A Formal Verification Method 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 the finite state machine with data path (FSMD) model of the high-level behavioural specification and the FSMD model of the behaviour transformed by the scheduler. The method consists in introducing cutpoints in one FSMD, visualizing its computations as concatenation of paths from cutpoints to cutpoints and finally, identifying equivalent finite path segments in the other FSMD; the process is then repeated with the FSMDs interchanged. The method is strong enough to accommodate merging of the segments in the original behaviour by the typical scheduler such as DLS, a feature very common in scheduling but not captured by many works reported in the literature. It also handles arithmetic transformations.

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 inter dependent sub-tasks such as, specification, compilation, scheduling, allocation and binding. The operations in the behavioural description are assigned time steps through scheduling process. Input to the scheduling phase is control data flow graph (CDFG)[4]. While a CDFG is better suited for the scheduling algorithms, an FSMD is a more appropriate model for verification. In fact, we construct FSMD automatically from a given input CDFG to the scheduler. In the process of scheduling, operations are often moved across basic block boundaries so that an optimization may be incorporated. In general several transformations may be made to improve the performance of the design. For example, path based scheduling techniques [10] 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 behaviours before and after scheduling, as represented by FSMDs, are computationally equivalent.

Methods to verify the high-level synthesis results against the original behavioural description are still evolving [7]. A formal synthesis system called *FRESH* was built and a new technique for verification was proposed. In this method the input behaviour is described by using equational specification (ES) and a set of derivation rules is applied consecutively on the ES. An automatic verification of scheduling by using symbolic simulation of labeled segments of behavioural description has been proposed in [1]. The method described in this paper transforms the original description into one which is bisimilar with the scheduled description. In the approach proposed in [7], break-points are introduced in both the FSMDs followed by construction of the respective path sets. Each path of one set is then shown to be equivalent to some path of the other set. This approach necessitates that the path structure of the input FSMD is not disturbed by the scheduling algorithm in the sense that the respective path sets obtained from the break points are assumed to be bijective. These approaches are likely to fail for the path based scheduling algorithms [10], [6] where path structures get changed by the scheduler.

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. The method is so devised that construction of the path set P_0 of one FSMD M_0 (say), so that any computation of M_0 can be captured by concatenation of the members of P_0, construction of the path set P_1 of the other FSMD M_1 equivalent to those of P_0 proceed hand-in-hand. The verification steps of our method are illustrated using the MODN(A.B mod N) example. This is a non-trivial example where the scheduler changes the path structure of the the input FSMD significantly.

This paper is organized as follows. In section 2, the nec-

Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED'06)
0-7695-2523-7/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

essary formalism is developed and the correctness problem is encoded. The method is described in section 3. An example has been treated in section 4 in detail to illustrate the working of the algorithm. Some experimental results have been given in section 5. The paper is concluded in section 6.

2 Problem Encoding

2.1 Finite State Machine with Data Path

An FSMD (*finite state machine with data-path*) is a universal specification model, proposed by Gajski in [3], that can represent all hardware designs. The model is used in the present work with the addition of a reset state, for encoding the specification and implementation of the circuit to be verified. This reset state is also called the start state of the FSMD. The FSMD is formally described below.

Definition 1 *The FSMD is defined as an ordered tuple* $\langle Q, q_0, I, V, O, f, h \rangle$*, where*

1. $Q = \{q_0, q_1, q_2, \dots q_n\}$ *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) $U = \{x \Leftarrow e | x \in O \cup V \text{ and } e \in E\}$ *represents a set of storage or output assignments, where* $E = \{g(x, y, z, \dots) \mid x, y, z, \dots \in I \cup V\}$ *represents a set of arithmetic expressions over the set* $I \cup V$ *of input and storage variables,*
   (b) $S = \{R(a, b) | a, b \in E \text{ and } R \text{ is any arithmetic relation}\}$ *represents a set of status signals as arithmetic relations between two expressions from the set* E.

Since, state transitions and updates have been represented as functions, an FSMD model is inherently deterministic. It may be noted that we have not introduced final states in the FSMD model. This is because we assume that the systems work in an infinite outer loop.

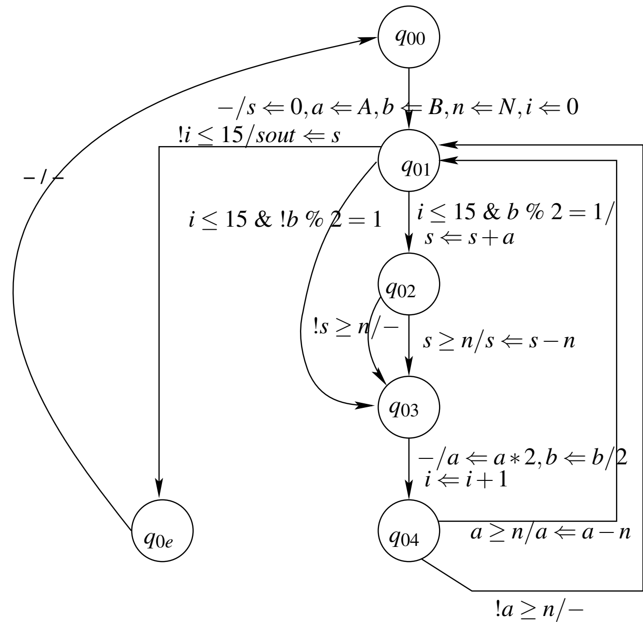

Figure 1. FSMD of MODN before scheduling

A *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 $\langle q_i = q_1 \xrightarrow[c_1]{} q_2 \xrightarrow[c_2]{}, \dots, \xrightarrow[c_{n-1}]{} q_n = q_j \rangle$ 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 path* $\alpha = \langle q_{l_0} \xrightarrow[c_0]{} q_{l_1} \xrightarrow[c_1]{} q_{l_2} \dots \xrightarrow[c_{k-1}]{} q_{l_k} \rangle$, 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 path 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 path), it is equivalent to the assignment $v \Leftarrow P_i$. The output of an expression e to a port P is represented as $OUT(P, e)$ and put as a member of a list preserved for each path. The *data transformation of a path* α over V, r_α, is the tuple $\langle s_\alpha, O_\alpha \rangle$, where s_α 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 path in terms of the initial data state (i.e., the values of the variables at the initial control state) of the path and the output list $O_\alpha = [OUT(P_{i_1}, e_1), OUT(P_{i_2}, e_2), \dots]$. For every expression e output to port P along the path α, there is an $OUT(P, e)$ in the list, in the order in which the outputs occurred.

Computation of the condition of execution R_α can be 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 execution of $y \leftarrow g(y)$,

0-7695-2523-7/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

then the predicate $c(g(y))$ must have been true before the execution of the statement [9]. The transformation s_α is found indirectly using the same principle. The forward substitution method of finding R_α is based on symbolic execution.

Let α and β be two paths and R_α, r_α be respectively the condition of the execution and the data transformation over α; let R_β, r_β be similarly defined for the path β. Then the condition of execution for the path $\alpha\beta$ ("α concatenated with β") is given by $R_\alpha(\bar{v}) \wedge R_\beta(s_\alpha(\bar{v}))$ and the data transformation over $\alpha\beta$ is $\langle s_\beta(s_\alpha(\bar{v})),\ O_\alpha\ (\bar{v})O_\beta(s_\alpha(\bar{v}))\rangle$, $\bar{v}$ represents a vector of values of the variables of $I \cup V$, $\bar{v}_f$ represents a vector of values of the variables of V. $O_\alpha(\bar{v})O_\beta(s_\alpha(\bar{v}))$ represents the concatenated output list of $O_\alpha(\bar{v})$ and $O_\beta(s_\alpha(\bar{v}))$.

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 each computation terminates. In other words, the behavioural representation has a non-terminating outermost loop from the reset state and each inner loop has a state from which there is a transition out of the loop.

Any computation c of an FSMD M can be looked upon as a computation along some concatenated path $[\alpha_1\alpha_2\alpha_3...\alpha_k]$ of M such that the path α_1 emanates from and the path α_k terminates in the reset state q_0 of M and $\alpha_i, 1 \leq i \leq k$, may not all be distinct. If $R_{\alpha_i}(\bar{v}_i)$, $r_{\alpha_i}(\bar{v}_i)$, $1 \leq i \leq k$, be the condition of execution and the data transformation respectively of the path α_i, then the condition of execution (R_c) and data transformation (r_c) of c are given by $R_{\alpha_1}(\bar{v}_1) \wedge R_{\alpha_2}(s_{\alpha_1}(\bar{v}_1)) \wedge \ldots \wedge R_{\alpha_k}(s_{\alpha_{k-1}}(\cdots(s_{\alpha_1}(\bar{v}_1))\ldots))$ and $\langle\ s_{\alpha_k}(s_{\alpha_{k-1}}(\ldots(s_{\alpha_1}(\bar{v}_1))\ldots)),\quad O_{\alpha_1}(\bar{v}_1)O_{\alpha_2}(s_{\alpha_1}(\bar{v}_1)) \ldots O_{\alpha_k}(s_{\alpha_{k-1}}(\ldots(s_{\alpha_1}(\bar{v}_1))\ldots)\ \rangle$, where $\bar{v}_i$ represents the vector of inputs and the data variables before the path α_i, $1 \leq i \leq k$.

Definition 2 *Two computations c_1 and c_2 of an FSMD are said to be equivalent if $R_{c_1} = R_{c_2}$, $r_{c_1} = r_{c_2}$.*

Computational equivalence of two paths can be defined in a similar manner. The fact that a path p_1 is computationally equivalent to p_2 is denoted as $p_1 \simeq p_2$.

Equivalence checking of paths, therefore, consists in establishing the computational equivalence of the respective conditions of execution and the respective data transformations. Since the condition of execution and the data transformation of a path involve the whole of integer arithmetic, checking of path equivalence 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 [8, 11].

Definition 3 *A finite set of paths $P = \{p_0, p_1, p_2, \ldots, p_k\}$ 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.*

2.2 Correctness Problem

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 bahaviour. 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. The following definition captures the notion of equivalence of FSMDs.

Definition 4 *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.*

From the above two definitions, following theorem can be concluded.

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 definition is used in the proposed verification method.

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 $q_{1j}, q_{1l} \in Q_1$.*

- *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:

IEEE COMPUTER SOCIETY

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.

Because 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 [2, 5, 8]. We choose the cutpoints in any FSMD as follows.

1. The reset state.

2. Any state with more than one outward transitions.

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.

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 := [ ] ;
η := {⟨q_00, q_10⟩} ;
P := {⟨q_00, q_10⟩} ;
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
  {
    ⟨q_0i, q_1j⟩ := 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
    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.

- findEquivalentPath(β, 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.

- endPtNd(β) : returns the state where the path β terminates.

0-7695-2523-7/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

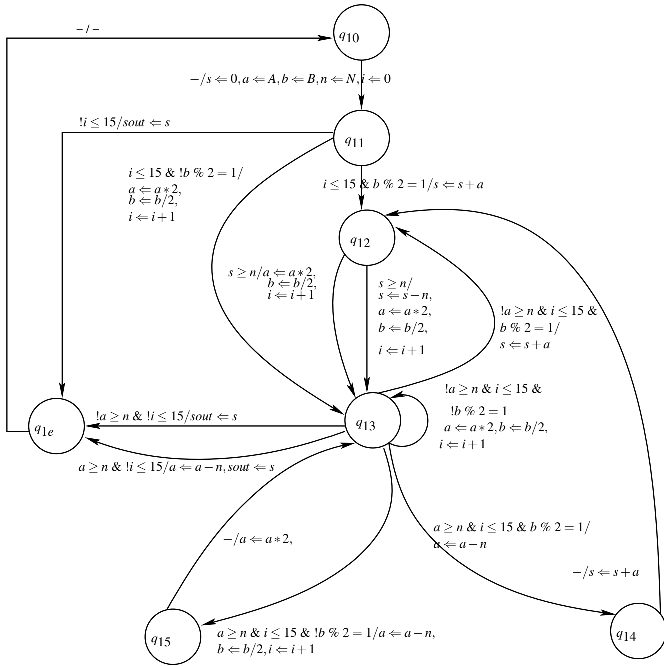

Figure 2. FSMD of MODN after DLS scheduling

4 An example

This method is explained below with MODN example.

Step 1: Cutpoints in M_0 (Fig. 1), FSMD of MODN before scheduling, are $\{q_{00}, q_{01}, q_{02}, q_{04}\}$.

Step 2: Initially, $F = [\ ], P_0 = [\ ], P_1^0 = [\ ], \eta = \{\langle q_{00}, q_{10}\rangle\}, P = \{\langle q_{00}, q_{10}\rangle\}$

Iterations:

1. $F = \{\langle q_{00},\ q_{01}\rangle\}$
2. $\beta = \langle q_{00},\ q_{01}\rangle$. Corresponding equivalent path in M_1 is $\alpha = \langle q_{10},\ q_{11}\rangle$. Put $\langle q_{01},\ q_{11}\rangle$ in η and P. Put β in P_0. Put α in P_1^0.
3. $F = \{\langle q_{01},\ q_{0e},\ q_{00}\rangle, \langle q_{01},\ q_{02}\rangle, \langle q_{01},\ q_{03},\ q_{04}\rangle\}$
4. $\beta = \langle q_{01},\ q_{0e},\ q_{00}\rangle$. Corresponding equivalent path in M_1 is $\alpha = \langle q_{11},\ q_{1e},\ q_{10}\rangle$. Put β in P_0 and put α in P_1^0.
5. $\beta = \langle q_{01},\ q_{02}\rangle$. Its equivalent path in M_1 is $\alpha = \langle q_{11},\ q_{12}\rangle$. Put $\langle q_{02},\ q_{12}\rangle$ in η and P. Put β in P_0 and put α in P_1^0.
6. $\beta = \langle q_{01},\ q_{03},\ q_{04}\rangle$. In M_1, $\alpha = \langle q_{11},\ q_{13}\rangle$ such that, $\beta \simeq \alpha$. Put $\langle q_{04},\ q_{13}\rangle$ in η and in P. Put β in P_0 and put α in P_1^0.
7. $F = \{\langle q_{02},\ q_{03},\ q_{04}\rangle, \langle q_{02},\ q_{03},\ q_{04}\rangle$.
8. $\beta = \langle q_{02},\ q_{03},\ q_{04}\rangle$. In M_1, $\alpha = \langle q_{12},\ q_{13}\rangle$ such that $\beta \simeq \alpha$. Put β in P_0 and put α in P_1^0.
9. $\beta = \langle q_{02},\ q_{03},\ q_{04}\rangle$. Corresponding equivalent path in M_1 is $\alpha = \langle q_{12},\ q_{13}\rangle$. Put β in P_0 and put α in P_1^0.
10. $F = \{\langle q_{04},\ q_{01}\rangle, \langle q_{04},\ q_{01}\rangle\}$.
11. $\beta_7 = \langle q_{04},\ q_{01}\rangle$. Function `findEquivalentPath` returns NULL.
    Thus, $tF = \{\langle q_{04},\ q_{01},\ q_{0e},\ q_{00}\rangle, \langle q_{04},\ q_{01},\ q_{02}\rangle, \langle q_{04},\ q_{01},\ q_{03},\ q_{04}\rangle\}$. Put tF in F.
12. $\beta = \langle q_{04},\ q_{01},\ q_{0e},\ q_{00}\rangle$. In M_1, $\alpha = \langle q_{13},\ q_{1e},\ q_{10}\rangle$ such that $\beta \simeq \alpha$. Put β in P_0 and put α in P_1^0.
13. $\beta = \langle q_{04},\ q_{01},\ q_{02}\rangle$. $R_\beta = \{a \geq n\ \&\ i \leq 15\ \&\ b\ \%\ 2 = 1\}$ and $r_\beta = \{a \Leftarrow a - n, s \Leftarrow s + a - n\}$. Corresponding equivalent path in M_1 is $\alpha = \langle q_{13},\ q_{14},\ q_{12}\rangle$. Put β in P_0 and put α in P_1^0.
14. $\beta = \langle q_{04},\ q_{01},\ q_{03},\ q_{04}\rangle$. $R_\beta = \{a \geq n\ \&\ i \leq 15\ \&\ !(b\ \%\ 2) = 1\}$ and $r_\beta = \{a \Leftarrow a - n, a \Leftarrow (a-n)*2,\ b \Leftarrow b/2,\ i \Leftarrow i+1\}$. In M_1, $\alpha = \langle q_{13},\ q_{15},\ q_{13}\rangle$ such that $\beta \simeq \alpha$. Put β in P_0 and put α in P_1^0.
15. $\beta = \langle q_{04},\ q_{01}\rangle$. Function `findEquivalentPath` returns NULL. Thus, $tF = \{\langle q_{04},\ q_{01},\ q_{0e},\ q_{00}\rangle, \langle q_{04},\ q_{01},\ q_{02}\rangle, \langle q_{04},\ q_{01},\ q_{03},\ q_{04}\rangle\}$. Put tF in F.

Each of these 3 extended paths has an equivalent path in M_1. They will be explored in the following iterations.

So, step 2 terminates successfully generating a path cover P_0 of M_0. Every path of P_0 has an equivalent path in P_1^0 corresponding to M_1. In iterations 11 and 15 paths had to be extended before their equivalent paths could be found.

Step 3: The cutpoints in M_1 are $\{q_{10}, q_{11}, q_{12}, q_{13}\}$.

Step 4: Iterations of steps 4 can be shown fashion that of step 2 and are not shown here for brevity.

Step 5: So M_0 and M_1 are computationally equivalent.

5 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

0-7695-2523-7/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

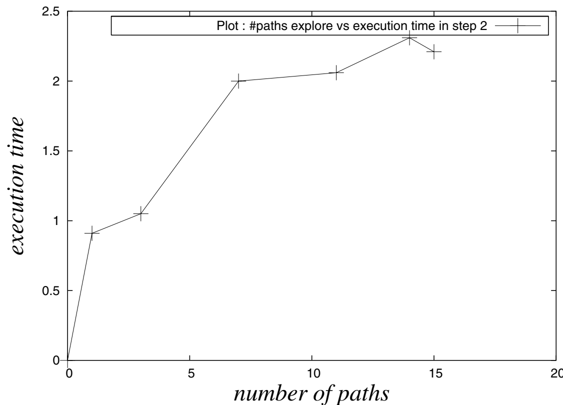

Figure 3. No. of paths explored in M0 Vs executions time of step 2

by the scheduler and the CPU time are tabulated for each benchmark example. The number of paths explored vs execution time have been plotted in Fig. 3. It is clear from this figure that execution time is sensitive to the number of paths explored. In contrast, 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.

6 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 is strong enough to accommodate merging of the segments in the original behaviour by the typical scheduler such as, DLS [10]. 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. The initial experiments show that the algorithm is usable for practical equivalence checking cases of scheduling.

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

References

[1] H. Eveking, H. Hinrichsen, and G. Ritter. Automatic verification of scheduling results in high-level synthesis. In *Proc. Conf. Design, Automation and Test in Europe 1999*, pages 59–64, March 1998.

[2] 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.

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

[4] 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.

[5] C. A. R. Hoare. An axiomatic basis of computer programming. *Commun. ACM*, pages 576–580, 1969.

[6] R. Jain, A. Majumdar, A. Sharma, and H. Wang. Empirical evalution of some high-level synthesis scheduling heuristics. In *Procs. of 28th DAC*, pages 210–215, 1991.

[7] Y. Kim, S. Kopuri, and N. Mansouri. Automated formal verification of scheduling process using finite state machine with datapath (FSMD). In *5th International Symposium on Quality Electronic Design (ISQED'04)*, pages 110–115, Carlifornia, March 2004.

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

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

[10] 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.

[11] 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.

0-7695-2523-7/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY