This file has been cleaned of potential threats.

If you confirm that the file is coming from a trusted source, you can send the following SHA-256 hash value to your admin for the original file.

605a5af7e4719e1ed994a18cd91dfeef50797b1747c776f22124437ab5fb2cb6

To view the reconstructed contents, please SCROLL DOWN to next page.
A Formal Verification Method of Scheduling in High-level Synthesis

C Karfa  C Mandal  D Sarkar  S R Pentakota
Department of Computer Se & 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 interdependent 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 input FSMD significantly.

This paper is organized as follows. In section 2, the nec-
necessary 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. \(Q = \{q_0, q_1, q_2, \ldots, q_n\}\) 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 \(\Sigma_I\) is the input alphabet,
4. \(V\) is the set of storage variables and \(\Sigma\) is the set of all data storage states or simply, data states,
5. \(O\) is the set of primary output signals and \(\Sigma_O\) 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, where \(E = \{g(x, y, z, \ldots) | x, y, z, \ldots \in I \cup V\}\) represents a set of arithmetic expressions over the set \(I \cup V\) of input and storage variables,

(b) \(S = \{R(a, b) | a, b \in E\) and \(R\) 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 \(\alpha\) from \(q_i\) to \(q_j\), where \(q_i, q_j \in Q\), is a finite transition sequence of states of the form \(\langle q_1 = q_1, q_2 = q_2, \ldots, q_n = q_n, q_{n+1}\rangle\) such that \(\forall i, 1 \leq i \leq n - 1, 3c_i \in S\) such that \(f(q_i, c_i) = q_{i+1}\), and \(q_{n+1}, 1 \leq k \leq n - 1\), are all distinct. The state \(q_n\) may be identical to \(q_1\).

The condition of execution of the path \(\alpha = \langle q_0 \tau_0 q_1 \tau_1 q_2 \tau_2 \ldots \tau_n q_n\rangle\), \(R_\alpha\), is a logical expression over the variables in \(V\) such that \(R_\alpha\) is satisfied by the (initial) data state at \(q_0\) iff the path \(\alpha\) is traversed.

We assume that inputs and outputs occur through named ports. The \(i^{th}\) input from port \(P\) is a value represented as \(P_i\). Thus if some variable \(v\) stores input from port \(P\) (for the \(i^{th}\) 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** \(\alpha\) over \(V\), \(r_\alpha\), is the tuple \(\langle s_\alpha, O_\alpha\rangle\), where \(s_\alpha\) 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 \(\alpha\) 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 \(O_\alpha = [OUT(P_{i_1}, c_{i_1}),\ldots, OUT(P_{i_m}, c_{i_m})]\). For every expression \(e\) output to port \(P\) along the path \(\alpha\), there is an \(OUT(P, e)\) in the list, in the order in which the outputs occurred.

Computation of the condition of execution \(R_\alpha\) 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 \(y \leftarrow g(y),\)
then the predicate \( c(g(y)) \) must have been true before the execution of the statement [9]. The transformation \( s_\alpha \) is found indirectly using the same principle. The forward substitution method of finding \( R_\alpha \) is based on symbolic execution.

Let \( \alpha \) and \( \beta \) be two paths and \( R_\alpha \), \( r_\alpha \) be respectively the condition of the execution and the data transformation over \( \alpha \); let \( R_\beta \), \( r_\beta \) be similarly defined for the path \( \beta \). Then the condition of execution for the path \( \alpha \beta \) ("\( \alpha \) concatenated with \( \beta \)) is given by \( R_\alpha (\mathcal{V}) \land R_\beta (s_\alpha (\mathcal{V})) \) and the data transformation over \( \alpha \beta \) is \( (s_\beta (s_\alpha (\mathcal{V})), O_\alpha (\mathcal{V})O_\beta (s_\alpha (\mathcal{V}))) \). \( \mathcal{V} \) represents a vector of values of the variables of \( I \cup V \), \( r_\beta \) represents a vector of values of the variables of \( V \). \( O_\alpha (\mathcal{V})O_\beta (s_\alpha (\mathcal{V})) \) represents the concatenated output list of \( O_\alpha (\mathcal{V}) \) and \( O_\beta (s_\alpha (\mathcal{V})) \).

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 \( \alpha_1\alpha_2\alpha_3\ldots\alpha_k \) of \( M \) such that the path \( \alpha_1 \) emanates from and the path \( \alpha_k \) terminates in the reset state \( q_0 \) of \( M \) and \( \alpha_i, 1 \leq i \leq k \), may not all be distinct. If \( R_{\alpha_i}(\mathcal{V}_i), r_{\alpha_i}(\mathcal{V}_i), \ 1 \leq i \leq k \), be the condition of execution and the data transformation respectively of the path \( \alpha_i \), then the condition of execution \( (R_c) \) and data transformation \( (r_c) \) of \( c \) are given by \( R_{\alpha_1}(\mathcal{V}_1) \land R_{\alpha_2}(s_{\alpha_1}(\mathcal{V}_1)) \land \ldots \land R_{\alpha_k}(s_{\alpha_{k-1}}(\ldots(s_{\alpha_1}(\mathcal{V}_1)) \ldots)) \) and \( (s_{\alpha_1}(s_{\alpha_2}(\ldots(s_{\alpha_k}(\mathcal{V}_1)) \ldots)), O_{\alpha_1}(\mathcal{V}_1)O_{\alpha_2}(s_{\alpha_1}(\mathcal{V}_1)) \ldots O_{\alpha_k}(s_{\alpha_{k-1}}(\ldots(s_{\alpha_1}(\mathcal{V}_1)) \ldots)) \), where \( \mathcal{V}_i \) represents the vector of inputs and the data variables before the path \( \alpha_i, 1 \leq i \leq k \).

**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 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. 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 \( R_0 = \{p_0, p_1, \ldots, p_0\} \) of \( M_0 \) for which there exists a set \( \{p_1^0, p_1^1, \ldots, p_1^j\} \) of paths of \( M_1 \) such that \( p_0 \simeq p_1^i, 0 \leq i \leq l \) and vice-versa.

The following definition is used in the proposed verification method.

**Definition 5** Corresponding states: Let \( M_0 = \{Q_0, q_00, I, V_0, O, f_0, h_0\} \) and \( M_1 = \{Q_1, q_10, I, V_1, O, f_1, h_1\} \) be the two FSMDs having identical input and output sets, \( I \) and \( O \), respectively, and \( q_{0k}, q_{1k} \in Q_0 \) and \( q_{1j}, q_{1j} \in Q_1 \).

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

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:**

```markdown
/*Main data stores:
 η: Set of corresponding nodes
 P₀: path cover of M₀
 P₁: paths in η₁ with matching paths in P₀
 Working data stores:
 F: list of paths of M₀ 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 := [ ]; P₀ := [ ]; P₁ := [ ];
 η := { ⟨q₀₀, q₁₀⟩ }; P := { ⟨q₀₀, q₁₀⟩ };
```

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
  { (q₀₀, q₁₀) := deQ P;
    put in F all the paths from q₀₀ to its successor cutpoints (in M₀);
  } else
  { // now work on the un-matched path frontier
    β := deQ F; // endPtNd ( β ) is un-matched!
    if ( ( β = findEquivalentPath ( β, q₁₀ ) ) != 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) ) };
        P₀ := P₀ ∪ { β }; P₁₀ := P₁ ∪ { 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 $M₁$

**Step 4:** Repeat the same procedure as described in Step 2 with the roles of $M_0$ and $M₁$ interchanged.

**Step 5:** If it succeeds for both Step 2 and Step 4 then report $M₀$ and $M₁$ are computationally equivalent. Otherwise report a failure.

The functions used are specified as follows.

- `findEquivalentPath(β, q₁₀)`: It tries to find a path $α$ in $M₁$ so that $R_α = R_β$ and $r_α = r_β$. If such an $α$ exist then this function returns $α$, otherwise a NULL path.

- `endPtNd(β)`: returns the state where the path $β$ terminates.
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, $F = [\ ]$, $P_0 = [\ ]$, $P_1 = [\ ]$, $\eta = \{\{q_{00}, q_{10}\}\}$, $P = \{\{q_{00}, q_{10}\}\}$

Iterations:

1. $F = \{\{q_{00}, q_{10}\}\}$
2. $\beta = \{q_{00}, q_{01}\}$. Corresponding equivalent path in $M_1$ is $\alpha = \{q_{10}, q_{11}\}$. Put $\{q_{01}, q_{11}\}$ in $\eta$ and $P$. Put $\beta$ in $P_0$. Put $\alpha$ in $P_1$.
3. $F = \{\{q_{01}, q_{02}\}, \{q_{01}, q_{02}\}, \{q_{01}, q_{02}, q_{03}\}\}$
4. $\beta = \{q_{01}, q_{02}\}$. Corresponding equivalent path in $M_1$ is $\alpha = \{q_{11}, q_{12}\}$. Put $\beta$ in $P_0$ and put $\alpha$ in $P_1$.
5. $\beta = \{q_{02}, q_{02}\}$. Its equivalent path in $M_1$ is $\alpha = \{q_{11}, q_{12}\}$. Put $\{q_{02}, q_{12}\}$ in $\eta$ and $P$. Put $\beta$ in $P_0$ and put $\alpha$ in $P_1$.
6. $\beta = \{q_{03}, q_{04}\}$. In $M_1$, $\alpha = \{q_{11}, q_{13}\}$ such that $\beta \simeq \alpha$. Put $\beta$ in $P_0$ and put $\alpha$ in $P_1$.
7. $F = \{\{q_{02}, q_{03}, q_{04}\}, \{q_{02}, q_{03}, q_{04}\}\}$
8. $\beta = \{q_{02}, q_{03}, q_{04}\}$. In $M_1$, $\alpha = \{q_{12}, q_{13}\}$ such that $\beta \simeq \alpha$. Put $\beta$ in $P_0$ and put $\alpha$ in $P_1$.
9. $\beta = \{q_{02}, q_{03}, q_{04}\}$. Corresponding equivalent path in $M_1$ is $\alpha = \{q_{12}, q_{13}\}$. Put $\beta$ in $P_0$ and put $\alpha$ in $P_1$.
10. $F = \{\{q_{04}, q_{01}\}, \{q_{04}, q_{01}\}\}$.
11. $\beta = \{q_{04}, q_{01}\}$. Function $\text{findEquivalentPath}$ returns NULL. Thus, $tF = \{\{q_{04}, q_{01}\}\}$. Put $tF$ in $F$.
12. $\beta = \{q_{04}, q_{01}\}$. In $M_1$, $\alpha = \{q_{13}, q_{14}, q_{15}\}$ such that $\beta \simeq \alpha$. Put $\beta$ in $P_0$ and put $\alpha$ in $P_1$.
13. $\beta = \{q_{04}, q_{01}, q_{01}\}$. $R_0 = \{a \geq n \& i \leq 15 \& b \% 2 = 1\}$ and $R_0 = \{a \Rightarrow a - n, s \Rightarrow s + a - n\}$. Corresponding equivalent path in $M_1$ is $\alpha = \{q_{13}, q_{14}, q_{15}\}$. Put $\beta$ in $P_0$ and put $\alpha$ in $P_1$.
14. $\beta = \{q_{04}, q_{01}, q_{03}, q_{04}\}$. $R_0 = \{a \geq n \& i \leq 15 \& (b \% 2) = 1\}$ and $R_0 = \{a \Rightarrow a - n, a \Rightarrow (a - n) / 2, b \Rightarrow b / 2, i \Rightarrow i + 1\}$. In $M_1$, $\alpha = \{q_{13}, q_{15}, q_{13}\}$ such that $\beta \simeq \alpha$. Put $\beta$ in $P_0$ and put $\alpha$ in $P_1$.
15. $\beta = \{q_{04}, q_{01}\}$. Function $\text{findEquivalentPath}$ returns NULL. Thus, $tF = \{\{q_{04}, q_{01}, q_{01}\}\}$. 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 $P_1$ 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
Fig. 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 may also 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>#state in FSMD</th>
<th>#path in cover</th>
<th>#path extrn</th>
<th>CPU time in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIFFEQ</td>
<td>4</td>
<td>12</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>EWF</td>
<td>4</td>
<td>35</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GCD</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>DCT</td>
<td>3</td>
<td>29</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TLC</td>
<td>7</td>
<td>8</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>MODN</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>PERFECT</td>
<td>9</td>
<td>6</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1. Results for different high-level synthesis benchmarks

References