that the input behavior has already been transformed and that the control

Manuscript received April 5, 2007; revised July 21, 2007. This work was supported in part by MHRD (Grant F.26-14/2003.TS.V) and in part by MCIT (SMDP-II projet), Government of India. This paper was recommended by Associate Editor A. Kuehlmann.

The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721 302, India (e-mail: ckarfa@iitkgp.ac.in; ckarfa@yahoo.co.in; ds@cse.iitkgp.ernet.in; chitta@iitkgp.ac.in; pkumar@cse.iitkgp.ernet.in).

Digital Object Identifier 10.1109/TCAD.2007.913390

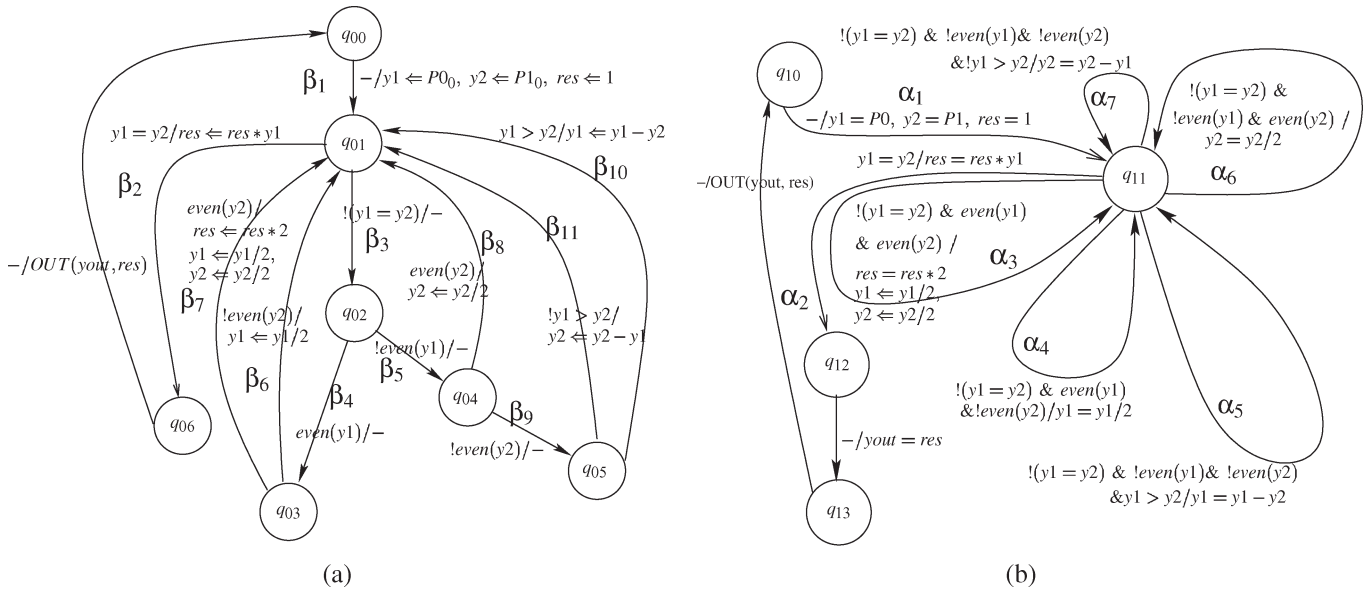


Fig. 1. (a) M_0: The FSMD of GCD before scheduling. (b) M_1: The FSMD of GCD after scheduling.

structure of the behavior is not modified during scheduling. In [10], the prescheduled and postscheduled behaviors are represented in the language of labeled segments. The method described in this paper transforms the original description into one, which is bisimilar with the scheduled description. In [25], the input and the output of the scheduler are encoded as finite state machines with data paths (FSMDs) [14]. In this approach, break points are introduced in both the FSMDs followed by the construction of the respective path sets. Each path of one set is then shown to be equivalent to some path of the other set. The approach presented in this paper requires that the path structure of the input FSMD is not disturbed by the scheduling algorithm, i.e., the respective path sets obtained from the break points remain bijective. All these verification approaches, however, are well suited for BB-based scheduling [22], [29] where the path structure of the input behavior does not modify due to scheduling or operations are not moved across the BB boundaries.

The path-based scheduling algorithms [7], [38], however, may modify the control structure of the input behavior as they try to merge the consecutive path segments. Let us consider the example given in Fig. 1. An example behavioral specification for finding greatest common divisor (GCD) is given in Fig. 1(a) and its scheduled behavior as obtained by a path-based scheduler is given in Fig. 1(b). It may be noted that the control structure of the input behavior is modified by the (path-based) scheduler. To the best of our knowledge, the algorithms proposed in the literature, some of which are cited previously, can successfully verify the BB-based scheduling but apparently fail when the path structure of the scheduled behavior differs from that of the input due to path-based transformation or code motions across the BB boundaries. In this paper, we propose a scheduling verification method which formally establishes equivalence between the FSMDs before and after scheduling. A preliminary version of this paper appears in [23].

The main contributions of this paper are as follows. The method has been devised to be strong enough to work even when the basic control structure is changed by the scheduler. This method is also capable of verifying several code-motion techniques applied during scheduling. Incorporation of normalization techniques during equivalence checking of paths in our method supports different arithmetic transformations. Care has been taken so that the method works when some storage variables of the scheduled behavior are removed or some new variables are introduced in the scheduled behavior. The extendability of the method for verification of behaviors involving pipelined and multicycle operations is also discussed. A formal treatment of soundness and complexity of the reported method has been included. Given the undecidability results of the equivalence problem of flowchart schemas [21], [31], completeness, however, is unattainable. The ability to handle path-based scheduling and several code transformations applied during scheduling underlines the usability of this method in the perspective of the inherent incompleteness of the problem.

The rest of this paper is organized as follows. In Section II, the FSMD model, the notions of computations on FSMDs and the equivalence of FSMDs are defined. The verification method is described in Section III. The working of the algorithm is illustrated with an example in Section IV. The correctness and the complexity of the proposed method are treated in Section V. Several code-transformation techniques which are applied during scheduling, along with whether and how they can be handled by our algorithm, are discussed in Section VI. Extendability of the method for multicycle and pipelined operations is discussed in brief in Section VII. Some experimental results have been given in Section VIII. This paper is concluded in Section IX.

II. FSMDs and Their Equivalence

A. Finite State Machines With Data Paths (FSMDs)

An FSMD is a universal specification model [14] that can represent all hardware designs. The model is used in this paper

with the addition of a reset state for encoding the designs to be verified. This reset state is also called the start state of the FSMD. The FSMD is defined as an ordered tuple $\langle Q, q_0, I, V, O, f, h\rangle$, where

1) $Q = \{q_0, q_1, q_2, \ldots, q_n\}$ is the finite set of control states,
2) $q_0 \in Q$ is the reset state,
3) I is the set of primary input signals,
4) V is the set of storage variables, and Σ is the set of all data storage states or simply data states,
5) O is the set of primary output signals,
6) $f : Q \times 2^S \rightarrow Q$ is the state transition function,
7) $h : Q \times 2^S \rightarrow U$ is the update function of the output and the storage variables, where S and U are defined as follows.
   a) $S = \{L \cup E\}$ is the set of status expressions, where L is the set of Boolean literals of the form b or $\neg b$, $b \in B \subseteq V$ is a Boolean variable and E is the set of arithmetic predicates over $I \cup (V - B)$. Any arithmetic predicate is of the form $eR0$, where e is an arithmetic expression and $R \in \{==, \neq, >, \geq, <, \leq\}$.
   b) U is a set of storage or output assignments of the form $\{x \Leftarrow e | x \in O \cup V$, and e is an arithmetic predicate or expression over $I \cup (V - B)\}$ that represents a set of storage or output assignments.

The implicit connective among the members of the set of status expressions, which occurs as the second argument of the function f (or h), is conjunction. Parallel edges between two states capture the disjunction of status expressions. Thus, the next (control and data) state and the output depend not only on the present state and the input signals but also on the conjunction of the status expressions that indicate whether a predicate holds on the data state of the storage and the input variables. The state transition function and the update function are such that the FSMD model remains deterministic. Thus, for any state q, if $f(q, S_1)$ and $f(q, S_2)$ are different, then the sets S_1 and S_2 of status expressions are disjointed. The same property holds for the update function h. It may be noted that we have not introduced any final state in the FSMD model as we assume that a system works in an infinite outer loop.

Example 1: The FSMD model M_0 for the behavioral specification of GCD example is depicted in Fig. 1(a). Specifically,

1) $M_0 = \langle Q, q_0, I, V, O, f, h\rangle$.
2) $Q = \{q_{00}, q_{01}, q_{02}, q_{03}, q_{04}, q_{05}, q_{06}\}$, $q_0 = q_{00}$, $V = \{\text{res}, y1, y2\}$, $I = \{P0, P1\}$, and $O = \{\text{yout}\}$.
3) $U = \{y1 \Leftarrow P0,\ \ y2 \Leftarrow P1,\ \ \text{res} \Leftarrow 1\ \ \text{res} \Leftarrow \text{res} * 2, = y1 \Leftarrow y1/2,\ \ y2 \Leftarrow y2/2,\ \ y1 \Leftarrow y1 - y2,\ \ y2 \Leftarrow y2 - y1,\ \text{res} \Leftarrow \text{res} * y1\}$.
4) $S = \{\text{even}(y1),\ \text{even}(y2),\ y1 == y2,\ y1 > y2\}$, where even$(y)$ is the abbreviation of "$y \bmod 2 = 0$."
5) f and h are as defined in the transition graph shown in Fig. 1(a).
6) Some typical values of f and h are as follows.
   a) $f(q_{00}, \{\text{true}\}) = q_{01}$.
   b) $f(q_{05}, \{y1 > y2\}) = q_{01}$.
   c) $h(q_{05}, \{y1 > y2\}) = \{y1 \Leftarrow y1 - y2\}$.
   d) $h(q_{04}, \{\text{even}(y2)\}) = \{y2 \Leftarrow y2/2\}$. □

B. Paths and Transformations Along a Path

A (finite) path α from q_i to q_j, where $q_i,\ q_j \in Q$, is a finite transition sequence of states of the form $\langle q_i = q_1 \overrightarrow{c1}\, q_2 \overrightarrow{c_2}\, \cdots \overrightarrow{c_{n-1}}\, q_n = q_j\rangle$ such that $\forall l, 1 \leq l \leq n-1, \exists c_l \in 2^S$ such that $f(q_l, c_l) = q_{l+1}$, and $q_k,\ 1 \leq k \leq n-1$, are all distinct. The state q_n may be identical to any $q_k,\ 1 \leq k \leq n-1$. The condition of execution R_α of the path $\alpha = \langle q_1 \overrightarrow{c_1}\, q_2 \overrightarrow{c_2}\, q_3 \cdots \overrightarrow{c_{n-1}}\, q_n\rangle$ is a logical expression over $I \cup V$ such that R_α is satisfied by the (initial) data state at q_1 if the path α is traversed. Thus, R_α is the weakest precondition of the path α [15]. Often, for brevity, the aforementioned path α is represented as $[q_1 \Rightarrow q_n]$.

We assume that inputs and outputs occur through named ports. The ith input from port P is a value symbolically represented as P_i. Thus, if some variable v stores an input from port P (for the ith time along a path), it is equivalent to the assignment $v \Leftarrow P_i$. In essence, P_i's comprise the input variable set I and each input variable in I is read only once in a computation (path).

The *simple data transformation* s_α of a path α over V is an ordered tuple $\langle e_j\rangle$ of algebraic expressions over $I \cup V$ such that the expression e_j represents the value of the variable v_j after execution of the path in terms of the initial data state (i.e., the values of the variables at the initial control state) of the path.

Taking into account the outputs that may occur in a path, the *data transformation* r_α of a path α over V is the tuple $\langle s_\alpha, O_\alpha\rangle$, where the output list $O_\alpha = [\text{OUT}(P_{i_1}, e_1), \text{OUT}(P_{i_2}, e_2), \ldots]$. More specifically, for every expression e output to port P along the path α, there is a member OUT(P, e) in the list appearing in the order in which the outputs occur in α.

C. Computation of R_α and r_α

Computation of the condition of execution R_α can be obtained by backward substitution or by forward substitution. The former is more easily perceivable and is based on the following rule: If a predicate $c(y)$ is true after the assignment $y \leftarrow g(y)$, then the predicate $c(g(y))$ must have been true before the assignment [31]. The transformation s_α is found indirectly using the same principle. The forward-substitution method of finding R_α is based on symbolic execution. The ordered pairs at various points in Fig. 2 represent the values of (R_α, s_α) at that point.

D. Characterization of Paths

The characteristic formula $\tau_\alpha(\overline{v}, \overline{v}_f, O)$ of the path α is $R_\alpha(\overline{v}) \wedge (\overline{v}_f = s_\alpha(\overline{v})) \wedge (O = O_\alpha(\overline{v}))$, where s_α is the data transformation, O_α is the output list in the path α, $\overline{v}$ represents a vector of variables of $I \cup V$ and $\overline{v}_f$ represents a vector of variables of V. The formula captures the following: If the condition of execution R_α of the path α is satisfied by the (initial) vector $\overline{v}$ at the beginning of the path, then the path is executed, and after execution, the final vector $\overline{v}_f$ of variable values becomes $s_\alpha(\overline{v})$, and the output $O_\alpha(\overline{v})$ is produced.

Let $\tau_\alpha(\overline{v}, \overline{v}_f, O) : R_\alpha(\overline{v}) \wedge (\overline{v}_f = s_\alpha(\overline{v})) \wedge (O = O_\alpha(\overline{v}))$ be the characteristic formula of the path α and $\tau_\beta(\overline{v}, \overline{v}_f, O)$:

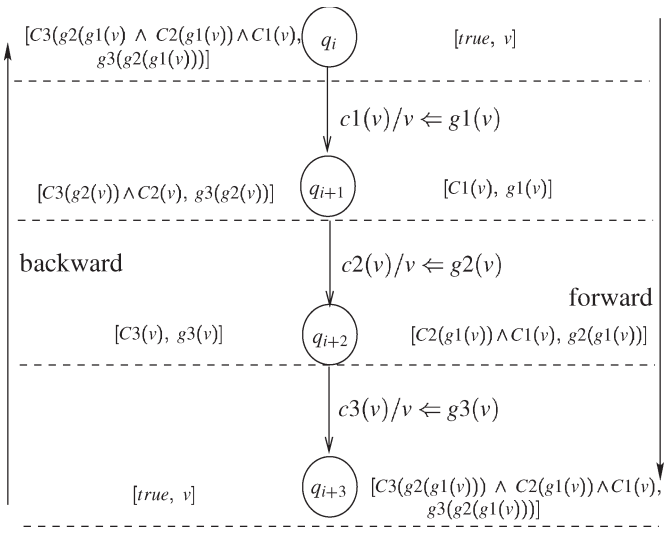


Fig. 2. Typical path, its condition of execution, and its simple data transformation.

$R_\beta(\overline{v}) \wedge (\overline{v}_f = s_\beta(\overline{v})) \wedge (O = O_\beta(\overline{v}))$ be the characteristic formula of the path β. The characteristic formula for the concatenated path $\alpha\beta$ is $\tau_{\alpha\beta}(\overline{v}, \overline{v}_f, O) = \exists \overline{v}_\alpha \exists O_1 \exists O_2 (\tau_\alpha(\overline{v}, \overline{v}_\alpha, O_1) \wedge \tau_\beta(\overline{v}_\alpha, \overline{v}_f, O_2)) = R_\alpha(\overline{v}) \wedge R_\beta(s_\alpha(\overline{v})) \wedge (\overline{v}_f = s_\beta(s_\alpha(\overline{v}))) \wedge (O = O_\alpha(\overline{v}) O_\beta(s_\alpha(\overline{v})))$. O is the concatenated output list of $O_\alpha(\overline{v})$ and $O_\beta(s_\alpha(\overline{v}))$. It is necessary to increment the input indices on each port in the formulas for β to start after the last index of the corresponding port in α; the detail is omitted here for notational clarity.

E. Computations and Path Covers of an FSMD

A computation of an FSMD is a finite walk from the reset state q_0 back to itself without having any intermediary occurrence of q_0. Such a computational semantics of an FSMD is based on the assumption that a revisit of the reset state means the beginning of a new computation and that each computation terminates. A computation μ of an FSMD M may be characterized as $\tau_\mu(\overline{v}_i, \overline{v}_f, O) : R_\mu(\overline{v}_i) \wedge (\overline{v}_f = s_\mu(\overline{v}_i)) \wedge (O = O_\mu(\overline{v}_i))$, where $\overline{v}_i$ is the vector of the initial input with which the computation is started, R_μ is a satisfiable condition over the domain of I, s_μ is a function over this domain to the codomain of values over V and O_μ is the concatenation of the output lists resulting from output operations along μ. The ordered pair $\langle s_\mu, O_\mu \rangle$ is denoted as r_μ.

Definition 1: Two computations μ_1 and μ_2 having the characteristic formula τ_{μ_1} and τ_{μ_2}, respectively, are said to be equivalent, denoted as $\mu_1 \simeq \mu_2$, if $R_{\mu_1} = R_{\mu_2}$ and $r_{\mu_1} = r_{\mu_2}$.

The computational equivalence of two paths p_1 and p_2 can be defined in a similar manner and is denoted as $p_1 \simeq p_2$. Equivalence checking of paths, therefore, consists in establishing the computational equivalence of the respective condition of execution and the respective data transformation.

Any computation μ of an FSMD M can be looked upon as a computation along some concatenated path $[\alpha_1 \alpha_2 \alpha_3, \ldots, \alpha_k]$ of M such that the path α_1 emanates from and the path α_k terminates in the reset state q_0 of M for $1 \leq i \leq k$, α_i terminates in the initial state of the path α_{i+1}, and α_i's may not all be distinct. The characteristic formula τ_μ of μ can accordingly be defined in terms of characteristic formula of the concatenated paths corresponding to μ. Hence, we have the following definition.

Definition 2—Path Cover of an FSMD: A finite set of paths $P = \{p_0, p_1, p_2, \ldots, p_k\}$ is said to be a path cover of an FSMD M if any computation μ of M can be looked upon as a concatenation of paths from P.

F. Equivalence of FSMDs

Let the behavior given as input to the scheduler be represented by the FSMD $M_0 = \langle Q_0, q_{00}, I, V_0, O, f_0, h_0 \rangle$ and the scheduled behavior be represented by the FSMD $M_1 = \langle Q_1, q_{10}, I, V_1, O, f_1, h_1 \rangle$. Our main goal is to verify whether M_0 behaves exactly as M_1. This means that for all possible input sequences, M_0 and M_1 produce the same sequences of output values and eventually, when the respective reset states are revisited, they are visited with the same storage element values. In other words, for every computation from the reset state back to itself of one FSMD, there exists an equivalent computation from the reset state back to itself in the other FSMD and vice versa.

Definition 3: An FSMD M_0 is said to be contained in an FSMD M_1, symbolically $M_0 \sqsubseteq M_1$, if, for any computation μ_0 of M_0, there exists a computation μ_1 of M_1 such that $\mu_0 \simeq \mu_1$.

Definition 4: Two FSMDs M_0 and M_1 are said to be computationally equivalent if $M_0 \sqsubseteq M_1$ and $M_1 \sqsubseteq M_0$.

Theorem 1: An FSMD M_0 is contained in another FSMD $M_1 (M_0 \sqsubseteq M_1)$ if there exists a finite cover $P_0 = \{p_{00}, p_{01}, \ldots, p_{0l}\}$ of M_0 for which there exists a set $P_1 = \{p_{10}, p_{11}, \ldots, p_{1l}\}$ of paths of M_1 such that $p_{0i} \simeq p_{1i}$, $0 \leq i \leq l$.

Proof: $M_0 \sqsubseteq M_1$ if, for any computation μ_0 of M_0, there exists a computation μ_1 of M_1 such that μ_0 and μ_1 are computationally equivalent [by Definition 3].

Now, let there exist a finite cover $P_0 = \{p_{00}, p_{01}, \ldots, p_{0l}\}$ of M_0. Corresponding to P_0, let a set $P_1 = \{p_{10}, p_{11}, \ldots, p_{1l}\}$ of paths of M_1 exist such that $p_{0i} \simeq p_{1i}$, $0 \leq i \leq l$.

Since P_0 covers M_0, any computation μ_0 of M_0 can be looked upon as a concatenated path $[p_{0i_1} p_{0i_2} \cdots p_{0i_n}]$ from P_0 starting from the reset state q_{00} and ending again at this reset state of M_0. From the above hypothesis, it follows that there exists a sequence Π_1 of paths $[p_{1j_1} p_{1j_2} \ldots p_{1j_n}]$ of P_1, where $p_{0i_k} \simeq p_{1j_k}$, $1 \leq k \leq n$. Therefore, in order that Π_1 represents a computation of M_1, it is required to prove that Π_1 is a concatenated path of M_1 from its reset state q_{10} back to itself. The following definition is in order.

Definition 5—Corresponding States: Let $M_0 = \langle Q_0, q_{00}, I, V_0, O, f_0, h_0 \rangle$ and $M_1 = \langle Q_1, q_{10}, I, V_1, O, f_1, h_1 \rangle$ be the two FSMDs having identical input and output sets, I and O, respectively, and $q_{0i}, q_{0k} \in Q_0$ and $q_{1j}, q_{1l} \in Q_1$.

1) The respective reset states q_{00} and q_{10} are corresponding states.
2) If $q_{0i} \in Q_0$ and $q_{1j} \in Q_1$ are corresponding states and there exist $q_{0k} \in Q_0$ and $q_{1l} \in Q_1$ such that, for some path α from q_{0i} to q_{0k} in M_0, there exists a path β from q_{1j} to q_{1l} in M_1 such that $\alpha \simeq \beta$, then q_{0k} and q_{1l} are corresponding states.

Now, let p_{0i_1}: $[q_{00} \Rightarrow q_{0f_1}]$. Since $p_{1j_1} \simeq p_{0i_1}$, from the above definition of corresponding states, p_{1j_1} must be of the form $[q_{10} \Rightarrow q_{1f_1}]$, where $\langle q_{00}, q_{10}\rangle$ and $\langle q_{0f_1}, q_{1f_1}\rangle$ are corresponding states. Thus, by repetitive application of the above argument, it follows that if p_{0i_1}: $[q_{00} \Rightarrow q_{0f_1}]$, p_{0i_2}: $[q_{0f_1} \Rightarrow q_{0f_2}], \ldots, p_{0i_n}$: $[q_{0f_{n-1}} \Rightarrow q_{0f_n} = q_{00}]$, then p_{1i_1}: $[q_{10} \Rightarrow q_{1f_1}]$, p_{1i_2}: $[q_{1f_1} \Rightarrow q_{1f_2}], \ldots, p_{1i_n}$: $[q_{1f_{n-1}} \Rightarrow q_{1f_n} = q_{10}]$, where $\langle q_{0f_m}, q_{1f_m}\rangle$, $1 \leq m \leq n$, are pairs of corresponding states. Hence, Π_1 is a concatenated path representing a computation μ_1 of M_1, where $\mu_1 \simeq \mu_0$. ■

G. Equivalence of Paths

Theorem 1 reduces the equivalence problem of two FSMDs into the problem of determining whether a path of one FSMD is equivalent to some path of the other FSMD. A computation μ_0 of FSMD M_0 can be compared with a computation μ_1 of M_1 as long as both M_0 and M_1 have the same input variable set I because R_{μ_0}, R_{μ_1}, r_{μ_0}, and r_{μ_1} are all defined over I. In contrast, a comparison of a path p_0 of M_0 with a path p_1 of M_1, however, is not so straightforward because, in general, M_0, M_1 may involve different storage variable sets V_0 and V_1, respectively. This may happen due to various code-motion techniques applied during scheduling. Since paths can start from any cutpoints of the FSMD, their conditions of execution and the data transformations will be in terms of the inputs and the storage variables. Thus, the path p_0 may involve variables from the set V_0 and the path p_1 may involve variables from $V_1 \neq V_0$. To handle such situations, the condition R_α, the data transformation s_α and the output list O_α of any path α should also be restricted over the variables $V_0 \cap V_1$ and the inputs I.

Any expression (arithmetic or status) is *defined* over the variable set $V_0 \cap V_1$ if all the variables it involves belong to $V_0 \cap V_1$. A tuple of expressions over V_0 or V_1, restricted to $V_0 \cap V_1$, is its projection over $V_0 \cap V_1$ in which all the component expressions are defined over $V_0 \cap V_1$. The restrictions of R_α and r_α of a path α follow from the restrictions of their constituent expressions as defined above. The restrictions of the condition of execution and the data transformation of a path α on the variable set $V_0 \cap V_1$ are denoted as $R_\alpha|_{V_0 \cap V_1}$ and $r_\alpha|_{V_0 \cap V_1}$, respectively. For example, let $V_0 = \{v_0, v_1, v_2\}$ and $V_1 = \{v_1, v_2, v_3\}$. Therefore, $V_0 \cap V_1$ is $\{v_1, v_2\}$. Let the condition of execution of a path in M_1 be $(v_1 - v_2 > 0 \wedge v_1 \leq v_2 + v_3)$; under restriction to V_1, the condition becomes undefined as v_3 occurs in the conditional expression. Let the data transformation of a path in M_1 be $\langle\langle v_1 - v_2, v_2 + 2, v_3 - v_1\rangle, -\rangle$, where the order of the variables is $v_1 \prec v_2 \prec v_3$. Under restriction to $\{v_1, v_2\}$, the transformation becomes $\langle\langle v_1 - v_2, v_2 + 2\rangle, -\rangle$. Thus, the transformation of this path under restriction to $V_0 \cap V_1$ is defined even if the final value of the variable v_3 is not restricted to $V_0 \cap V_1$ because the final values of v_0 and v_3 are not considered during checking the equality of the data transformations of paths of M_0 and M_1. Consider another path in M_1 whose data transformation is $\langle\langle v_1 - 1, v_2 + v_3, v_2 + 1\rangle, -\rangle$. This transformation becomes undefined when restricted to $\{v_1, v_2\}$ as v_3 occurs in the expression value of $v_2 \in V_0 \cap V_1$. Hence, we have the following definition.

Definition 6: A path α of $M_0 = \langle Q_0, q_{00}, I, V_0, O, f_0, h_0\rangle$ and a path β of $M_1 = \langle Q_1, q_{10}, I, V_1, O, f_1, h_1\rangle$ are said to be equivalent if R_α, r_α, R_β and r_β are defined over $V_0 \cap V_1$ and $R_\alpha|_{V_0 \cap V_1} = R_\beta|_{V_0 \cap V_1}$ and $r_\alpha|_{V_0 \cap V_1} = r_\beta|_{V_0 \cap V_1}$.

The $R_\alpha|_{V_0 \cap V_1}$, $R_\beta|_{V_0 \cap V_1}$, $r_\alpha|_{V_0 \cap V_1}$, and $r_\beta|_{V_0 \cap V_1}$ become undefined even if the scheduler transformations are correct when some of the variables (in $V_0 \cap V_1$) are eliminated or some of the variables (in $V_1 \cap V_0$) are introduced (defined) prior to the start state of the path under consideration. In these cases, our algorithm reports false negative. The second limitation is elaborated in Section VI.

III. Verification Method

Theorem 1 suggests a verification method for checking equivalence of two FSMDs which consists of the following steps.

1) Construct the set P_0 of paths of M_0 so that P_0 covers M_0. Let $P_0 = \{p_{00}, p_{01}, \ldots, p_{0k}\}$.
2) Show that $\forall p_{0i} \in P_0$, there exists a path p_{1j} of M_1 such that $p_{0i} \simeq p_{1j}$.
3) Repeat steps 1 and 2 with M_0 and M_1 interchanged.

Because of loops, it is difficult to find a path cover of the whole computation comprising only finite paths. Therefore, any computation is split into paths by putting *cutpoints* at various places in the FSMD so that each loop is cut in at least one cutpoint. The set of all paths from a cutpoint to another cutpoint without having any intermediary cutpoint is a path cover of the FSMD. The method of decomposing an FSMD by putting cutpoints is identical to the Floyd–Hoare's method of program verification [12], [20], [26]. The choice of cutpoints, however, is nonunique and it is not guaranteed that a path cover of one FSMD obtained from any choice of cutpoints in itself will have the corresponding set of equivalent paths for the other FSMD. It is, therefore, necessary to search for a suitable choice of cutpoints. The question remains whether such a choice can algorithmically be hit upon.

The equivalence problem of FSMDs (EPFSMD) is the same as the equivalence problem of flowchart schemas [21], [31] which is undecidable and not even partially decidable [31]. However, since the final targeted hardware has only a finite data path, the restricted problem can be reduced to the equivalence problem of finite state machine models (EPFSM) which is decidable. Unfortunately, an FSMD with an n-bit data-path results in a number of states of order 2^{kn}, where k is the number of storage elements of n bits. The value of kn easily exceeds several hundreds. Thus, deciding the EPFSMD with a finite data path by reducing them to EPFSM is of little use in practice. On the other hand, specialized analytical treatments, such as the work described here, may aid in revealing problems in the working of the algorithm which may never use the finiteness in producing the output which is to be checked. In this case, the equivalence checking algorithm would identify paths that are not matched, which could be particularly helpful in fixing the scheduling algorithm. This benefit would normally be lost by reducing a finite EPFSMD to EPFSM.

We, therefore, devise a good strategy for setting the cutpoints which would work for many cases but not for all cases. In the

following, we propose one such method which combines the first two steps listed previously into one. More specifically, the method constructs a path cover of M_0 and also finds its equivalent path set in M_1 hand in hand.

We choose the cutpoints in any FSMD as follows.

1) The reset state is chosen.
2) A state q_i is chosen if there is a divergence of flow from q_i. More formally, q_i is a cutpoint if $\exists c_1, c_2 \in S$ such that $c_1 \neq c_2$ and $\langle q_i, c_1, q_j\rangle \in f$ and $\langle q_i, c_2, q_l\rangle \in f$; q_j, q_l are not necessarily distinct.

Obviously, the cutpoints chosen by the aforementioned rules cut each loop of the FSMD in at least one cutpoint because each internal loop has an exit point (ensured by our notion of computation in § II). Corresponding to each path from a cutpoint to a cutpoint without having an intermediary cutpoint, we have to find an equivalent path in the other FSMD which, however, may not exist. Let $p : [q_{0i} \Rightarrow q_{0j}]$ be such a path. The path is modified by concatenating with p all the paths from q_{0j} to the subsequent cutpoints and trying to find their equivalent paths in the other FSMD. The process continues until a path of M_1 that is equivalent to an extended path is obtained or the extension needs to be carried beyond the reset state or the extended path becomes a loop; in the last two cases, the algorithm reports the FSMDs to be possibly non-equivalent. This capability of the present method (of extending paths during equivalence checking) is central to its ability to handle situations where the path structure of the input behavior is changed by a path-based scheduler or an operation is moved beyond BB boundaries. The verification algorithm is as follows.

A. Verification Algorithm

Input: The FSMDs M_0 and M_1.
Output: P_0: a path cover of M_0,
E: ordered pairs $\langle \beta, \alpha\rangle$ of paths of M_0 and M_1, respectively, such that $\beta \in P_0$ and $\beta \simeq \alpha$.

Step 1: Let η be the set of corresponding state pairs.
Let $\eta \leftarrow \langle q_{00}, q_{10}\rangle$.
Insert cutpoints in M_0 using the rule stated in the last section.
Let P_0' be the set of all paths of M_0 from a cutpoint to a cutpoint having no intermediary cutpoint.
Let P_0 and E be empty.

Step 2: If $P_0' = empty$, then return P_0 as a path cover of M_0 and E as the set of ordered pairs of equivalent paths of M_0 (from P_0) and M_1 and *exit (success)*; else go to Step 3.

Step 3: Find a path of the form $\langle q_{0i} \Rightarrow q_{0f}\rangle$ from P_0' s.t. q_{0i} has a corresponding state q_{1j}.
If no path is obtained, then go to Step 4;
else go to Step 5.

Step 4: If $P_0' \neq empty$, then report "M_0 *may not be contained in* M_1" and *exit (failure)*; else return P_0 as a path cover of M_0 and E as a set of ordered pairs of equivalent paths of M_0 (from P_0) and M_1 and *exit (success)*.

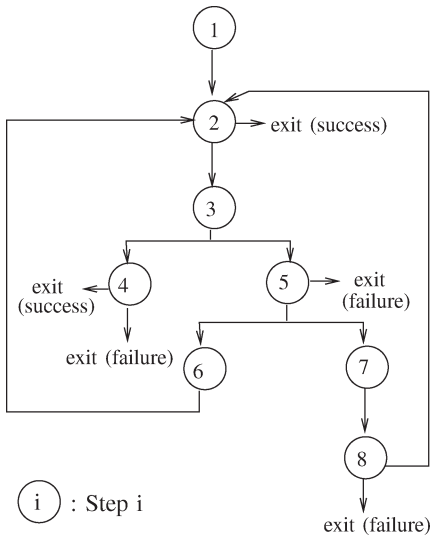


Fig. 3. Control flow of the verification algorithm in terms of steps.

Step 5: Let the path obtained in Step 3 be $\beta = \langle q_{0i} \Rightarrow q_{0f}\rangle$. Let $\langle q_{0i}, q_{1j}\rangle$ be the corresponding state pair in η.
If R_β or r_β is undefined, then report "The R_β and/or r_β of β is not defined and exit (failure),"
else find a path of M_1 emanating from q_{1j} which is equivalent to the path β.
If such a path is found, then go to Step 6;
else go to Step 7.

Step 6: Let this path of M_1 be α.
$\eta \leftarrow \eta \bigcup \{\langle endState(\beta), endState(\alpha)\rangle\}$,
$E \leftarrow E \bigcup \{\langle \beta, \alpha\rangle\}$,
$P_0 \leftarrow P_0 \bigcup \{\beta\}$,
$P_0' \leftarrow P_0' - \{\beta\}$.
go to Step 2.

Step 7: $P_0' \leftarrow P_0' - \{\beta\}$.
Extend $\beta (= \langle q_{0i} \Rightarrow q_{0f}\rangle)$ in M_0 by moving through the cutpoint q_{0f} until the next cutpoints but without moving through the reset state or any cutpoint more than once.
Let B_m be the set of all such extensions of the path β.
$P_0' \leftarrow P_0'$—{paths of P_0' originating from q_{0f} which got appended to β}.
$P_0' \leftarrow P_0' \bigcup B_m$. go to Step 8.

Step 8: If $B_m = empty$, then report "β *may not have any equivalent* in M_1 *and cannot be extended*" and *exit (failure)*; else go to Step 2.

The control flow among the steps of the algorithm is shown in Fig. 3 for clarity. The algorithm examines whether $M_0 \sqsubseteq M_1$. In order to establish the computational equivalence between M_0 and M_1, the aforementioned algorithm is rerun with M_0, M_1 interchanged to determine whether $M_1 \sqsubseteq M_0$ or not.

B. Normalization of Arithmetic Expressions

While finding the equivalent path for a path, it is required to check the equivalence of the respective data transformations

as well as the conditions of execution of the paths which involve arithmetic logic. Linear arithmetic logic is decidable, and an efficient decision procedure has been reported in [6]. However, specifications for digital system implementing algorithmic computation involve the whole of integer arithmetic. Therefore, checking equivalence of paths reduces to the validity problem of a first-order logic which is undecidable; thus, a canonical form does not exist for integer arithmetic. Instead, in this paper, we use the following normal form adapted from [41]. The normalization process reduces many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure [41]. In the following, the normal form chosen for the formulas and the simplification carried out on the normal form during the normalization phase are briefly described.

The data transformation of a path is an ordered tuple $\langle e_i \rangle$ of algebraic expressions such that the expression e_i represents the value of the variable v_i after the execution of the path in terms of the initial data state. Therefore, each arithmetic expression in data transformation can be represented in the *normalized sum form*. A normalized sum is a sum of terms with at least one constant term; each term is a product of primaries with a nonzero constant primary; each primary is a storage variable, an input variable, or of the form $\mathrm{abs}(s)$, $\mathrm{mod}(s_1, s_2)$, or $\mathrm{div}(s_1, s_2)$, where s, s_1, and s_2 are normalized sums. This syntactic entities are defined by means of productions of the following grammar.

Definition 7:

1) $S \rightarrow S + T | c_s$, where c_s is an integer.
2) $T \rightarrow T * P | c_t$, where c_t is an integer.
3) $P \rightarrow \mathrm{abs}\ (S) \mid (S)\ \mathrm{mod}\ (S) \mid S \div C_d\ |v|\ c_m$, where $v \in I \cup V$, and c_m is an integer.
4) $C_d \rightarrow S \div C_d | S$.

Thus, the (integer) modulus and division functions are depicted by infix notation and all functions have arguments in the form of normalized sums. In addition to the aforementioned structure, any normalized sum is arranged by lexicographic ordering of its constituent subexpressions from the bottom most level, i.e., from the level of simple primaries.

A condition of execution of a path is a conjunction of relational and Boolean literals. A Boolean literal is a Boolean variable or its negation. A relational literal is an arithmetic relation of the form $S\ r\ 0$, where S is a normalized sum, and $r \in \{\leq, \geq, =, \neq\}$. The relation $>$ $(<)$ can be reduced to $\geq$ $(\leq)$ over integers. For example, $x - y > 0$ can be reduced to $x - (y - 1) >= 0$. Negated relational literals are suitably modified to absorb the negation.

In addition, the following simplifications are carried out. The common subexpressions in a sum are collected. Thus, $x^2 + 3x + 4z + 7x$ is simplified to $x^2 + 10x + 4z + 0$. A relational literal is reduced by a common constant factor, if any, and the literal is accordingly simplified. For example, $3x^2 + 9xy + 6z + 7 \geq 0$ is simplified to $x^2 + 3xy + 2z + 2 \geq 0$, where $\lfloor 7/3 \rfloor = 2$. Literals are deleted from a conjunction by the rule "if $(l \Rightarrow l')$ then $l \wedge l' \equiv l$". If $l \Rightarrow \neg l'$, then a conjunction having literals l and l' is reduced to false. Implication of a relational literal by another is identified by the method described in [41]. Associativity and commutativity of $+$, $-$, $*$, symmetry of $\{=, \neq\}$, reflexivity of $\{\leq, \geq, =\}$, and irreflexivity of $\{\neq\}$ are accounted for by the aforementioned transformations.

IV. Example

In this section, the working of our algorithm for a path-based scheduler is briefly discussed with the GCD example shown in Fig. 1.

The initial set of cutpoints in M_0 is $\{q_{00}, q_{01}, q_{02}, q_{03}, q_{04}, q_{05}\}$ [Fig. 1(a)] and the initial path cover $P_0' = \{\beta_1 = \langle q_{00} \xrightarrow[\text{true}]{} q_{01}\rangle,\ \beta_2 = \langle q_{01} \xrightarrow[(y1==y2)]{} q_{06} \rightarrow q_{00}\rangle,\ \beta_3 = \langle q_{01} \xrightarrow[!(y1==y2)]{} q_{02}\rangle,\ \beta_4 = \langle q_{02} \xrightarrow[\text{even}(y1)]{} q_{03}\rangle,\ \beta_5 = \langle q_{02} \xrightarrow[!\text{even}(y1)]{} q_{04}\rangle, \beta_6 = \langle q_{03} \xrightarrow[\text{even}(y2)]{} q_{01}\rangle, \beta_7 = \langle q_{03} \xrightarrow[!\text{even}(y2)]{} q_{01}\rangle,\ \beta_8 = \langle q_{04} \xrightarrow[\text{even}(y2)]{} q_{01}\rangle,\ \beta_9 = \langle q_{04} \xrightarrow[!\text{even}(y2)]{} q_{05}\rangle,\ \beta_{10} = \langle q_{05} \xrightarrow[y1>y2]{} q_{01}\rangle, \beta_{11} = \langle q_{05} \xrightarrow[!y1>y2]{} q_{01}\rangle\}$.

The algorithm proceeds in the following sequence:

1) finds α_1 as the equivalent path of β_1;
2) finds α_2 as the equivalent path of β_2;
3) fails to find equivalent path of β_3, hence, extends it; the extended paths are $\beta_3\beta_4$ and $\beta_3\beta_5$;
4) fails to find equivalent path of $\beta_3\beta_4$, hence, extends it; the extended paths are $\beta_3\beta_4\beta_6$ and $\beta_3\beta_4\beta_7$;
5) fails to find equivalent path of $\beta_3\beta_5$, hence, extends it; the extended paths are $\beta_3\beta_5\beta_8$ and $\beta_3\beta_5\beta_9$;
6) and 7) finds α_4 and α_3 as the respective equivalent paths of $\beta_3\beta_4\beta_6$ and $\beta_3\beta_4\beta_7$;
8) finds α_6 as the equivalent path of $\beta_3\beta_5\beta_8$;
9) fails to find equivalent path of $\beta_3\beta_5\beta_9$, hence, extends it; the extended paths are $\beta_3\beta_5\beta_9\beta_{10}$ and $\beta_3\beta_5\beta_9\beta_{11}$;
10) and 11) find α_5 and α_7 as the respective equivalent path of $\beta_3\beta_5\beta_9\beta_{10}$ and $\beta_3\beta_5\beta_9\beta_{11}$.

V. Correctness and Complexity

A. Correctness

Theorem 2 (Partial Correctness): If the algorithm terminates in step 2 or in the else clause of step 4, then $M_0 \sqsubseteq M_1$.

Proof: Let the algorithm terminate in step 2 or in the else clause of step 4. From theorem 1, it follows that $M_0 \sqsubseteq M_1$ if P_0 is a path cover of M_0 and E is the set of ordered pairs of equivalent paths of P_0 and those of M_1. Steps 5 and 6 of the algorithm ensure that E contains only pairs of equivalent paths of M_0 (belonging to P_0) and M_1; this property of E, therefore, is an invariant. Therefore, what remains to be proved is on termination; P_0 is a path cover of M_0 which follows from the following lemma. ■

Lemma 1: When the algorithm terminates successfully the set P_0 gives a path cover of M_0.

Proof: Let C be the set of all the cutpoints in M_0 obtained by the rule given in Section III. Let $C' \subseteq C$ such that every cutpoint in C' has a corresponding state in M_1. There are two "success" exits: one in step 2 and the other in step 4. The algorithm ensures on its "success" exits that P_0 contains all the paths of the form $\langle q_{0l} \Rightarrow q_{0m}\rangle$, where q_{0l}, $q_{0m} \in C'$, and there is no other intermediary cutpoint in the paths belonging

to C'. This property follows from the following observations. If $C' = C$, then the final value of the set P_0 is the same as the initial value of the set P_0' whose members are ensured to satisfy the above property in step 1. If $C' \subset C$, then some of the original paths had to be extended in step 7. Since such extensions took place in all possible ways, i.e., up to all possible successor cutpoints, the above assertion again holds.

Let (the final) P_0 be not a path cover of M_0, i.e., there exists some computation μ_1, for example, of M_0 that cannot be represented by concatenation of members of P_0. Now, since (the initial) P_0' is known to be a path cover of M_0, there exists a concatenated path $\Pi_1 = [p'_{j_1} p'_{j_2} \cdots p'_{j_n}]$, $p'_{j_m} \in P_0'$, $1 \leq m \leq n$, such that $\mu_1 \simeq \Pi_1$. Let p'_{j_m} be $\langle q_{0j_{m-1}} \Rightarrow q_{0j_m} \rangle$. In particular, $p'_{j_1} = \langle q_{0j_0} \Rightarrow q_{0j_1} \rangle$ and $p'_{j_n} = \langle q_{0j_{n-1}} \Rightarrow q_{0j_n} \rangle$, where $q_{0j_0}, q_{0j_n} = q_{00} \in C'$.

Let $\Pi_2 = [p'_{j_k} p'_{j_{k+1}} \cdots p'_{j_l}]$ be the first subsequence in Π_1 such that the start state of p'_{j_k} and the end state of p'_{j_l} are in C', i.e., $q_{0j_{k-1}}, q_{0j_l} \in C'$, and the end state of p'_{j_k}, the start state of p'_{j_l}, and the terminal states of $p'_{j_{k+1}}, \ldots, p'_{j_{l-1}}$ are not in C'. It is obvious that the prefix sequence in Π_1 preceding Π_2 is composed of paths from P_0. We now show that Π_2 can be replaced by a subsequence of paths from P_0. None of $p'_{j_k}, \ldots, p'_{j_l}$ contains any intermediary cutpoint from C and, hence, from C'. Therefore, Π_2 is essentially a path between two cutpoints in C' without having any intermediary cutpoint from C'. Since P_0 contains all such paths, Π_2 belongs to P_0. By repeated applications of the aforementioned argument, all such subsequent instances of Π_2 in Π_1 can be shown to be paths of P_0. Thus, Π_1 is rendered a concatenation of paths from P_0 (contradiction). ■

Theorem 3 (Termination): The algorithm always terminates.

Proof: There are two loops, namely, $\langle 2, 3, 5, 6, 2 \rangle$ and $\langle 2, 3, 5, 7, 8, 2 \rangle$. The first loop pertains to the situation where the equivalent of a path $\beta \in P_0'$ is found and β is deleted from P_0'. Thus, when step 2 is revisited after one execution of this loop, the cardinality $\|P_0'\|$ of P_0' decreases by 1. The second loop pertains to the situation where the equivalent of $\beta = [q_{0i} \Rightarrow q_{0f}] \in P_0'$ is not found. In this case, β is extended by concatenating to itself all the paths of P_0' which emanate from q_{0f} and satisfy the condition stated in the step 7 of the algorithm. There are $\|B_m\|$ such paths. In step 7, the path β and the original $\|B_m\|$ paths emanating from q_{0f} are deleted, and $\|B_m\|$ extended paths are added amounting to a net reduction of $\|P_0'\|$ by 1. Hence, each execution of either loop reduces $\|P_0'\|$ by 1. Since $\|P_0'\|$ is in the well-founded set [31] of nonnegative numbers having no infinite decreasing sequence, the algorithm cannot execute (any combination of) the loops infinitely long. ■

B. Complexity

The overall complexity of the algorithm depends on two issues: the complexity of finding an equivalent path for a given path from a given state [step 5] and the number of iterations of the algorithm. Therefore, the complexity of the algorithm is of the order of number of iterations times the complexity of finding an equivalent path.

Let us first identify the complexity of normalizing a formula F. If $\|F\|$ be the length of the formula (i.e., the number of variables plus that of the operations in F), then the complexity of normalization of F is $O(\|F\|^2)$ due to multiplication of normalized sums. For all other operations, it is linear in $\|F\|$.

Let the function invocation $findequivalent(q_{1j}, \beta)$ find the path of M_1 which starts from the state q_{1j} of M_1 and is equivalent to the path β of M_0. Let us assume that there are n number of states in the FSMD M_1 and the maximum number of parallel edges between any two states is k. Therefore, the maximum possible state transitions from a state are $k \cdot n$. All the transitions emanating from a state have distinct conditions of execution. The function checks all transitions from q_{1j}. The condition of at most one of these transitions matches (may be partially) with R_β. Let the transition be $q_{1j} \rightarrow q_{1k}$. The partially constructed equivalent path α' then becomes $q_{1j} \rightarrow q_{1k}$. If the condition and the data transformation of α' match totally, then the equivalent path has been found. If it matches partially, the function will concatenate the transitions from the end node of α' with this path one by one and check for the equivalence. This process will continue until any repetition of nodes occurs in α' other than between the start node and the end node or the equivalent path has been found. In the worst case, the function iterates n times. Therefore, the complexity of finding the equivalent path is $O(kn \cdot n \cdot \|F\|^2) = O(kn^2 \|F\|^2)$. However, the equivalent path can be found in $O(1 \cdot \|F\|^2)$ time when there is only one transition from q_{1j} which is equivalent to β.

It is required to find the equivalent path for every path in P_0'. Initially, this set contains at most $O(n^2)$ number of paths because, in the worst case, all the nodes of M_0 are cutpoints and the number of paths from one cutpoint to another without any intermediary cutpoint is $n(n-1)/2$. In the best case, the equivalent path for each member of this set can be found directly, and no path extension is required. Also, the number of paths is $O(n)$ (for structured flowchart). In the worst case, one path may be required to be extended n times. In this case, we have to consider $k \cdot (n-1) + k^2 \cdot (n-1) \cdot (n-2) + \cdots + k^{(n-1)} \cdot (n-1) \cdot (n-2), \ldots, 2.1 \simeq k^{(n-1)} \cdot (n-1)^{(n-1)}$ number of paths.

Therefore, the complexity of our algorithm $O(k^{(n-1)}(n-1)^{(n-1)} \cdot kn^2 \cdot \|F\|^2) = O(k^n n^{(n+1)} \cdot \|F\|^2)$ in the worst case and $O(n \cdot 1 \cdot \|F\|^2) = O(n\|F\|^2)$ in the best case.

VI. Performance of the Algorithm for Several High-Level Code Transformations

It may be possible to transform the input behavior to some equivalent description which results in a more efficient scheduled behavior. This fact underlines the need for incorporating high-level code-transformation techniques in the scheduling phase of synthesis to overcome the effects of programming style on the quality of generated circuits. Needless to say, these transformations increase the scheduling verification challenges. In this section, several high-level code-transformation techniques, along with how these can be handled by our algorithm, are discussed. In the examples of this section, the FSMDs that are considered are segments from a cutpoint to cutpoint(s) of the original FSMDs.

A. Renaming and Common Subexpression Elimination (CSE)

Extra variables are used to rename some variables of the original behavior. It provides for parallel execution of some

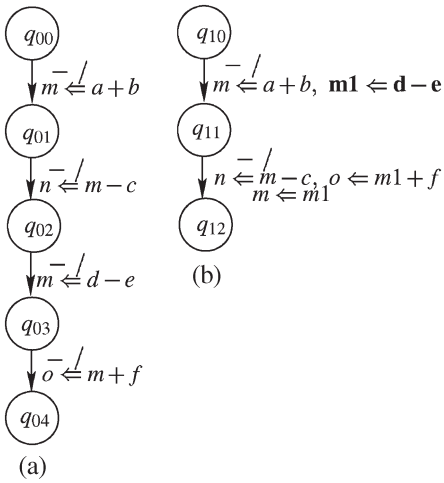


Fig. 4. Scheduling using variable-renaming technique: An example. (a) M_0: The original behaviour. (b) M_1: Scheduled behaviour.

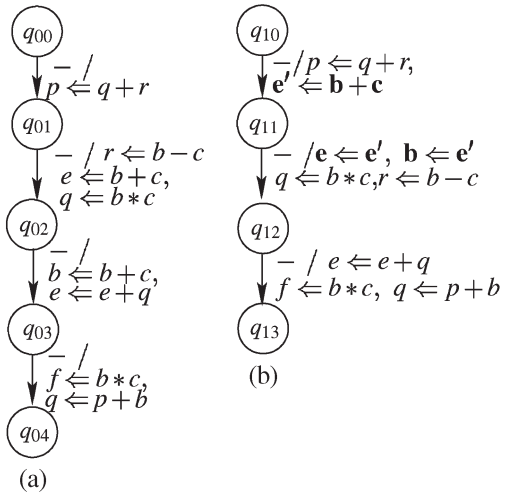


Fig. 5. Scheduling using elimination of common subexpressions: An example. (a) M_0: Original behaviour. (b) M_1: Scheduled behaviour.

operations which were sequential due to data dependency among the variables in the input behavior [19].

Consider the example given in Fig. 4. No parallel execution is possible for the original behavior and it requires four time steps to schedule the behavior. In contrast, the use of an extra variable $m1$ to store the result of the operation $d-e$ removes the write-after-write dependency in the behavior. Consequently, the behavior can be scheduled in two time steps [as shown in Fig. 4(b)].

CSE detects the repeating subexpressions in a piece of code, stores each of them in a variable and reuses the variable wherever the corresponding subexpression occurs subsequently [19].

Consider the FSMD M_0 in Fig. 5. Let two adder/subtractors and a multiplier be available for the design. Then, it requires at least four time steps to schedule the operations in M_0. On the other hand, if the subexpression $b+c$ is stored in a variable e' and is reused subsequently, then this behavior can be scheduled in three time steps, as shown in Fig. 5(b).

The extra variables used during renaming of variables or CSE are defined first and then used. Hence, the data transformation of the common variables would be the same in both the FSMDs. Our algorithm is able to establish the equivalence by considering the restriction of condition of execution (R) and data transformation (r) over the common variables.

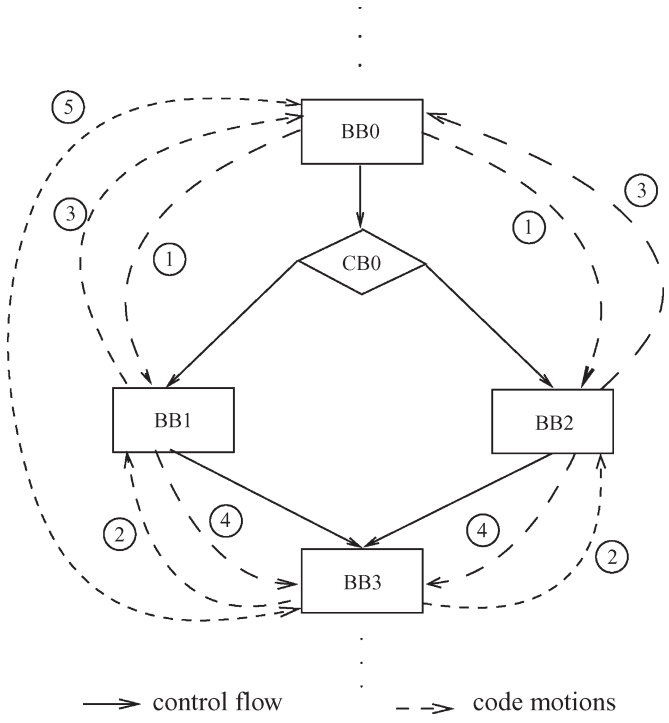


Fig. 6. Various code-motion techniques.

B. Code-Motion Techniques

Code-motion techniques are used for moving operations across the BB boundaries. Several code-motion techniques for global scheduling have been reported in the literature [16], [19], [28], [39], [40]. They can be categorized as follows [39] using Fig. 6.

1) Duplicating Down: It moves operations from a BB preceding a conditional block (CB) to both the succeeding BBs. This is shown by the arcs marked 1 in Fig. 6. Reverse speculation [19], lazy execution [39], and early condition execution [19] belong to this category.

Our algorithm can verify this type of transformations through path extensions. Recall that a conditional state (from which the conditional branches emanate) is a cutpoint in our algorithm. Let q_{0i} be such a state and α be a path that ends in q_{0i}. As some of the operations from α are moved into the conditional branches by this transformation, the equivalent of path α does not exist in the scheduled FSMD. Hence, the algorithm extends the path α through q_{0i}. If the equivalent of the paths obtained by extending α exist in the scheduled FSMD, they can be found by our algorithm. Let us consider the example in Fig. 7. The operation $d \Leftarrow a-e$ is duplicated down to the conditional branches. Here, α is $q_{00} \rightarrow q_{01} \rightarrow q_{02}$ in Fig. 7(a). No equivalent of this path exists in M_1 in Fig. 7(b). Hence, α will be extended through q_{02}. The extended paths are $q_{00} \rightarrow q_{01} \rightarrow q_{02} \underset{c}{\rightarrow} q_{03}$ and $q_{00} \rightarrow q_{01} \rightarrow q_{02} \underset{!c}{\rightarrow} q_{04}$. The corresponding equivalent paths in M_1 are $q_{10} \rightarrow q_{11} \underset{c}{\rightarrow} q_{12}$

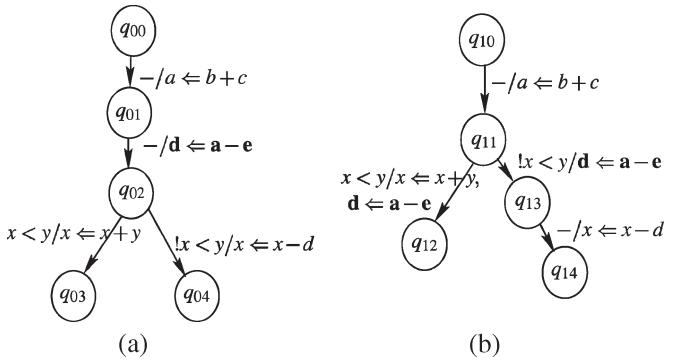


Fig. 7. Duplicating down: An example. (a) M_0. (b) M_1.

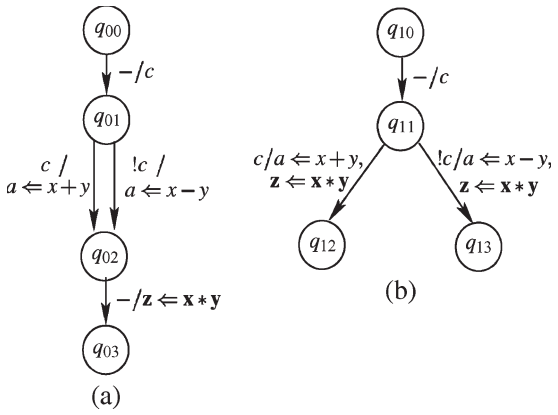


Fig. 8. Duplicating-up technique: An example. (a) M_0. (b) M_1.

and $q_{10} \rightarrow q_{11} \overrightarrow{!c} q_{13} \rightarrow q_{14}$, respectively and are found by the algorithm.

2) Duplicating Up: It moves operations from a BB in which conditional branches merge to its preceding BBs in the conditional branches. This type of code motions is shown by arcs marked 2 in Fig. 6. Conditional speculation [19] and branch balancing [16] fall under this category.

We do not consider the states in which the conditional branches merge as cutpoints. Hence, duplicating up the code motion essentially involves moving up some operations within a path ensuring that the path remains computationally the same in both the FSMDs, i.e., each path has the same condition of execution and data transformation. Hence, the equivalence can be found by our algorithm even without path extension. Let us consider the example shown in Fig. 8. Here, the operation $z \Leftarrow x * y$ is duplicated up in the conditional branches. For the FSMD M_0 in Fig. 8, the paths are $q_{00} \rightarrow q_{01}, q_{01} \overrightarrow{c} q_{02} \rightarrow q_{03}$, and $q_{01} \overrightarrow{!c} q_{02} \rightarrow q_{03}$. The corresponding equivalent paths in Fig. 8(b) are $q_{10} \rightarrow q_{11}$, $q_{11} \overrightarrow{c} q_{12}$, and $q_{11} \overrightarrow{!c} q_{13}$, respectively, and are found successfully by the algorithm.

3) Boosting Up: It moves operations from a BB within a conditional branch to its preceding BB. This category is shown as arcs 3 in Fig. 6. The code-motion techniques, like speculation [19] and loop shifting [18], fall under this category. In this approach, the result of a speculated operation is stored in a new variable. If the condition under which the operation was to execute evaluates to true, then the stored result is copied to the variable from the original operation; else, the stored result is discarded.

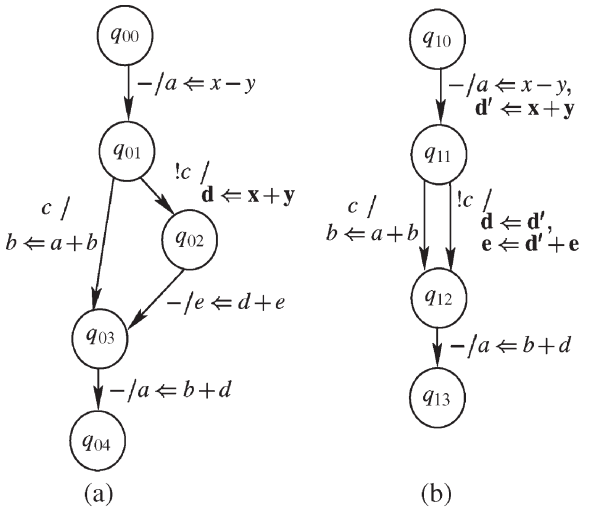


Fig. 9. Boosting-up technique: An example. (a) M_0. (b) M_1.

Our algorithm fails to find the equivalence in this case. As we consider only the variables in $V_0 \cap V_1$ during equivalence checking, the result of the speculated operation is ignored; the algorithm fails to find the equivalent of the subsequent paths as they use the resultant value of the speculated operation. Let us consider the example in Fig. 9. Here, the operation $d \Leftarrow x + y$ is boosted up from the branch with condition $!c$ of the FSMD M_0, and the result of that operation is stored in d'. In this case, our algorithm fails to find the equivalent path of $q_{01} \overrightarrow{!c} q_{02} \rightarrow q_{03} \rightarrow q_{04}$ in M_1 as the values of the common variables a, e, and d, transformed over its equivalent path $q_{11} \overrightarrow{!c} q_{12} \rightarrow q_{13}$, are in terms of the new variable $d' \notin V_0$.

4) Boosting Down: It moves operations from a BB within a conditional branch to a BB following the merging of the conditional branches [39]. This is shown as (broken) arcs 4 in Fig. 6.

In this case also, the operations are moved down within a path ensuring that the path remains computationally the same in both the FSMDs. Hence, our algorithm can successfully establish the equivalence.

5) Useful Move: It moves an operation to a control and data equivalent block. This is shown as the (bidirectional) arc 5 in Fig. 6. Moving an operation from BB3 to BB0 and vice versa is possible if this operation is independent of the operations in BB1 and BB2. This code motion is preferred since substantial fine-grain parallelism can be extracted at no extra cost [11]. This type of transformation can also be verified by our algorithm through path extensions.

VII. Multicycle and Pipelined Operations

Multicycle execution of an operation requires two or more control steps [Fig. 10(a)]. Pipelining the pairs of operands over the cycles of a multicycle execution of an operation p allows concurrent execution of p on these pairs of operands, each pair getting partially computed in one stage of the pipelined unit [Fig. 10(b)] resulting in a better utilization of the (pipelined) unit.

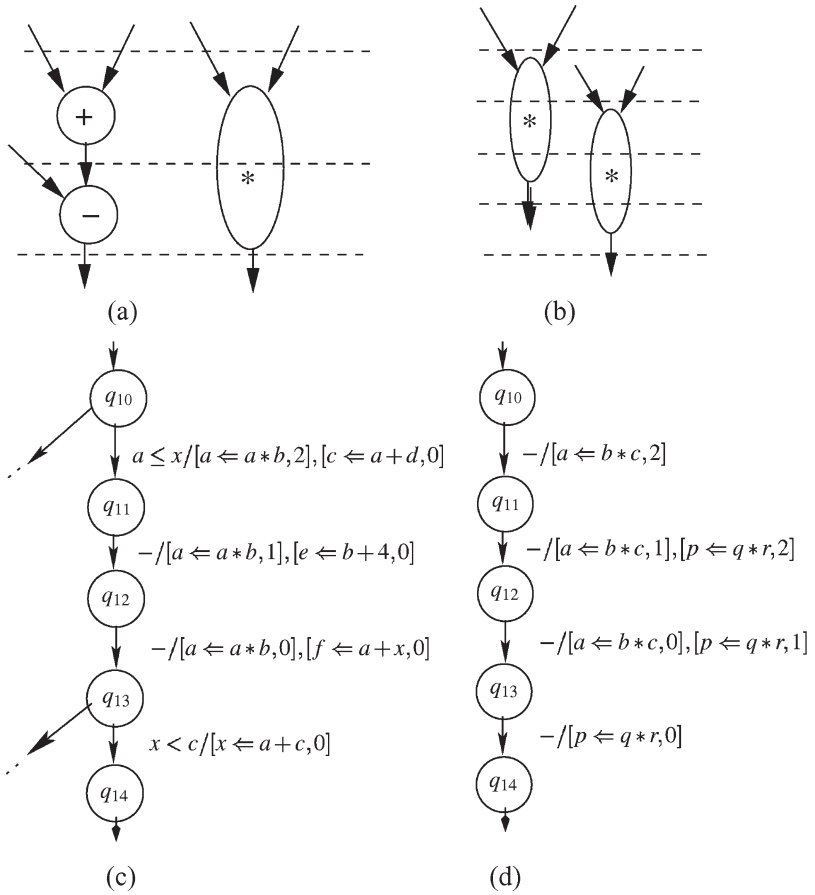


Fig. 10. (a) Schedule with a two-cycle multiplier. (b) Schedule with a three-stage pipelined multiplier. (c) A sample path in a scheduled FSMD with a multicycle operation. (d) A sample path in a scheduled FSMD with a pipelined operation.

Our algorithm can handle multicycle and pipelined execution of the operations with the following modification of the update function in the FSMD definition in Section II: $h : Q \times 2^S \rightarrow U \times N$, where Q, S and U are the same as before and N represents the set of natural numbers. Specifically, $h(q, \text{true}) = (a \Leftarrow a * b, t)$ represents that t more time steps are required to complete the operation "$*$;" this second member t of the value tuple of h is called the span of that operation. For a single-cycle operation, t should be zero. The input behavior does not specify whether an operation is multicycle or pipelined. Hence, all operations in the input FSMD are represented as single-cycle ones. Depending upon the type of the operators, the operations are scheduled in a multicycle or pipelined manner by the scheduler. Therefore, these would be reflected in the scheduled FSMD. In Fig. 10(c), the operation $a \Leftarrow a * b$ is scheduled in a three-cycle multiplier. In Fig. 10(d), the operations $a \Leftarrow b * c$ and $p \Leftarrow q * r$ are scheduled using a three-stage pipelined multiplier in Fig. 10(d). The value produced by a multicycle or a pipelined operation is available only when the span of the operation becomes zero. Therefore, during computation of the condition of execution and the data transformations of a path in a scheduled FSMD, an operation will be considered only when its span value becomes zero. For example, for the path shown in Fig. 10(c), the condition of execution is $a \le x \wedge x < (a + d)$ and the data transformation is $\langle\langle a * b, b, a + d, d, b + 4, a + x, a * b + a + d\rangle, -\rangle$, where the variables are in order $\langle a \prec b \prec c \prec d \prec e \prec f \prec x\rangle$. It may be noted from the data transformation that the operation $f \Leftarrow a + x$ uses the old value a whereas the operation $x \Leftarrow a + c$ uses the updated value of a.

TABLE I
CHARACTERISTICS OF THE HLS BENCHMARKS USED IN OUR EXPERIMENT

Benchmark	*#BBs*	*#ifs*	*#loops*	*#operations*	*#variables*
DIFFEQ	4	1	1	19	12
EWF	1	0	0	53	37
DCT	1	0	0	58	53
GCD	7	5	1	9	3
TLC	17	6	5	28	5
MODN	7	4	1	12	9
PERFECT	6	3	1	8	5
IEEE754	40	28	7	74	37
LRU	22	19	7	49	19
DHRC	7	14	10	131	72
BARCODE	28	25	11	52	17
PRAWN	85	53	0	227	96

The presence of multicycle and/or pipelined operations does not create any problem for selecting the cutpoints. Let a three-cycle operation p be conceived in the prescheduled FSMD in a transition $q_l \rightarrow q_m$; let, in the scheduled FSMD, p span over the transition sequence $q_i \rightarrow q_j \rightarrow q_k \rightarrow q_s$, where q_i corresponds to q_l, and q_s corresponds to q_m. It is obvious that there cannot be any bifurcation of flow from the intermediary states q_j or q_k in the scheduled behavior. For the pipelined operation, however, another aspect is to be checked: whether the sequence in which the operand pairs appear at the input stage of the pipeline is in

TABLE II
VERIFICATION RESULTS FOR SEVERAL HLS BENCHMARKS

Benchmarks	in M_0		*for path-based scheduler*				*for SPARK*			
	#states	*#paths in* P_0'	*#paths in* P_0	*#path-merges*	*iterations*	*#path extensions*	*#paths in* P_0	*#operations moved across BB*	*iterations*	*#path extensions*
DIFFEQ	9	3	3	0	3	0	3	0	3	0
EWF	17	1	1	0	1	0	1	0	1	0
DCT	10	1	1	0	1	0	1	0	1	0
GCD	7	11	7	8	11	4	7	0	11	0
TLC	7	20	16	14	16	4	20	4	20	0
MODN	6	9	8	3	14	2	9	4	11	3
PERFECT	9	7	7	3	7	1	7	0	7	0
IEEE754	55	59	50	33	61	11	59	10	73	6
LRU	33	39	38	3	39	1	39	8	39	0
DHRC	62	27	24	6	27	3	27	0	27	0
BARCODE	32	55	57	21	79	18	55	0	55	0
PRAWN	122	154	152	20	186	28	154	15	172	14

keeping with that in which the destination registers are strobed. This aspect is verified in the data path and controller synthesis phase, which is outside the purview of this paper.

VIII. EXPERIMENTAL RESULTS

The equivalence checking algorithm described in this paper has been implemented in C and has been run on a 1.70-GHz Intel Pentium 4 machine with 256-MB RAM for 12 HLS benchmarks shown in Table I. Some of the benchmarks, such as DIFFEQ [14], EWF, and DCT, are data intensive, some are control intensive, such as GCD, TLC, BARCODE [35], etc., whereas some are both data and control intensive, such as IEEE754, LRU, and PRAWN [35]. Table I lists the characteristics of these HLS benchmarks in terms of numbers of if-then-else CBs, loops, nonempty BBs, three address operations, and variables. The benchmarks are scheduled using three types of scheduler: a BB-based scheduler SAST [30], a path-based scheduler [7] and the scheduler of the synthesis tool SPARK [17] that employs a set of code transformations during scheduling. The SPARK scheduler is allowed to perform only those code transformations which have been identified in the previous section as successfully verifiable by our algorithm.

The verification results for the path-based scheduler and the SPARK scheduler are shown in Table II. Since, the BB-based scheduler neither changes the path structure of the input behavior nor moves operations beyond BB boundaries, the performance of the algorithm for this scheduler has been straightforward and, hence, not presented here. The number of states in the FSMD M_0 and the number of paths in the initial path cover (P_0') of M_0 for these benchmarks are given in this table. For both the path-based scheduler and the SPARK scheduler, the number of paths in the computed path cover P_0, the number of iterations of the algorithm, that is, the number of visits of step 2) (Fig. 3), and that of path extensions required to show the equivalence between the FSMDs are tabulated for each benchmark example. In addition, for the path-based scheduler, the number of path merges during scheduling and for the SPARK scheduler, the number of operations that moved across BB boundaries, are shown. It is clear from the table that these schedulers transformed the input behavior significantly. As for example, 20 paths were merged during the path-based scheduling of PRAWN. However, the algorithm took only 186 iterations out of which 28 needed path extensions. The initial path cover P_0' of PRAWN contains 154 paths. It may be noted that the equivalence checker iterated 154 times if it was scheduled using a BB-based scheduler. Therefore, the algorithm took only 32 extra iterations for the path-based scheduler. Similarly, it took only 18 extra iterations for the SPARK scheduler in this case. The numbers of iterations for all these examples in both the cases also suggest that the upper bound of complexity is not necessarily hit for practical scheduling verification cases.

TABLE III
COMPARISON OF EXECUTION TIMES OF THE VERIFIER FOR DIFFERENT TYPES OF SCHEDULERS ON SEVERAL HLS BENCHMARKS

Benchmark	*BB-based* (in ms)	*path based* (in ms)	*SPARK* (in ms)
DIFFEQ	2.4	2.4	2.4
EWF	1.8	1.8	1.8
DCT	2	2	2
GCD	3	4	3
TLC	3.5	4.5	4
MODN	3	4.3	3.5
PERFECT	2.5	4.1	3
IEEE754	150	203	222
LRU	25	26	34
DHRC	60	60	67
BARCODE	102	197	144
PRAWN	332	1002	460

The average CPU time units needed by the algorithm for BB-based scheduler, the path-based scheduler and the SPARK scheduler are shown in Table III. It is evident from this table that the average CPU time needed for the path-based scheduler and that for the SPARK scheduler are not too high compared to that needed for the BB-based scheduler even though the input behaviors are transformed significantly in the former two cases. For example, 33 paths were merged during the path-based scheduling, and 10 operations were moved across the BB boundaries for the IEEE754 example but the CPU time units were only increased by 53 and 72 ms, respectively, which are compared to its CPU time for the BB-based scheduler. Also, the run time of this algorithm is less sensitive to the number of states in the FSMDs than to the number of paths. For example, the run times of EWF and DCT are small compared to TLC and MODN even though EWF and DCT have greater number of states.

IX. Conclusion

Advances in VLSI technology have enabled its deployment into complex circuits. Synthesis flow of such circuits comprises of various phases where each phase performs the task algorithmically, providing for ingenious interventions of experts. The gap between the original behavior and the finally synthesized circuits is too wide to be analyzed by any reasoning mechanism. The validation tasks, therefore, must be planned to go hand in hand with each phase of synthesis. This paper concerns itself with the validation of the scheduling phase. Both the behaviors prior to and after scheduling have been modeled as FSMDs. The validation task has been treated as an EPFSMD.

The method presented in this paper has been proved to be sound, with its completeness being ruled out by the fact that the EPFSMD has been reported to be not even partially decidable. The method is strong enough to accommodate merging of the segments in the original behavior by typical path-based schedulers. The method is also capable of verifying several code-transformation techniques such as renaming, CSE, duplicating down, duplicating up, boosting down and useful move. On the other hand, boosting up cannot be handled by the present algorithm. Because of the normalization of expressions at all levels of processing, the method is also able to handle arithmetic transformations. Similar methods reported in the literature have been found to fail under such situations. The extendability of the method for verification of behaviors involving pipelined and multicycle operations is also discussed. The applicability of our method for the allocation, binding, and data-path and controller generation phases of HLS has been demonstrated in [24]. Although the proposed method is found to have a nonpolynomial worst case complexity, the best case complexity is found to be linear. The experimental results demonstrate that the worst case complexity is not necessarily hit for practical and nontrivial equivalence checking problems. In fact, none of the examples in this paper hit the worst case bound. The experiments on several HLS benchmarks, including some of the significant complexities such as IEEE754 floating point unit and PRAWN processor, indicate that the algorithm is usable for checking many commonly used scheduling optimizations.

Acknowledgment

The authors would like to thank the anonymous reviewers of this paper for their comments and suggestions which helped immensely to improve this paper.

References

[1] D. P. Anderson and J. Ainscough, "The verification of scheduling algorithms," in *Proc. IEE Colloq. Structured Methods Hardware Syst. Des.*, 1994, pp. 7/1–7/5.

[2] P. Ashar, S. Bhattacharya, A. Raghunathan, and A. Mukaiyama, "Verification of RTL generated from scheduled behavior in a high-level synthesis flow," in *Proc. IEEE/ACM ICCAD*, 1998, pp. 517–524.

[3] R. A. Bergamaschi, R. A. O'Connor, L. Stok, M. Z. Moricz, S. Prakash, A. Kuehlmann, and D. S. Rao, "High-level synthesis in an industrial environment," *IBM J. Res. Develop.*, vol. 39, no. 1/2, pp. 131–148, Jan.–Mar. 1995.

[4] C. Blumenrohr, D. Eisenbiegler, and R. Kumar, "Applicability of formal synthesis illustrated via scheduling," in *Proc. IWLAS*, 1996, pp. 345–352.

[5] D. Borrione, J. Dushina, and L. Pierre, "A compositional model for the functional verification of high-level synthesis results," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 5, pp. 526–530, Oct. 2000.

[6] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani, "An incremental and layered procedure for the satisfiability of linear arithmetic logic," in *Proc. TACAS*, 2005, pp. 317–333.

[7] R. Camposano, "Path-based scheduling for synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 10, no. 1, pp. 85–93, Jan. 1991.

[8] R. Chapman, G. Brown, and M. Leeser, "Verified high-level synthesis in BEDROC," in *Proc. DAC*, 1992, pp. 59–63.

[9] R. Ernst and J. Bhasker, "Simulation-based verification for high-level synthesis," *IEEE Des. Test Comput.*, vol. 8, no. 1, pp. 14–20, Mar. 1991.

[10] H. Eveking, H. Hinrichsen, and G. Ritter, "Automatic verification of scheduling results in high-level synthesis," in *Proc. DATE*, Mar. 1999, pp. 59–64.

[11] Y. Fann, M. Rim, and R. Jain, "Global scheduling for high-level synthesis applications," in *Proc. 31st DAC*, 1994, pp. 542–546.

[12] R. W. Floyd, "Assigning meaning to programs," in *Proc. 19th Symp. Appl. Math.*, J. T. Schwartz, Ed., 1967, pp. 19–32.

[13] M. Fujita, "Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 4, pp. 610–626, Oct. 2005.

[14] D. Gajski, N. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Norwell, MA: Kluwer, 1992.

[15] D. Gries, *The Science of Programming*. New York: Springer-Verlag, 1987.

[16] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs," in *Proc. DATE*, 2003, pp. 270–275.

[17] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark: A high-level synthesis framework for applying parallelizing compiler transformations," in *Proc. 16th Int. Conf. VLSI Des.*, Jan. 2003, pp. 461–466.

[18] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Loop shifting and compaction for the high-level synthesis of designs with complex control flow," in *Proc. DATE*, Feb. 2004, pp. 114–119.

[19] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Using global code motions to improve the quality of results for high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 2, pp. 302–312, Feb. 2004.

[20] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[21] W. E. Howden, *Functional Program Testing and Analysis*. New York: McGraw-Hill, 1987.

[22] R. Jain, A. Majumdar, A. Sharma, and H. Wang, "Empirical evaluation of some high-level synthesis scheduling heuristics," in *Proc. 28th IEEE DAC*, 1991, pp. 686–689.

[23] C. Karfa, C. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade, "A formal verification method of scheduling in high-level synthesis," in *Proc. ISQED*, Mar. 2006, pp. 71–78.

[24] C. Karfa, D. Sarkar, C. Mandal, and C. Reade, "Hand-in-hand verification of high-level synthesis," in *Proc. GLSVLSI*, 2007, pp. 429–434.

[25] Y. Kim, S. Kopuri, and N. Mansouri, "Automated formal verification of scheduling process using finite state machines with datapath (FSMD)," in *Proc. ISQED*, San Jose, CA, Mar. 2004, pp. 110–115.

[26] J. C. King, "Program correctness: On inductive assertion methods," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 5, pp. 465–479, Sep. 1980.

[27] R. Kumar, C. Blumenhr, and D. Schmid, "Formal synthesis in circuit design—A classification and survey," in *Proc. FMCAD*, 1996, vol. 1166, pp. 294–309.

[28] G. Lakshminarayana, A. Raghunathan, and N. K. Jha, "Incorporating speculative execution into scheduling of control-flow-intensive designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 3, pp. 308–324, Mar. 2000.

[29] J. Lee, Y. Hsu, and Y. Lin, "A new integer linear programming formulation for the scheduling problem in data path synthesis," in *Proc. ICCAD*, 1989, pp. 20–23.

[30] C. Mandal and R. M. Zimmer, "A genetic algorithm for the synthesis of structured data paths," in *Proc. 13th Int. Conf. VLSI Des.*, 2000, pp. 206–211.

[31] Z. Manna, *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.

[32] N. Mansouri and R. Vemuri, "A methodology for automated verification of synthesized RTL designs and its integration with a high-level synthesis tool," in *Proc. FMCAD*, 1998, pp. 204–221.

[33] J. M. Mendias, R. Hermida, M. C. Molina, and O. Penalba, "Efficient verification of scheduling, allocation and binding in high-level synthesis," in *Proc. Digit. Syst. Des.*, 2002, pp. 308–315.
[34] N. Narashima, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri, "Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis," in *Proc. ICCAD*, 1998, pp. 392–399.
[35] P. R. Panda and N. D. Dutt, "1995 high level synthesis design repository," in *Proc. 8th Int. Symp. Syst. Synth.*, 1995, pp. 170–174.
[36] R. Radhakrishnan, E. Teica, and R. Vemuri, "Verification of basic block schedules using RTL transformations," in *Proc. CHARME*, 2001, pp. 173–178.
[37] R. Radhakrishnan, E. Teica, and R. Vermuri, "An approach to high-level synthesis system validation using formally verified transformations," in *Proc. HLDVT*, Washington DC, 2000, pp. 80–85.
[38] M. Rahmouni and A. A. Jerraya, "Formulation and evaluation of scheduling techniques for control flow graphs," in *Proc. EuroDAC*, Brighton, U.K., Sep. 18–22, 1995, pp. 386–391.
[39] M. Rim, Y. Fann, and R. Jain, "Global scheduling with code-motions for high-level synthesis applications," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 3, no. 3, pp. 379–392, Sep. 1995.
[40] L. C. V. dos Santos and J. A. G. Jress, "A reordering technique for efficient code motion," in *Proc. 36th ACM/IEEE DAC*, 1999, pp. 296–299.
[41] D. Sarkar and S. C. De Sarkar, "Some inference rules for integer arithmetic for verification of flowchart programs on integers," *IEEE Trans. Softw. Eng.*, vol. 15, no. 1, pp. 1–9, Jan. 1989.

Chandan Karfa received the B.Tech. degree in information technology from the University of Kalyani, Kalyani, India, in 2004 and the M.S. (by research) degree in computer science and engineering from the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India, in 2007, where he is currently working toward the Ph.D. degree.

His research interests include synthesis and verification of digital VLSI circuits.

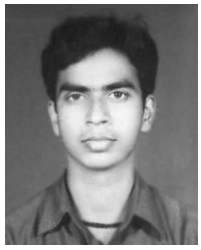
Dipankar Sarkar received the B.Tech. and M.Tech. degrees in electronics and electrical communication engineering and the Ph.D. degree in engineering from the Indian Institute of Technology (IIT), Kharagpur, India.

He has been a faculty member with the IIT for the last 26 years. His area of research interest is in formal verification of circuits and systems.

Chittaranjan Mandal received the Ph.D. degree from the Indian Institute of Technology (IIT), Kharagpur, India, in 1997.

He is currently an Associate Professor with the Department of Computer Science and Engineering, IIT. Prior to joining IIT, he served as a Reader with Jadavpur University, Calcutta, India. His research interests include formal modeling, high-level design, and web technologies.

Prof. Mandal has been an Industrial Fellow of Kingston University, London, U.K., since 2000. He was a recipient of a Royal Society Fellowship.



Pramod Kumar is currently working toward a dual degree in the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India.

His special research interests include algorithm design and VLSI design.