Verification of Scheduling in High-level Synthesis

C Karfa C Mandal D Sarkar S R Pentakota
Department of Computer Sc & Engg
Indian Institute of Technology, Kharagpur
WB 721302, INDIA
{ckarfa, chitta, ds}@iitkgp.ac.in, satya@ti.com

Chris Reade
Kingston Business School
Kingston University
England KT2 7LB, UK
Chris.Reade@king.ac.uk

Abstract

This paper describes a formal method for checking the equivalence between two descriptions of the target system, one before and the other after scheduling. The descriptions are represented as finite state machines with data paths (FSMD). The basic principle is to show that any computation of one FSMD is covered by a computation on the other, a computation being characterized by a concatenation of paths in the FSMD. These notions are formalized in the paper. The method is strong enough to accommodate merging of the segments in the original behaviour by the typical scheduler such as DLS, a feature common in scheduling. The method also works for limited arithmetic transformations. Although the proposed method is found to have a non-polynomial worst case complexity, many non-trivial examples encounter a low polynomial order of complexity. The technique is illustrated with an example.

1 Introduction

High-level synthesis is the process of generating the register transfer level (RTL) design from the behavioural description. The synthesis process consists of several interdependent sub-tasks such as, specification, compilation, scheduling, allocation and binding. The operations in the behavioural description are assigned time steps through the scheduling process. Input to the scheduling phase is a control data flow graph (CDFG)[3]. While a CDFG is better suited to scheduling algorithms, an FSMD is a more appropriate model for verification. We therefore construct FSMDs from the CDFGs before and after scheduling. In the process of scheduling, operations are often moved across basic block boundaries for various optimizations. In general several transformations may be made to improve the performance of a design. For example, path based scheduling techniques [7] perform several such non-trivial path based transformations. Hence, it is important to ensure that the scheduling process preserves the behaviour of the original specification, irrespective of the scheduling technique that is used. The objective of this work is to check that the design descriptions before and after scheduling, as represented by FSMDs, are computationally equivalent.

The equivalence problem of FSMDs (EPFSMD) is the same as the equivalence problem of flowchart schemas[4, 6] which is undecidable and not even partially decidable[6]. However, since the final targeted hardware has only a finite datapath, the restricted problem can be reduced to the equivalence problem of FSM models (EPFSM) which is decidable. Unfortunately, an FSMD with an n-bit datapath results in a number of states of the order of 2^{kn}, where k is the number of storage elements of n bits. The value of kn easily exceeds several hundreds. Thus, deciding EPFSMD with a finite datapath 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 up, which could be particularly helpful in fixing the scheduling algorithm. This benefit would normally be lost by trying to reduce a finite EPFSMD to EPFSM.

Most of the algorithms proposed in the literature can successfully verify the basic block based scheduling but apparently fail to verify when structure of the scheduled FSMD differs from the input FSMD due to path based transformation. In this paper, we propose a scheduling verification method which is strong enough to work even when the basic path structure is changed by the scheduler. This method formally establishes equivalence between the FSMDs before and after scheduling.

This paper is organized as follows. In section 2, FSMDs and the notions of computations on FSMDs and the equivalence of FSMDs are defined. The verification method is described in section 3. the complexity of the proposed method is treated in section 4. An example has been treated in sec-

Proceedings of the 2006 Emerging VLSI Technologies and Architectures (ISVLSI'06)
0-7695-2533-4/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

tion 5 to illustrate the working of the algorithm. Some experimental results have been given in section 6. The paper is concluded in section 7.

2 FSMDs and their Equivalence

2.1 FSMDs

An FSMD (*finite state machine with data-path*) is a universal specification model, proposed by Gajski in [2], that can represent all hardware designs. The model is used in the present work with the addition of a reset state, for encoding the designs to be verified. 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 and Σ_I is the input alphabet,
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 and Σ_O is the output alphabet,
6. $f : Q \times S \rightarrow Q$, is the state transition function and
7. $h : Q \times S \rightarrow U$, is the update function of the output and the storage variables, where U and S are as defined below.
   (a) $U = \{x \Leftarrow e | x \in O \cup V$ and $e \in E\}$ represents a set of storage or output assignments, from variables (storage or output) or expressions constructed over (input or storage) variables. Thus, $E = \{g(x, y, z, \ldots) \mid x, y, z, \ldots \in I \cup V\}$ represents a set of arithmetic expressions over the set $I \cup V$.
   (b) $S = \{R(a, b) | a, b \in E$ and R is any arithmetic relation$\}$ represents a set of status signals as a result of comparisons $(=, \neq, >, \geq, <, \leq)$ between two expressions from the set E.

It may be noted that we have not introduced final states in the FSMD model as we assume that the systems work in an infinite outer loop.

2.2 Walks and Transformations along a Walk

A (finite) *walk* α from q_i to q_j, where $q_i, q_j \in Q$, is a finite transition sequence of states of the form $\langle q_i = q_1 \xrightarrow[c_1]{} q_2 \xrightarrow[c_2]{}, \ldots, \xrightarrow[c_{n-1}]{} q_n = q_j \rangle$ such that $\forall l, 1 \leq l \leq n-1, \exists c_l \in S$ 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 q_1. The *condition of execution of the walk* $\alpha = \langle q_{l_0} \xrightarrow[c_0]{} q_{l_1} \xrightarrow[c_1]{} q_{l_2} \cdots \xrightarrow[c_{k-1}]{} q_{l_k} \rangle$, R_α, is a logical expression over the variables in V such that R_α is satisfied by the (initial) data state at q_{l_0} iff the walk α is traversed.

We assume that inputs and outputs occur through named ports. The ith input from port P is a value represented as P_i. Thus if some variable v stores input from port P (for the ith time along a walk), it is equivalent to the assignment $v \Leftarrow P_i$.

The simple data transformation of a walk α over V (s_α): It is an ordered tuple $\langle e_i \rangle$ of algebraic expressions over the variables in V and the inputs in I such that the expression e_i represents the value of the variable v_i after the execution of the walk in terms of the initial data state (i.e., the values of the variables at the initial control state) of the walk.

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

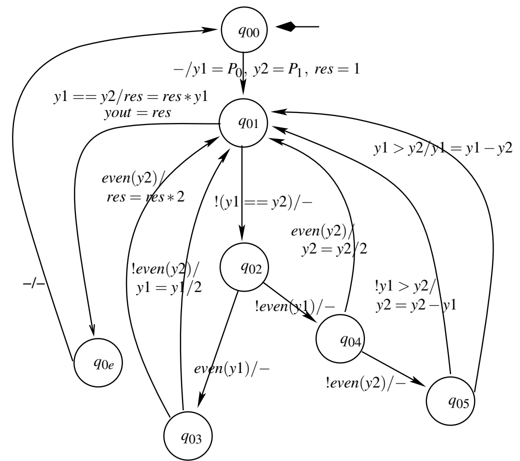

Figure 1. M_0, the FSMD of GCD before scheduling

Computation of the condition of execution R_α can be by *backward* substitution or by *forward* substitution. The former is more readily described and is based on the following rule: If a predicate $c(y)$ is true after execution of $y \leftarrow g(y)$, then the predicate $c(g(y))$ must have been true before the execution of the statement [6]. The transformation s_α is found indirectly using the same principle. The forward substitution method of finding R_α is based on symbolic execution.

0-7695-2533-4/06 $20.00 © 2006 **IEEE**

IEEE COMPUTER SOCIETY

2.3 Characterization of Walks and their Concatenations

The characteristic formula τ_α of a walk α with initial storage and input variables as $\overline{v}$, final variables as $\overline{v}_f$ and outputs along the walk as O is $\tau_\alpha(\overline{v},\overline{v}_f,O) = R_\alpha(\overline{v}) \wedge (\overline{v}_f = s_\alpha(\overline{v})) \wedge (O = O_\alpha(\overline{v}))$, where s_α is the data transformation and O_α output list in the walk α.

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 walk α and $\tau_\beta(\overline{v},\overline{v}_f,O) : R_\beta(\overline{v}) \wedge (\overline{v}_f = s_\beta(\overline{v})) \wedge (O = O_\beta(\overline{v}))$ be the characteristic formula of the walk β. The characteristic formula for the concatenated walk $\alpha\beta$ 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 $O_\beta(s_\alpha(\overline{v}))$. The detail of incrementing the input indices on each port in the formulas for β to start after the last index of the corresponding port in α has been omitted for notational clarity.

2.4 Computations on FSMDs and their Path Covers

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 (as a new computation starts from the reset state). A computation c of an FSMD M may be characterized as $\tau_c(\overline{v}_i,\overline{v}_f,O) : R_c(\overline{v}_i) \wedge (\overline{v}_f = s_c(\overline{v}_i)) \wedge (O = O_c(\overline{v}_i))$, where $\overline{v}_i$ is the vector of initial input and data state with which the computation is started, R_c is a satisfiable condition over the domain of I and V, s_c is a function over this domain to the co-domain of values over V and O_c is the concatenation of the output lists resulting from output operations along c.

Two computations c_1 and c_2 having the characteristic formulae τ_{c_1} and τ_{c_2}, respectively, are said to be equivalent if $R_{c_1} = R_{c_2}$, $r_{c_1} = r_{c_2}$. The computational equivalence of two walks p_1 and p_2 is denoted as $p_1 \simeq p_2$. Equivalence checking of walks, therefore, consists in establishing the computational equivalence of the respective conditions of execution and the respective data transformations.

A finite set of paths[1] $P = \{p_0, p_1, p_2, \ldots, p_k\}$ is said to cover an FSMD M if any computation c of M can be looked upon as a concatenation of paths from P. P is said to be a *finite path cover* of the FSMD M.

[1] A path is a walk in which all the states (nodes) are distinct. A cycle is like a path where the first and last nodes are identical but all other nodes are distinct. Here we allow our paths to be cycles also.

2.5 Arithmetic Expressions and their Normalization

Since the condition of execution and the data transformation of a walk involve the whole of integer arithmetic, checking equivalence of walks reduces to the validity problem of first order logic; the latter is undecidable because a canonical form does not exist for integer arithmetic. Instead, in this work we use the following normal form adapted from [5, 8].

Every formula is converted into the conjunctive normal form; every conjunct, therefore, is a disjunction of literals where a literal is an atomic formula (atom) or its negation. An atom is a boolean variable or an arithmetic relation of the form $S\ r\ 0$, where S is a normalized sum, $r \in \{\leq, \geq, =, \neq\}$. The relation $>$ ($<$) can be reduced to $\geq$ ($\leq$) over integers. A normalized sum is a sum of terms with at least one constant term; each term is a product of primaries with a non-zero constant primary; each primary is a storage variable, an input variable or an output variable or of the form $abs(s)$, $mod(s_1,s_2)$, $exp(s_1.s_2)$ or $div(s_1,s_2)$, where s, s_1, and s_2 are normalized sums. Any normalized sum is arranged by lexicographic ordering of its constituent subexpressions from the bottom-most level. 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 $\lfloor 7/3 \rfloor = 2$. A conjunct $C = l_1 \vee l_2 \vee \ldots \vee l_n$ is first expressed as $\neg(\neg l_1 \wedge \neg l_2 \wedge \ldots \wedge \neg l_n)$ and then literals are deleted by the rule "if $(l \Rightarrow l')$ then $l \wedge l' \equiv l$." C reduces to *true* if $\neg l_i \Rightarrow l_j$ for $1 \leq i, j \leq n$. Symmetry of $\{=, \neq\}$, reflexivity of $\{\leq, \geq, =\}$ and irreflexivity of $\{\neq\}$ are accounted for by the above transformations. The above normal form may be shown to be canonical for multivariate polynomials.

2.6 Equivalence of FSMDs

Let M_0 be the FSMD representation of the CDFG given as the input to the scheduler and M_1 be the FSMD of the scheduled behaviour. 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 re-visited, 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.

Thus two FSMDs M_0 and M_1 are said to be computationally equivalent if for any computation c_0 of M_0, there exists a computation c_1 of M_1 such that c_0 and c_1 are computationally equivalent and vice-versa.

0-7695-2533-4/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

The following theorem, stated without proof, is key to our algorithm for checking the equivalence of two FSMDs.

Theorem 1 *Two FSMDs M_0 and M_1 are computationally equivalent if there exists a finite cover $P_0 = \{p_{00}, p_{01}, \ldots, p_{0l}\}$ of M_0 for which there exists a set $P_1^0 = \{p_{10}^0,\ p_{11}^0, \ldots,\ p_{1l}^0\}$ of paths of M_1 such that $p_{0i} \simeq p_{1i}^0$, $0 \leq i \leq l$ and vice-versa.*

The following (inductive) notion of *corresponding states* will be used in the algorithm to be presented. 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 $q_{1j}, q_{1l} \in Q_1$.

- The respective reset states q_{00}, q_{10} are corresponding states.
- 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.

3 Verification Method

The above theorem, therefore, suggests a verification method 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 $P_0 = \{p_{00}, p_{01}, \cdots, p_{0k}\}$.
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.

Owing to the presence of loops it is difficult to find a path cover of the whole computation comprising only finite paths. So 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 [1, 5]. We choose the cutpoints in any FSMD as follows.

1. The reset state is chosen.
2. Any state with more than one outward transitions is also chosen.

Obviously, cutpoints chosen by the above 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 §2).

In the following we propose one method which combines the first two steps listed above 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. An initial set of cutpoints is chosen for M_0 as described above. The reset state of M_0 is always a cutpoint of M_0 and the reset states of M_0 and M_1 is the initial pair of *corresponding states*. Starting from a corresponding state q_{0i} of M_0 the algorithm traverses *all* the paths leading out of q_{0i} to the next cutpoint in M_0 and for each path it tries to find a corresponding equivalent path in M_1. On success the end points of the two paths may be recorded as a new pair of corresponding points. Otherwise, the path in M_0 is extended in all possible ways (without re-entering loops) and again matching paths are sought in M_1 for each extension. When all possibilities are exhausted without finding a match the algorithm reports a failure. The algorithm continues until all pairs of corresponding points are processed. The following pseudocode describes this process more precisely.

3.1 Verification Algorithm

Step 1: Insert cutpoints in M_0 by the following rules.

- the start state is a cutpoint,
- any state with more than one outward transition is a cutpoint.

Step 2:

/*Main data stores:
  η: Set of corresponding nodes
  P_0: path cover of M_0
  P_1^0: paths in M_1 with matching paths in P_0
Working data stores:
  F: list of paths of M_0 starting with nodes having corresponding nodes but ending with nodes whose corresponding nodes have not yet been found
  P: Working list of corresponding nodes from which paths will be examined */
F := [ ] ; P_0 := [ ] ; $P_1{}^0$:= [ ] ;
$\eta := \{\langle q_{00}, q_{10}\rangle\}$;
P := $\{\langle q_{00}, q_{10}\rangle\}$;
while (P is not empty || F is not empty)
{ // main loop continues till termination
  if (F is empty)
  // new paths starting from entries in P to be examined
  { $\langle q_{0i}, q_{1j}\rangle$:= deQ P ;
    put in F all the paths from q_{0i} to its successor cutpoints (in M_0) ;
  } else // now work on the un-matched path frontier

IEEE COMPUTER SOCIETY

```
{ β := deQ F ; // endPtNd ( β ) is un-matched!
  if ( ( α = findEquivalentPath ( β , q_1j ) ) != NULL )
  { if ( ! ⟨ endPtNd(β), endPtNd(α)⟩ ∈ η)
      enQ ( P, ⟨ endPtNd(β), endPtNd(α) ⟩ );
      // new paths will start from here
    η := η ∪ {⟨ endPtNd(β), endPtNd(α) ⟩ ;
    P_0 := P_0 ∪ {β} ; P_1^0 := P_1 ∪ {α} ;
  } else // no match
  { // so continue along all paths through successors
    if ( the path is marked NOT_EXTENDIBLE ) fail ;
    tF := all the paths obtained by concatenating to β
    all the paths from endPtNd (β) to all the successor
    cutpoints of endPtNd ( β ) ;
    if ( endPtNd of any member of tF is a node of the
      same path other than its start node )
      fail;
    if ( endPtNd of any member of tF is same
      as its start node || the reset state)
      mark the path as NOT_EXTENDIBLE ;
    F := append ( tF, F ) ;
  } // else-if
 } // else-if
} // end while
```

Step 3: Identify the cutpoints in M_1

Step 4: Repeat the same procedure as described in Step 2 with the roles of M_0 and M_1 interchanged.

Step 5: If it succeeds for both Step 2 and Step 4 then report M_0 *and* M_1 *are computationally equivalent*. Otherwise report a failure.

The functions used are specified as follows.

- $\mathtt{f}indEquivalentPath(\beta, q_{1j})$: It tries to find a path α in M_1 so that $R_\alpha = R_\beta$ and $r_\alpha = r_\beta$. If such an α exist then this function returns α, otherwise a NULL path.

- $\mathtt{e}ndPtNd(\beta)$: returns the state where the path β terminates.

4 Algorithm Complexity

The complexity of the algorithm is determined in step 2. Let there be up to n control states (nodes) in M_0 or M_1. This is an upper bound on the number of cutpoints in M_0 or M_1. The complexity of the function $\mathtt{f}indEquivalent(\beta, q_{1j})$ is proportional to the number of paths from q_{1j} to be examined, for a given β. If there are up to k parallel edges between any two states, then up to $O(kn)$ paths of length $O(n)$ need to be examined. We avoid the k^n possibilities of branching here because the paths are examined with respect to β; only the k possibilities at each node of the path need to be examined for only a single viable choice. Thus the time complexity of $\mathtt{f}indEquivalent$ is $O(kn^2)$. This is not a tight upper bound, so we also consider the lower bound which is $\Omega(1)$. This represents the case when we obtain straight matches without any exploration.

While finding the equivalent paths in M_1 corresponding to those in M_0, in the best case we only need to consider a path from one cutpoint to the next. However, path extensions may be required. The maximum length of any path in M_0 after repeated path extensions is $O(n)$. Starting from a cutpoint, considering repeated path extensions and an edge multiplicity of k, up to $O(k^{n-1})$, if $k \neq 1$ or $O(n^2)$, if $k = 1$, paths of M_0 may need to be examined. Note that if $k \neq 1$, then along a path of n nodes there are k path segments of length one, k^2 path segments of length two, etc. The total number of such paths is $k + k^2 + k^3 + \ldots + k^{n-1} = k\frac{k^{n-1}-1}{k-1}$, which is $O(k^{n-1})$. Since paths may start from any cutpoint, $O(nk^{n-1})$, if $k \neq 1$ or $O(n^3)$, if $k = 1$, paths of M_0 may need to be examined. The total number of pairs of the form $\langle\beta, q_{1j}\rangle$ that need to be examined is $O(n^2k^{n-1})$, if $k \neq 1$ or $O(n^4)$, if $k = 1$. Thus, the overall worst case complexity of step 2 is $O(n^4k^n)$ if $k \neq 1$ or $O(n^6)$, if $k = 1$ and $\Omega(n)$. In practice this is often not hit.

5 An Example

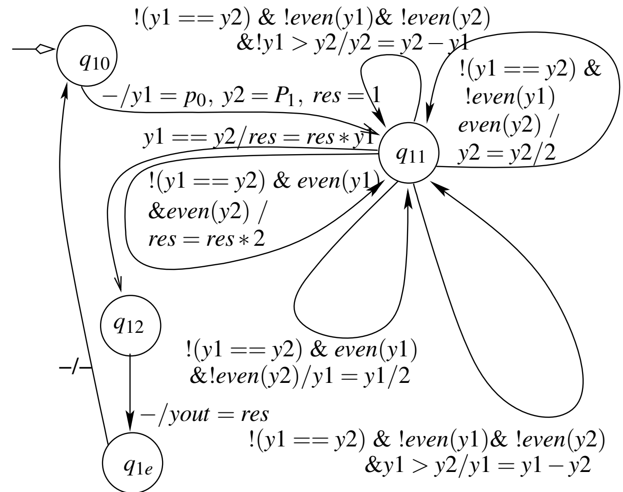

Figure 2. M_1**, the FSMD of GCD after scheduling**

The flow of step 2 of the proposed algorithm applied to the GCD example (fig 1 and fig 2) is briefly discussed here. The algorithm first considers the path $\langle q_{00} \rightarrow q_{01}\rangle$. The `findEquivalentPath` function successfully finds an equivalent path which is $\langle q_{10} \rightarrow q_{11}\rangle$. So, $\langle q_{01},\ q_{11}\rangle$ are the corresponding pair of states. The path $\langle q_{01} \rightarrow q_{0e} \rightarrow q_{00}\rangle$ has to be considered next. Its equivalent path is $\langle q_{11} \rightarrow$

0-7695-2533-4/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

$q_{12} \rightarrow q_{1e} \rightarrow q_{10}\rangle$. For the path $\langle q_{01} \rightarrow q_{02}\rangle$, there is no equivalent path in M_1. So, this path needs to be concatenated with its successor paths. These concatenated paths are $\beta_1 = \langle q_{01} \rightarrow q_{02} \rightarrow q_{03}\rangle$ and $\beta_2 = \langle q_{01} \rightarrow q_{02} \rightarrow q_{04}\rangle\}$. β_1 is again concatenated with its successor paths as it has no equivalent path in M_1. The concatenated paths have the same sequence of states $(q_{01} \rightarrow q_{02} \rightarrow q_{03} \rightarrow q_{01})$ with different conditions of execution and data transformations. Similarly, β_2 is also concatenated with its successor paths. These concatenated paths are $\beta_{21} = \langle q_{01} \rightarrow q_{02} \rightarrow q_{04} \rightarrow q_{05}\rangle\}$ and $\beta_{22} = \langle q_{01} \rightarrow q_{02} \rightarrow q_{04} \rightarrow q_{01}\rangle\}$. Path β_{21} will be concatenated with two of its successor paths next. Now each of these concatenated paths has an equivalent path in M_1. These will be explored one by one in the following iterations. It may be noted that Step 2 takes only 18 iterations which is much better than the worst case complexity of $6^6 = 46656$ (here $k = 1, n = 6$).

6 Experimental Results

The proposed algorithm has been implemented in 'C' and has been run for some standard high-level synthesis benchmarks as shown in table 1. These have been run on an Intel Pentium 4, 1.70 MHz, 256MB RAM machine. The number of states, number of paths explored in each FSMD M_0 and M_1, number of consecutive path segments merged by the scheduler and the CPU time are tabulated for each benchmark example. It is evident from table that execution time is sensitive on number of paths explored. It also may be noted from the table that run time of this algorithm is less sensitive on the number of states in the FSMDs. For example, in table 1, the run times of EWF and DCT are small compared to GCD and MODN even though EWF and DCT have greater number of states. These examples also suggest that the upper bound is not necessarily hit for practical scheduling verification cases.

Name	*#state in FSMD*		*#path in cover*		#path extn	*CPU time* in ms
	M_0	M_1	M_0	M_1		
DIFFEQ	4	12	3	3	0	2.442
EWF	4	35	1	1	0	1.820
GCD	7	4	11	7	3	3.976
DCT	3	29	1	1	0	1.754
TLC	7	8	13	14	2	4.196
MODN	6	7	8	12	2	4.324
PERFECT	9	6	7	5	2	4.028

Table 1. Results for different high-level synthesis benchmarks

7 Conclusions

Advances in VLSI technology have enabled its deployment into complex circuits. Synthesis flow of such circuits comprises various phases where each phase performs the task algorithmically providing for ingenious interventions of experts. The gap between the original behaviour 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. The present work concerns itself with the validation of the scheduling phase. Both the behaviours prior to and after scheduling have been modeled as FSMDs. The validation task has been treated as an equivalence problem of FSMDs.

The method presented in this paper has been proved to be sound, completeness being ruled out by the fact that the equivalence problem of FSMDs has been reported to be not even partially decidable. The method is strong enough to accommodate merging of the segments in the original behaviour by the typical scheduler such as, DLS [7]. It is also able to handle arithmetic transformations and expected to handle simple code motion. Similar methods reported in the literature have been found to fail under such situations. Although the proposed method is found to have a non-polynomial worst case complexity primarily because of presence of parallel edges between the same pair of states, the best case complexity is found to be linear. The initial experiments show that the algorithm is usable for practical equivalence checking cases of scheduling.

References

[1] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proceedings the 19th Symposium on Applied Mathematics*, pages 19–32, Providence, R.I., 1967. American Mathematical Society. Mathematical Aspects of Computer Science.

[2] D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE transactions on Design and Test of Computers*, pages 44–54, 1994.

[3] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[4] W. E. Howden. *Functional program testing and analysis*. McGraw-Hill, New York, 1987.

[5] J. C. King. Program correctness: On inductive assertion methods. *IEEE Trans. on Software Engineering*, SE-6(5):465–479, 1980.

[6] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Kogakusha, Tokyo, 1974.

[7] M. Rahmouni and A. A. Jerraya. Formulation and evaluation of scheduling techniques for control flow graphs. In *Proceedings of EuroDAC'95*, pages 386–391, Brighton, 18-22 September 1995.

[8] D. Sarkar and S. C. De Sarkar. Some inference rules for integer arithmetic for verification of flowchart programs on integers. *IEEE Trans. Softw. Eng.*, 15(1):1–9, 1989.

0-7695-2533-4/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY