ON THE VERIFICATION OF SEQUENTIAL MACHINES AT DIFFERING LEVELS OF ABSTRACTION

by

Srinivas Devadas, Hi Keung Ma, and
A. Richard Newton

Memorandum No. UCB/ERL M86/93

9 December 1986

ON THE VERIFICATION OF SEQUENTIAL MACHINES AT DIFFERING LEVELS OF ABSTRACTION

by

Srinivas Devadas, Hi Keung Ma, and A. Richard Newton

Memorandum No. UCB/ERL M86/93

9 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ON THE VERIFICATION OF SEQUENTIAL MACHINES AT DIFFERING LEVELS OF ABSTRACTION

by

Srinivas Devadas, Hi Keung Ma, and A. Richard Newton

Memorandum No. UCB/ERL M86/93

9 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

On The Verification of Sequential Machines At Differing Levels of Abstraction

Srinivas Devadas, Hi Keung Ma and A. Richard Newton.
Department of Electrical Engineering and Computer Sciences
550 Cory Hall
University of California, Berkeley, CA. 94720

Abstract

We present an algorithm for the verification of the equivalence of two sequential circuit descriptions at differing levels of abstraction, namely at the register-transfer (RT) level and the logic level. The descriptions represent general finite automata at the differing levels - a finite automaton can be described in a ISP-like language and its equivalence to a logic level implementation can be verified using our algorithm. Previous approaches to sequential circuit verification have been restricted to verifying relatively simple descriptions with small amounts of memory. Unlike these approaches, our technique is shown to be computationally efficient for much more complex circuits. The efficiency of our algorithm lies in the exploitation of don't care information derivable from the RTL description (e.g invalid input and output sequences) during the verification process. Using efficient cube enumeration procedures at the logic level we have been able to verify the equivalence of finite automata with a large number of states/latches in small amounts of cpu-time.

Acknowledgements

This work is supported in part by the Digital Equipment Corporation, the Semiconductor Research Corporation, and the Defense Advanced Research Projects Agency under contract N00039-86-R-0365. Their support is gratefully acknowledged.

1. INTRODUCTION

Verifying the equivalence of logic circuit descriptions at differing levels of abstraction is an extremely important problem and has many possible applications. For example, after the synthesis of a logic level finite automaton from a higher level register-transfer description, it is essential to be able to verify that the two descriptions actually represent the same machine.

One approach to the general verification problem is exhaustive simulation. Unfortunately, the number of simulations required grows exponentially with the number of inputs for even a purely combinational logic circuit, and grows even faster for sequential circuits since all possible input vector *sequences* have to simulated to prove equivalence. A different approach is to use *formal verification* techniques which are input pattern independent and can guarantee functional equivalence.

Many formal verification approaches have been taken to prove/disprove the equivalence of two combinational logic circuits, at the gate level and at differing levels[don76,rot77] [rot80,oda86]. In particular a package of programs called PROTEUS[wei86] incorporates several efficient algorithms for verifying combinational logic circuits and has successfully been used on circuits with a large number of gates.

Sequential circuit verification is a considerably more difficult problem, in the general case when there is no correspondence between the latches (states) of the two circuits. (In the special case of a one-to-one correspondence between the two circuits, the problem reduces to a combinational circuit verification problem). The few approaches taken to solve the sequential verification problem include the use of temporal logic[bro85] and PROLOG[mar85]. The use of temporal logic helps for asynchronous circuits[dil85] but is not necessary in the synchronous circuit case; algorithms have been proposed for formally verifying the equivalence of two gate level sequential circuit descriptions with differing numbers of latches using symbolic boolean

manipulation[sup86]. However because of the intractability of the problem, all the approaches taken so far have been restricted to small to medium sized circuits with a small amount of memory.

In this paper we present an algorithm for formally verifying the equivalence of two sequential machines - one described at the register-transfer level and the other at the logic level. By exploiting the don't care information available at the register-transfer level (e.g. invalid input and output sequences) we have drastically reduced the complexity of the verification problem and successfully verified the equivalence of finite automata with a large number of states/latches and gates.

Our approach involves extracting the state transition graphs (STG) of the two finite automata - the first from the register-transfer level description and the second from the gate level circuit. These two STG's are then checked for equivalence. While extracting the second STG from the logic level circuit, the use of don't care information from the first STG enables us to reduce the number of states and the number of edges in the second STG. The number of states of a finite automaton grows exponentially with the number of latches in the circuit. However, for large machines the number of states actually visited given the input sequences is typically a small fraction of the total number of possible states. This is especially true if a state assignment program[dem85] has been used in the synthesis process which minimizes combinational logic and may or may not produce a minimum bit encoding of the states. The use of invalid output sequence information and cube enumeration on the combinational logic part of the gate level finite automaton enables us to detect these invalid states (actually decode the internal state encoding) thereby reducing the complexity in checking equivalence. Given a large number of states, the number of edges in the STG may be prohibitively large. Allowing only valid input sequences enables us to reduce the number of edges in the STG, again reducing the time required to check for

equivalence. As opposed to the symbolic boolean manipulation techniques used in [sup86,oda86] for sequential circuit verification we use modifications of backward justification algorithms on the combinational logic parts to **implicitly enumerate** the input combinations.

The algorithms used in the extraction of STG's of deterministic finite automata (DFA/Moore machine) described at the gate level exploiting don't care information are described in Section 2. The extraction of the STG from the register-transfer level finite automaton and the algorithm used for formally checking the equivalence of two DFA's represented by STG's are described in Section 3. The algorithms described for DFA's (Moore machine) in Section 2 are extended to NFA's (Mealy machine) in Section 4 and results for several examples are given.

2. EXTRACTION OF MOORE MACHINE STATE TRANSITION GRAPHS

2.1. Definitions and Notation

A finite automaton whose output is associated with the state is called a Moore machine. A Moore machine is a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where Q is a finite set of states, Σ is a finite input alphabet, Δ is the output alphabet, λ is a mapping from Q to Δ giving the output associated with each state and q_0 in Q is the initial state.

A general model for the Moore machine at the logic level is shown in Figure 1. The output combinational logic block performs the Q to Δ mapping. The next state logic block generates the next state given the present state and input vector. D latches constitute the memory elements. The two combinational logic blocks will henceforth be referred to as the OL and NSL blocks respectively. The Moore machine is constructed in such a fashion that the output is only a function of the present state and not a function of the present input vector. For a Moore machine we have

$$o_i = f_i(ps_1, ps_2, \ldots, ps_{N_s}) \quad 1 \leq i \leq N_o$$

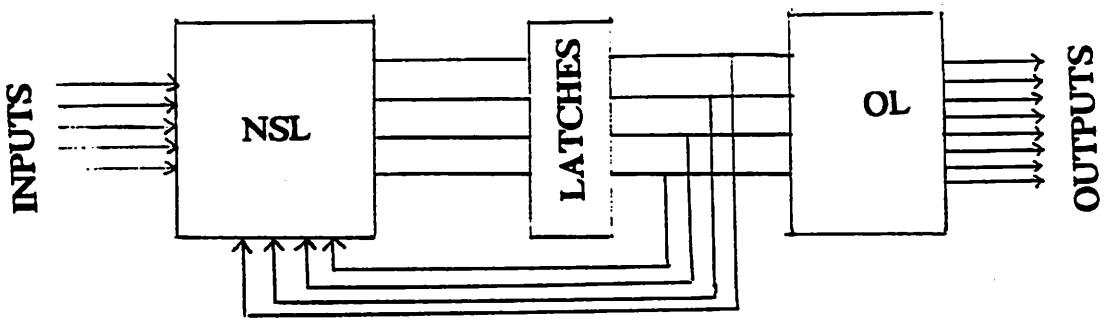

Fig. 1 General Moore Machine Model

$$ns_i = g_i(ps_1, ps_2, \ldots, ps_{N_s}, i_1, i_2, \ldots, i_{N_i}) \quad 1 \leq i \leq N_s$$

where ps_j and ns_j denote the present state and next state values respectively and i_k denotes the input.

A **cube** c for a single output function is specified by a row vector $c = [c_1, c_2, c_n]$ where c_i can take values of 1, 0 and * (don't care). n is the length of the cube. It has the usual boolean meaning associated with it[bra84].

A set of cubes $\chi = \{c^1, c^2, \ldots, c^k\}$ is said to be a **ON-set** cover for a logic function if the logic function evaluates to 0, for any cube not in χ. (The **OFF-set** of a logic function can be similarly defined).

Boolean **intersection** $\bigcap$ and **union** $\bigcup$ are two operations that can be performed between two or more sets of cubes[bra84]. For example one can find all the cubes for which two outputs of a multi-output function are 1, by intersecting the ON-sets of the

two outputs. The complement of a set of cubes χ is denoted $\bar{\chi}$.

2.2. Extraction Task

Given a logic level description of consisting of gates and latches of a machine the goal is to extract the state transition graph of the corresponding DFA. This can be done in a number of ways.

One method which has been used in this context[sup86,oda86] is *symbolic boolean simulation*[bry85] of the combinational logic blocks in the circuit which expresses each output as a algebraic function (with boolean operations) of the inputs. Given these algebraic functions, the edge transitions between two arbitrary states can be expressed as a conjunction of these functions and possibly their complements. Thus the equations corresponding to f_i and g_i can be extracted from the OL and NSL blocks of the machine and the edge transitions can be expressed as a function of the g_i and $\overline{g_i}$ and the output for each state can be found using f_i. Unfortunately, for combinational logic blocks with a large number of gates the size/length of the equations becomes prohibitively large and the extraction process becomes inefficient. Also, before including an edge in the state transition graph, it has to be checked for satisfiability which is an NP-complete operation[gar79].

Flattening the logic circuit is another alternative. Flattening involves reducing the combinational logic blocks into two level (PLA) form. Given a truth table for these combinational logic blocks the edges in the transition table can easily be found by inspection. Flattening however may require exponential cpu time and memory requirements and is not viable in many cases.

A third approach involves using backward justification algorithms to enumerate the ON and OFF sets of a logic function. Enumeration has been successfully used in the combinational logic verification problem[wei86]. An *implicit cube enumeration*

algorithm is described in detail in the following section and a method for STG extraction using this algorithm while exploiting don't care information is described in Section 2.4.

2.3. Implicit Enumeration

Justification algorithms like the D-algorithm and PODEM try to find *a* value for the inputs of a logic network given an output value (0 or 1) if such a input cube exists or prove its non-existence. The goal of enumeration is to find *all* possible values of inputs which produce the given output value. It is easy to see that justification algorithms with proper modification on the termination condition can become enumeration algorithms. The ON-set (OFF-set) of a logic function can be implicitly enumerated with a small number of cubes using enumeration algorithms based on efficient justification algorithms.

We have used an enumeration algorithm based on the PODEM[goe81] justification algorithm. In PODEM, given an output signal and a desired value on the output, a path is traced from the signal to the primary inputs (PI) to obtain a PI assignment. This PI assignment is simulated to see if the desired value of the signal has been set up. If so, the procedure terminates. If the opposite value has been set, an opposite value is assigned to the PI and this value propagated. If the signal remains unspecified, path tracing is repeated. The above procedure continues until either a successful PI assignment has been found or all the PI assignments have been exhausted.

To use PODEM for enumeration[wei86], the justification procedure is *not* terminated once a successful PI assignment is found, but only when all PI assignments have implicitly but exhaustively been enumerated. It is easy to see that PODEM provides the OFF-set (ON-set) of the output signal simultaneously while enumerating the

ON-set (OFF-set).

2.4. Extraction Using Enumeration

The inputs to the logic level extraction program are the combinational logic blocks OL and NSL and the state transition table. *STT* 1, generated from the RTL description. The output is a state transition table *STT* 2. The extraction of a state transition table from a ISP-like RTL description is described in Section 3. The input and output sequence don't care information can be derived from *STT* 1 and used in generating *STT* 2. The following steps are performed during the extraction process.

(1) The ON-set and OFF-set is found for each output of the OL and NSL blocks. using implicit cube enumeration. We denote $C_A{}^{OFF}$ ($C_A{}^{ON}$) as the OFF-set (ON-set) for a line A.

(2) If there exist N_s latches in the logic description, i.e. N_s outputs to the NSL block, the number of states which can exist in the corresponding finite automaton is 2^{N_s}. However, given a set of *valid input sequences*, some of these states may never be reachable from the starting state q_0. For example, given a 4 bit encoding of states in a 9 state finite automaton, 7 (2^4 - 9) such states exist. These states which can never be reached from q_0 can be discarded, since we wish to verify the outputs of the two machines under the valid input set alone. Finding these *invalid states* is non-trivial since the internal encoding of the states in the logic level finite automaton is not known. However, the output sequence information can be used to find some if not all of these invalid states.

The set of valid output vector cubes $VO = \{vo_1, vo_2, ..., vo_z\}$ is constructed by inspecting *STT* 1. If the two machines are equivalent, all output vectors in *STT* 2 $\in VO$. Using a logic minimizer like ESPRESSO-II[bra84] $\overline{VO}$ is found. A logic minimizer is invoked so as to reduce following computations which are proportional to the number

of literals (0 and 1) in $\overline{VO}$.

Lemma 1: If a state q produces a invalid output vector $vo \in \overline{VO}$, q is an invalid state.

Lemma 1 is the basis of finding invalid state encodings while generating $STT2$. The following section of code illustrates how invalid states are found given $\overline{VO}$ and the ON and OFF-sets of the OL block outputs.

$INVAL\_S = \phi$
foreach(cube $iv \in \overline{VO}$) {

/* find all input cubes producing this invalid output cube */

$$INVAL\_S = INVAL\_S \bigcup \{ \bigcap_{i \in I^+}^{N_s} C_i^{ON} \bigcap \bigcap_{i \in I^-}^{N_s} C_i^{OFF} \}$$

}

---

where iv is a cube of length N_s (inputs to the OL block), $I^+ = \{i : iv_i = 1\}$ and $I^- = \{i : iv_i = 0\}$. The - literals in the cube iv are ignored in this computation. As one can see the number of cube set intersections performed in this step is proportional to the number of 0 and 1 literals in $\overline{VO}$ which is why a fast logic minimization while computing $\overline{VO}$ is employed. This technique cannot find an invalid state which produces a valid output cube. However, in large examples, typically a significant number of invalid states can be found using a small fraction of the total cpu time spent in the verification process as indicated in Section 4.

(3) $INVAL\_S$ is complemented to find the set of valid states VAL_S. The set of all valid input cubes VAL_I is constructed by inspecting $STT1$. The edges in $STT2$ are generated using the NSL block enumerations. The section of code shown below illus-

trates this process.

---

foreach(state $QF \in VAL\_S$) {

/* find all inputs to NSL producing this state as output */

$$INPUT\_PS = \{ \bigcap_{i \in I^+}^{N_i+N_s} C_i^{ON} \bigcap \bigcap_{i \in I^-}^{N_i+N_s} C_i^{OFF} \}$$

foreach(cube $ip \in INPUT\_PS$) {

$input = ip\langle 0{:}N_i-1\rangle$
$QP = ip\langle N_i{:}N_s+N_i-1\rangle$

if ($QI \in VAL\_S$ AND $input \in VAL\_I$)
include edge $QI \rightarrow QF$ on $input$ in $STT2$
}
}

---

where $INPUT\_PS$ is a cube of length N_i+N_s (inputs to the NSL block), $I^+=\{i:QF_i=1\}$ and $I^-=\{i:QF_i=0\}$. Checking to see if $input \in VAL\_I$ can significantly reduce the number of edges in $STT2$. The output corresponding to each state is found by simulating the state vector on the OL block.

(4) A state s_2 in $STT2$ which produces the same output as s_1 the starting state in $STT1$ is picked as the starting state of $STT2$. All the states which cannot be reached from s_2 in $STT2$ (if any) are deleted.

Cube set intersections require time complexity $O(n^2+m^2)$ given two sets of cubes with n and m cardinality. Two things are done to speed up cube intersections during the invalid state detection and edge generation process. Firstly, intersections within each cube (iv or ip) are performed in an order of increasing cube set cardinality so the number of intersected cubes at any point is minimum. Secondly, invalid output/state cubes are grouped in such a fashion that repetition of intersections between ON/OFF-sets of the same pair of outputs is minimized, without storing more than two intermediate results. This technique cannot find an invalid state which produces a valid

output cube. However, in large examples, typically a significant number of invalid states can be found using a small fraction of the total cpu time spent in the verification process as indicated in Section 4.

3. DFA EQUIVALENCE AND EXTRACTION FROM RTL DESCRIPTIONS

3.1. Input RTL Description

The input description is at the register-transfer level, and has the following main constructs.

(1) Procedures and functions

(2) If and Select for control/branching

(3) Loops - While and For.

The description is ISP-like[bar79] except that clock boundaries are explicitly delineated using a *wait* statement on the rising/falling edge of the clock (or clock phase) ϕ_1. A sample input description is shown in Figure 2.

3.2. Extraction from RTL Description

The extraction process is only concerned with the control flow in the RTL description, we wish to generate a DFA controller for the input specification. The DFA will have the control variables as the inputs (e.g. instruction bits, ALU status bits) and assert outputs (e.g. register load, ALU add) depending on the present state.

The following steps are carried out during the DFA extraction:

(1) In the first pass, a one-to-one correspondence between the controlling input variables and output signals of the RTL description and the logic level description is made. For example, in the description shown in Figure 2, the variables run and pb are two inputs. The output signals associated which each micro-instruction are

```
MAIN()
BEGIN
 run = 1;
 WHILE run DO
  BEGIN
   fetch_instruction();
   effective_address();
   execute();
   IF interrupt.enable EQL 1 THEN
     IF interrupt.request EQL 1 THEN
    BEGIN
     MBR = PC ;
     MP[0] = MBR ;
     PC = 1;
     wait(φ1);
    END
  END
END


! subroutine for effective address calculations

ROUTINE effective_address()
BEGIN

SELECT pb FROM
 [0]: BEGIN MA = 0 @ pa; END
 [1]: BEGIN MA = last.pc<0:4> @ pa; END
ENDSELECT ;

wait(φ1);

IF ib EQL 1 THEN
BEGIN
 MA = MP[MA] ;
 wait(φ1);
END

END ! end of routine effective_address()
```

Fig. 2 Sample Input RTL Description

specified along with the RTL description. e.g. the micro-instruction MA = 0 @ pa may require (a) the load signal of the MA register be high and (b) the load signal

of the ALU be high with the ALU operation code 111.

(2) Given the inputs and outputs, the description is parsed starting from the routine MAIN and entering and exiting all procedures in the order they are called in. If a micro-instruction is encountered then the corresponding outputs of the micro-instruction are asserted in the output of the present state. If a wait(ϕ_1) statement is encountered a new state is generated.

(3) If a branch statement i.e. IF/SELECT is encountered, two or more states are generated depending on the number of branching conditions, an transition edge between the previous state and each possible present state created with the corresponding input pattern. The extraction process continues with each possible present state recursively enumerating all the possible combinations. The recursion may terminate at the end of the MAIN routine or terminate if any input condition is violated.

The state transition graph for the routine effective_address() is shown in Figure 3. Only the local inputs and outputs are shown.

3.3. Verifying the equivalence of two DFA's

Verifying that two incompletely specified finite automata are equivalent is done using a modified form of the method used to test completely specified finite automata[hop79]. Given the completely specified finite automata M_1 and M_2 accepting languages L_1 and L_2 respectively. $(L_1 \bigcap \overline{L_2}) \bigcup (\overline{L_1} \bigcap L_2)$ is accepted by some finite automaton M_3. M_3 accepts a non-empty language if and only if $L_1 \neq L_2$.

One way of handling incompletely specified DFA's[sup86] is to enter a third sequential machine which accepts the inputs M_1 and M_2 don't accept and perform an extra intersection. However, a more efficient way is to add a single dummy state in

each of $STT\,1$ and $STT\,2$ to which all the don't care transitions fanout to.

```
foreach(state q ∈ STT 1 ) {
  fanout = φ

  foreach( fanout edge E from q )
    fanout = fanout ⋃ E.input

  dcfanout = ¬fanout
  add dcfanout edges from q to dummy_s
}
```

The same is done for $STT\,2$. $STT\,1'$ and $STT\,2'$ are now completely specified DFA's and if they are equal it follows that $STT\,1$ and $STT\,2$ are equal.

A composite finite automaton $STT\,3$ given $STT\,1'$ and $STT\,2'$ is constructed which is the "multiplication" of $STT\,1'$ and $STT\,2'$.

```
foreach( edge e1 in STT 1' ) {
  foreach( edge e2 in STT 2' )
    if ( e1 ⋂ e2 ≠ φ )
      include edge {e1.From , e1.To} → {e2.From , e2.To} in STT 3

}
```

---

where *From* and *To* denote the fanout and fanin nodes of an edge. $STT\,3$ may have as many as $N_1 * N_2$ states given N_1 and N_2 states in $STT\,1'$ and $STT\,2'$ respectively. If a path exists from $\{s_1, s_2\}$ to any final node in $STT\,3$ the machines are not equivalent (if a path does not exist $STT\,1 = STT\,2$). The final nodes in $STT\,3$ are found as illustrated

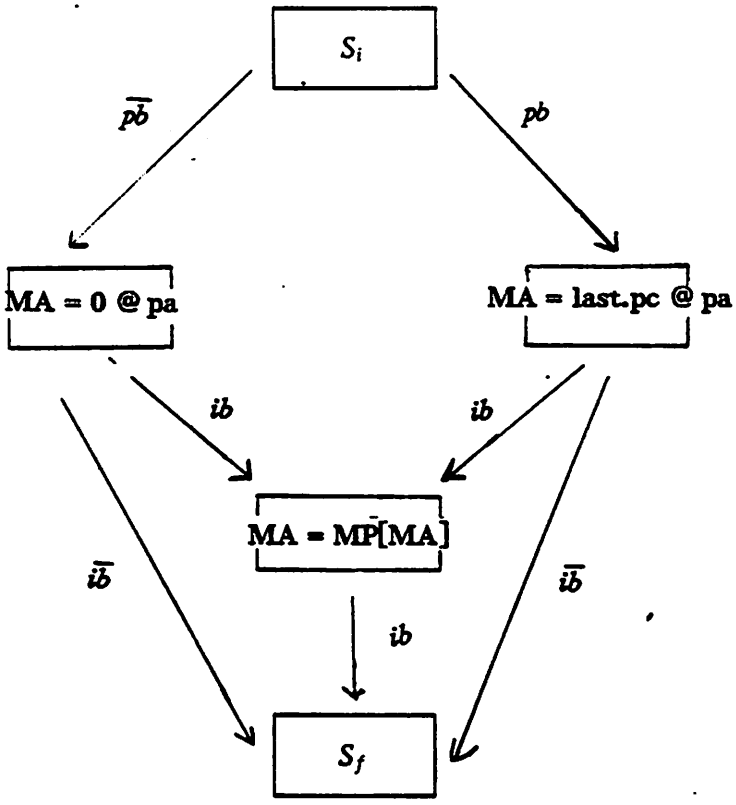

Fig. 3 State Transition Graph for effective_address()

below.

```
for( i = 1 To N_o ) {
  foreach( state {q_1 , q_2} ∈ STT 3 ) {
    if (( q_1.output_i = 1 AND q_2.output_i = 0 )
                          OR
       ( q_1.output_i = 0 AND q_2.output_i = 1 ))
      mark {q_1 , q_2} as a final node
  }
}
```

---

4. EXTENSION OF ALGORITHMS TO MEALY MACHINES, EXAMPLES AND RESULTS

4.1. Extension to NFA/Mealy Machines

A finite automaton whose output is associated both with the input and the state is called a Mealy machine. A Mealy machine is also a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where all is as in the Moore machine, except that λ maps $Q \times \Sigma$ to Δ. For a Mealy machine we have

$$o_i = f_i(ps_1, ps_2, \ldots, ps_{N_s}, i_1, i_2, \ldots i_{N_i}) \quad 1 \leqslant i \leqslant N_o$$

$$ns_i = g_i(ps_1, ps_2, \ldots, ps_{N_s}, i_1, i_2, \ldots i_{N_i}) \quad 1 \leqslant i \leqslant N_s$$

where ps_j and ns_j denote the present state and next state values respectively and i_k denotes the input.

Finding the complement of a NFA requires conversion to a DFA[hop79]. Hence to verify the equivalence of two Mealy machine STG's by constructing $(L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$ we need to convert them into Moore machine STG's. This transformation is always possible, however the resulting Moore machine will have a larger number of states and edges than the original Mealy machine. The number of extra states required is $\sum_{i=1}^{N} (D_i - 1)$ where D_i is the number of different output vec-

tors for state i, and N is the number of states in the Moore machine.

The invalid state detection process is different for a Mealy machine because f_i is now a function of both ps and i.

Lemma 2a: If i @ q always produces a invalid output vector for all $i \in VAL\_I$, then q is an invalid state.

Lemma 2b: If i @ q always produces an invalid next state for all $i \in VAL\_I$, then q is an invalid state.

where @ denotes concatenation and VAL_I is the set of valid inputs. For Lemma 2b to apply, at least one invalid state has to be detected using Lemma 2a. The two lemmas are alternately used until neither apply.

4.2. Examples

We give results for four small-large examples, whose statistics are described in Table 1, in Table 2. The first example is small and the total cpu time for extraction and verification is under 12 cpu seconds on a VAX 8650. Example 2 is large and is verified in about 1.5 minutes. The third example is a very large machine with 128 states and is also successfully verified within 10 cpu minutes. The first three examples are Moore machines, example 4 is a Mealy machine comparable in size to example 2, but takes almost twice as long due to conversion to a larger Moore machine for equivalence checking.

Note that for all the examples, the invalid state detection time is a small fraction of the total cpu time but a very significant number of invalid states are found. A minimum bit encoding of states minimizes the number of invalid states in a logic level finite automaton, but typically state assignment programs like KISS use encodings with a few more bits than the minimum necessary to implement the machine since great

savings in combinational logic can be made with extra state field bits[dem85].

Verifying equivalence between the two state transition diagrams has a time complexity of $O(E_1{*}E_2)$ where E_1 and E_2 are the number of edges in the two machines. The number of edges in a machine grows approximately as the square of the number of states in the machine. Thus finding invalid states and invalid edges is a big gain - example 3 when run without using don't care information and required 68 minutes to

EXAMPLE	RTL Description		Logic Level Description				
	#states in STT	#edges in STT	#inputs	#outputs	#latches	OLB #gates	NSLB #gates
1	5	10	2	2	4	9	15
2	33	300	10	10	7	220	388
3	128	529	27	56	8	368	667
4	29*	240*	8	16	6	0**	511

* After conversion to Moore, #states = 61, #edges = 417

** Only one block of logic for a Mealy machine.

Table 1. Description of examples

EXAMPLE	Logic Description			Cpu Times (seconds on VAX 8650)				
	#states initial	#invalid states detected	#edges	enum-eration	invalid state detection	edge gener-ation	equiv-alence check	total
1	16	11	2	1.0	1.9	2.4	6.0	11
2	128	94	1165	4.3	12.1	32.1	48.2	97
3	256	126	1372	21.1	71.2	126.2	368.1	587
4	64*	24*	912*	7.6	30.3	46.2	84.1	168

* After conversion to Moore, #states = 74, #edges = 1356.

Table 2. Run-Time Statistics Of Examples

verify.

5. CONCLUSIONS

We have presented an effective method for the verification of two sequential machines at differing levels of abstraction. Previous work in this area involved verifying relatively small sequential circuits at the logic level. By exploiting the don't care information present at the register-transfer level description of a sequential machine we have successfully compared descriptions of large machines at the RTL and logic levels.

Future work in this area includes development of a more efficient algorithm for verifying the equivalence of two state transition graphs of Moore/Mealy machines and more efficient cube enumeration techniques to speed up the verification process.

6. REFERENCES

[bar79]
M. Barbacci, G. Barnes, R. Cattell and D. P. Siewiorek, "The Symbolic Manipulation of Computer Descriptions: ISPS Computer Description Language", Carnegie-Mellon University 1979.

[bra84]
R. K. Brayton, G. D. Hachtel, C. T. McMullen and A. L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, 1984.

[bro85]
M. Browne, E. Clarke, D. Dill and B. Mishra, "Automatic Verification of Sequential Circuits Using Temporal Logic", Technical Report CMU-CS-85-100, Dept. of Computer Science, Carnegie-Mellon University, 1985.

[bry85]
R. E. Bryant, "Symbolic Manipulation of Boolean Functions", Chapel Hill Conference on VLSI, May 1985.

[dem85]
G. De Micheli, R. K. Brayton and A. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines", IEEE Transactions on CAD, July 1985.

[don77]
W. E. Donath and H. Ofek, "Automatic Identification of Equivalence Points for Boolean Logic Verification", IBM Technical Disclosure Bulletin, vol. 18, No 8, Jan. 1976.

[dil85]
D. Dill and E. M. Clarke, "Automatic Verification of Asynchronous Circuits Using Temporal Logic", 1985 Chapel Hill Conference on VLSI, 1985.

[hop79]
J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages and Computation", Addison Wesley, Reading Mass.,1979.

[gar79]
M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the theory of NP-Completeness", W. H. Freeman and Company, 1979.

[goe81]
P. Goel, "An Implicit Enumeration Algorithm To Generate Tests for Combinational Logic Circuits" , IEEE Transactions on Computers, Vol C-30, Mar. 1981.

[mar85]
F. Maruyama and M. Fujita, "Hardware Verification", IEEE Computer. Feb. 1985.

[oda86]
G. Odawara et. al. "A Logic Verifier based on Boolean Comparison", Proc. 23rd Design Automation Conf. June 1986.

[rot77]
P. Roth, "Hardware Verification", IEEE Transactions on Computers, Vol C-26, 1977.

[rot80]
P. Roth, "Computer Hardware Testing and Verification", Computer Science Press, Potomac, Maryland, 1980.

[sup86]
K. Supowit and S. J. Friedman, "A New Method for Verifying Sequential Circuits", Proc. of 23rd Design Automation Conference, June 1986.

[wei86]
R.S. Wei and A. Sangiovanni-Vincentelli, "PROTEUS: A Logic Verification System for Combinational Logic Circuits", Proc. of International Testing Conference, Sept. 1986.