An Efficient Algorithm For Scheduling Verification

Chandan Karfa[1], Chittaranjan Mandal[1], Dipankar Sarkar[1], Chris Reade[2], S R Pentakota[3]

Dept of Computer Sc & Engg, IIT Kharagpur, WB 721302, {ckarfa, chitta, ds}@iitkgp.ac.in
[2]Kingston Business School, Kingston University, UK, Chris.Reade@kingston.ac.uk
[3]Texas Instruments, Bangalore, INDIA, satya@ti.com

Abstract—Advances in the VLSI field have been a major driving force behind the information revolution being witness today. In this paper we describe 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 datapaths (FSMD). Hence, the checking is between FSMDs. 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 arithmatic transformations.

I. 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 the scheduling process. Input to the scheduling phase is a control data flow graph (CDFG)[1]. 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 [2], 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.

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 II, FSMDs and the notions of computations on FSMDs and the equivalence of FSMDs are defined. The verification method is described in section III. Some experimental results have been given in section IV. The paper is concluded in section V.

II. FSMDs and their Equivalence

A. Finite state machines with data paths

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 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 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, from variables (storage or output) or expressions constructed over (input or storage) variables. Thus, $E = \{g(x,y,z,\ldots) \,|x,y,z,\ldots \in I \cup V\}$ represents a set of arithmetic expressions over the set $I \cup V$.

(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.

Since, state transitions and updates have been represented as functions, an FSMD model is inherently deterministic.

B. 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 $\langle q_i = q_1 \xrightarrow{c_1} q_2 \xrightarrow{c_2} , \cdots, \xrightarrow{c_{n-1}} q_n = q_j\rangle$ 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. In the rest of the paper a (finite) walk $\langle q_1 \xrightarrow{c_1} q_2 \xrightarrow{c_2} \cdots \xrightarrow{c_{n-1}} q_n\rangle$ is also represented as $\langle q_1 \Rightarrow q_n\rangle$ for brevity whenever it is possible to do so without ambiguity. 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 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 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.

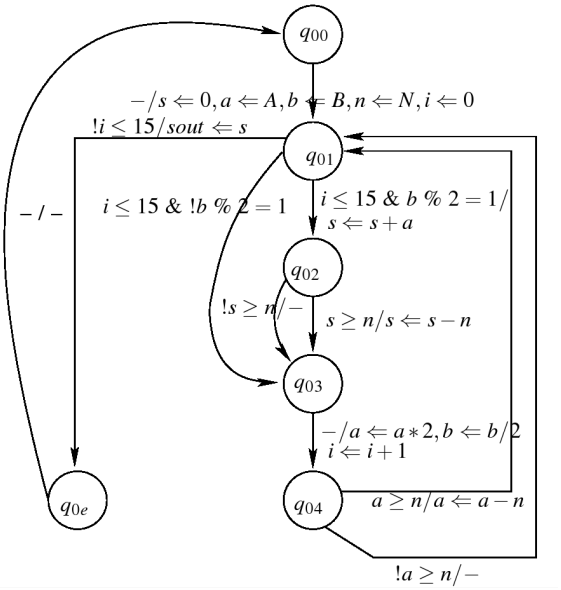

Fig. 1
FSMD OF A*B MOD N BEFORE SCHEDULING

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 [4]. The transformation s_α is found indirectly using the same principle. The forward substitution method of finding R_α is based on symbolic execution.

C. 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.

D. 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. It means that if R_c is satisfied by $\overline{v}_i$, then the computation c takes place and after completion of c the data state is given by $s_c(\overline{v}_i)$ and the sequence of outputs is given by $O_c(\overline{v}_i)$.

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.

E. 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

[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.

exists a computation c_1 of M_1 such that c_0 and c_1 are computationally equivalent and vice-versa.

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.

III. Verification Method

The above discussion 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_{ij} of M_1 such that $p_{0i} \simeq p_{1j}$.
3. Repeat steps 1 and 2 starting from M_1.

Because of loops it is difficult to find a finite set 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 [5], [6], [7]. Choice of cutpoints, however, is non-unique 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. Therefore, it will be necessary to search for a suitable choice of cutpoints. The question remains whether such a choice can be algorithmically hit upon. The *equivalence problem of FSMDs (programs) is undecidable [8]; moreover, the problem is not even partially decidable as shown for flowchart schemas [8].* Therefore, we can at best devise a good strategy for setting the cutpoints which would work for most of the cases but not for all cases. In the following we propose one such method which combines the first two steps listed above into one. More specifically, the method construct a path cover of M_0 and also finds its equivalent path set in M_1 hand-in-hand. The method for verification is as follows.

A. Verification algorithm

Step 1:
Let η, the set of corresponding states, Initially, $\{< q_{00}, q_{10} >\}$;
Insert cutpoints in M_o by the following rules.

- reset state is a cutpoint,
- any state with more than one outward transition is a cutpoint;

Let C be the set of cutpoints;
Let P_{0i} be the set of paths of M_0, where each path spans from a cutpoint to a cutpoint with no intermediary cutpoint;
Step 2:
$P'_{oi} = P_{oi}$. /* P'_{0i} is the working set */
while (P'_{oi} is not empty)
do
  β = $getPath(P'_{0i})$;
  if (β = NULL) then
    if ($anyUncovered(\eta, C, P'_{0i}, P_{0i})$) then
      Report "M_0 and M_1 may not be equivalent";
      else;
    else /* $\beta \neq NULL$ */
    do
      Let $\beta =< q_{0i} \Rightarrow q_{0f} >$ and $\langle q_{0i}, q_{1j} \rangle \in \eta$
      α = $findEquivalent(\beta, q_{1j})$;
      if α = empty then
        do
          /* Extend β of the form $(q_{0i} \Rightarrow q_{0f})$ in M_0 by moving *through cutpoint q_{0f} till some subsequent* cutpoint, but without moving through the reset state or any cutpoint more than once; search for any equivlent path in M_1. Returns a set E of ordered pairs $\langle \beta_m, \alpha_m \rangle$, where β_m's are extensions of β (in all possible ways) satisfying the above constraint and α_m's are the corresponding equivalent paths in M_1. */
          $E = \phi$;
          E = $extendEquivalent(\beta, q_{1j}, \eta, E)$;
          if ($E \neq \phi$) then
            Delete β from P'_{0i};
            $\forall r = \langle \beta_m, \alpha_m \rangle \in E$
              if ($\alpha_m \neq empty$) then
                Add β_m to P_0;
                Add α_m to P_1^0;
                $\eta = \eta \bigcup \{< endSt(\beta_m), endSt(\alpha_m) >\}$
                else *notEquivalent* (β_m); /* report. */
            else *notEquivalent* (β); /* report. */
        end do /* if α = empty */
        else /* α nonempty */
          Add β to P_0.
          Add α to P_1^0.
          Delete β from p'_{0i};
          $\eta = \eta \bigcup \{< endSt(\beta), endSt(\alpha) >\}$.
  end do /* if β = NULL - else */
end do /* P'_{oi} is not empty */
Step 3:
Identify the cutpoints in M_1;
Find P_{1i}, initial path cover of M_1;
Step 4:
Repeat the same procedure as described in Step 2 for M_1;
Step 5:

If both Step 2 and Step 4 succeed then report M_0 and M_1 are computationally equivalent.
Otherwise report a failure.

Function : extendEquivalent (β, q_{1j}, η, E)
Let β be $\langle q_{0i} \Rightarrow q_{0f} \rangle$;
$\forall\ \beta'' \in P_{0i}$ such that β'' emerges from q_{0f}.
do
  $\beta' = \beta$;
  $\beta' = concat(\beta', \beta'')$;
  $\alpha = findEquivalent(\beta', q_{1j})$;
  if α = empty then
    if (endSt(β') is node of β or endSt(β') = q_{00})
      then $E \leftarrow E \bigcup \{\langle \beta', empty \rangle\}$;
      else $E \leftarrow E \bigcup extendEquivalent(\beta', q_{1j}, \eta, E)$;
    else do
      $\eta = \eta \bigcup \{\langle endSt(\beta'), endSt(\alpha) \rangle\}$;
      $E = E \bigcup \{\langle \beta', \alpha \rangle\}$;
    end do /* if α = empty - else */
end do /* $\forall \beta'' \in P_0$ and β'' emerges from q_{0f} */
return E;

Other functions used are specified as follows.
- $getPath(P'_{0i})$: Returns a path β of the form $\langle q_{0i} \Rightarrow q_{0f} \rangle$ from the path list P'_{0i} such that q_{0i} has a corresponding state, q_{1j} say, in M_1; i.e. $\langle q_{0i}, q_{1j} \rangle \in \eta$. If such a path does not exist in P'_{0i}, then return NULL.
- *anyUncovered* : It tries to find the paths in P'_{0i} which are already covered by some paths in P_0 and need not be considered for equivalence checking. If it finds such paths then they are deleted from P'_{0i} and 'false' is returned; otherwise return 'true'.
- $findEquivalent(\beta, q_{1j})$: It tries to find a path α in M_1 starting from q_{ij} so that $R_\alpha = R_\beta$ and $r_\alpha = r_\beta$. If such an α exists, then this function returns α, otherwise returns "empty" path.
- $concat(\alpha, \beta)$: returns the concatenated path $\alpha\beta$.
- $endSt(\beta)$: returns the state where the path β terminates.

IV. Experimental Results

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 I
Results for different high-level synthesis benchmarks

The proposed algorithm has been implemented in 'C' and has been run for some standard high-level synthesis benchmarks as shown in table I. 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 I, 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.

V. 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 [2]. 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.

References

[1] 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.

[2] M. Rahmouni and A. A. Jerraya, "Formulation and evaluation of scheduling techniques for control flow graphs," in *Proceedings of EuroDAC'95*, (Brighton), pp. 386–391, 18-22 September 1995.

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

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

[5] R. W. Floyd, "Assigning meaning to programs," in *Proceedings the 19th Symposium on Applied Mathematics* (J. T. Schwartz, ed.), (Providence, R.I.), pp. 19–32, American Mathematical Society, 1967. Mathematical Aspects of Computer Science.

[6] C. A. R. Hoare, "An axiomatic basis of computer programming," *Communications ACM*, pp. 576–580, 1969.

[7] J. C. King, "Program correctness: On inductive assertion methods," *IEEE Trans. on Software Engineering*, vol. SE-6, no. 5, pp. 465–479, 1980.

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