An Automatic Controller Extractor for HDL Descriptions at the RTL

Chien-Nan Jimmy Liu

Jing-Yang Jou
National Chiao Tung University, Taiwan

Extracting controlling finite-state machines can significantly reduce state space and thereby speed functional verification. The controller extraction algorithm uses an approach that frees it from restrictions on HDL code writing style.

With the increasing complexity of modern circuit designs, verification has become the major bottleneck in the entire design process.[1] To cope with the exponential state-space growth, researchers have proposed some techniques[2,3] to reduce this state space in functional verification at the register transfer level (RTL). Because most design errors are related to the design's control part, one possible solution is to separate the data paths from the controllers and verify the control part only. However, in the proposed techniques, the capability of extracting controllers relies on specific control-register labels, which users must assign manually. In large designs, labeling the hundreds of control registers is inconvenient. More importantly, if the original designers are not available—for example, when using vendor-provided intellectual property—assigning labels becomes very difficult.

In modern designs, almost all controllers consist of finite-state machines (FSMs). Therefore, by locating the FSMs, we can find the possible locations of controllers. Some vendors claim their tools[4] can automatically extract FSMs in the original hardware description language (HDL) code. However, most of these tools depend on a specific coding style or user intervention. The literature offers some approaches for translating HDL code into FSMs by compiler techniques.[5,6] As we know, a compiler translates predefined language constructs into other forms. Therefore, compiler-based approaches must also limit users' coding styles. It can be difficult to deal with real designs from various designers with varying coding styles.

To overcome the problems of existing approaches, we propose a novel method for extracting FSMs in HDL code written at the RTL by recognizing the general patterns of FSMs in the process-module (PM) graph. These general patterns are derived from the relationship between an FSM's current states and its next states, not the language constructs. Therefore, the writing style of HDL code is almost entirely unrestricted. Hints or comments in the source code aren't needed either. We already reported on the preliminary stage of this work.[7] The reported experimental results on several real designs from different designers with various coding styles have shown the effectiveness and efficiency of our algorithm.

Because we use the general FSM patterns in the recognition process, some special designs, such as the program counters and the accumulators used in arithmetic logic, may be identified as general FSMs. Although these have general FSM structures, they are in fact only

0740-7475/00/$10.00 © 2000 IEEE

data operators and should not be included in the controller part of the design. Therefore, we propose another method to further reduce controller size by filtering out any such FSMs. With the proposed extraction techniques, we can provide an initial selection of control registers in a very short time. If users have some special consideration about control register selection, we can provide a simple interface that lets them edit the control register list. Then control register selection can be finished quickly and correctly with minimum user intervention.

Besides contributing to state-space reduction in functional verification, the automatic FSM extraction technique can be helpful in many applications. Because FSMs and data paths have significantly different properties, many CAD applications tend to deal with them differently, as revealed in power estimation research.[8] If an automatic FSM extractor is available, this work can be fully automated. In addition, FSM optimization techniques such as state minimization and state assignment are widely available in synthesis tools. If the FSMs in HDL code can be extracted, optimization can be performed automatically, giving users greater convenience and better results. Furthermore, in the HDL debugging tools, the automatic FSM extraction technique can help convert HDL text into graphics so that users can understand designs more quickly.

HDL modeling

To keep the hierarchies in an RTL description, we model the description with a hierarchical PM graph G, as shown in Figure 1. In each hierarchy, a module m will have its own PM graph, say G_m, which is a directed graph $G_m(V, E)$. Each node, $v \in V$, represents a sequential process, a concurrent dataflow statement, or a module instantiation in both VHDL and Verilog. Each directed edge, $e(i, j) \in E$, i and $j \in V$, indicates that node i is a fan-in of node j. To simplify the explanation, in the following discussion we will treat the concurrent dataflow statement as a process. *Label*(v), for each $v \in V$, represents the attribute of each node. This attribute plays an important role in our algorithm and will be further explained in the next section. The different names and meanings of *label*(v) are listed below.

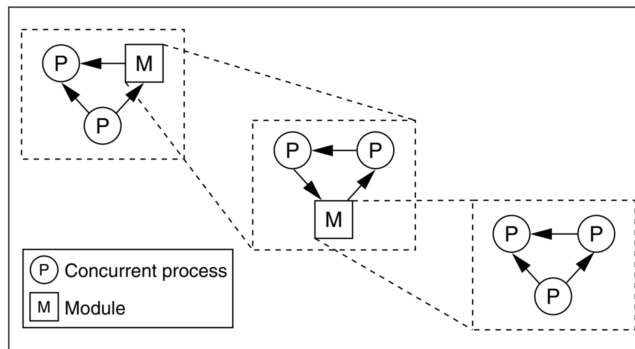

Figure 1. An illustration of HDL modeling.

- SEQ_P: The process has an edge-triggered clock signal.
- COM_P: The process has no edge-triggered clock signal.
- FSM_P: The process has been recognized as part of an FSM.
- SEQ_M: The module contains a sequential process (SEQ_P).
- COM_M: The module contains no sequential process.
- FSM_M: The module has been recognized as part of an FSM.

FSM recognition

HDL's rich constructs afford many different ways to describe the same design. To simplify the problem, and to ensure that our algorithm will retain generality, we make some reasonable preliminary assumptions based on our observation of many examples.

- In RTL descriptions, all the state registers of one FSM must be associated with only one variable.
- The FSM is a synchronous design with edge-triggered flip-flops.
- There are no FSMs embedded in other FSMs.
- All statements in a process must belong to the same FSM if the process is recognized as part of the FSM.

Finding FSMs

Because FSMs' next states always functionally depend on their current states, the signals emanating from the state registers will return to them after going through some combinational paths, regardless of whether the FSMs are Mealy

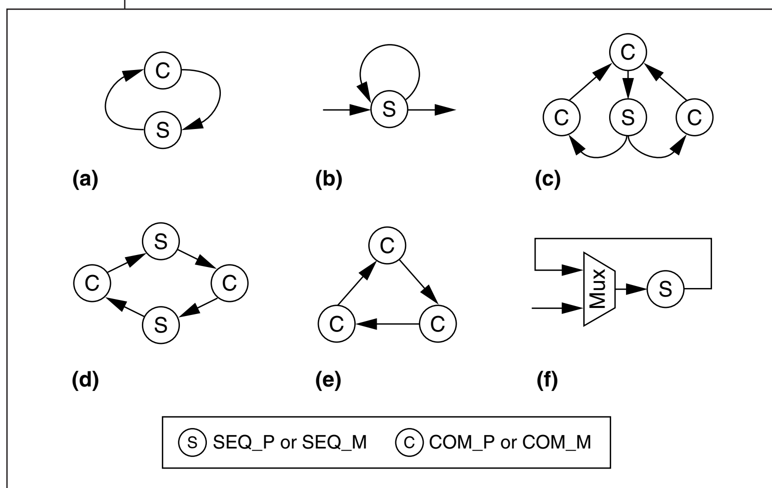

Figure 2. Common loop patterns in HDL code: (a) two process, (b) one process, (c) multiloop, (d) pipeline, (e) combinational, (f) direct feedback.

or Moore type. In other words, we can find the FSMs in the HDL code by finding these identifying loops. Thus, we can derive the most common FSM patterns appearing in HDL code, as shown in Figures 2a-2c. By detecting these loops in the PM graph, we find the possible locations of FSMs. However, not all loops belong to FSMs. According to our preliminary assumptions, those shown in Figures 2d and 2e cannot be considered indicative of FSMs. Neither can the so-called scan flip-flop shown in Figure 2f.

Even if we find a valid loop pattern topologically, we cannot be sure that it belongs to an FSM. Given a node in the PM graph with multiple inputs and outputs, some outputs may not depend on all inputs. Therefore, except for a flattened gate-level design, in which the output of each gate depends functionally on every input of the gate, functional dependency checking is needed after topological loop searching to ensure that every reported loop is valid. So the first step in FSM recognition is to find the loops that start from a sequential node and go through only combinational paths in the PM graph. If those found are functionally dependent, they are considered the main frames of FSMs.

Boundary decision

Using the technique just described, we can easily find the processes that describe the next-state logic of an FSM. However, if the output logic is described in another process, we cannot find that process. Because HDL is a highly modularized language, we can reasonably assume that users will follow the modular writing style, meaning that the output logic will be located in the same module with the next-state logic and the state registers. On the basis of this assumption, we propose a maximum-possible-set-within-module strategy. For each FSM, all combinational nodes in the same module having a relationship with the state registers are recognized as the FSM's output logic. However, nodes having a relationship with two or more FSMs are excluded because they are not likely to belong to any particular FSM (on the basis of our fourth preliminary assumption). They are the communication hardware of these FSMs.

Hierarchy traversal

Our algorithm handles hierarchical descriptions through a strategy called "bottom-up with lumped information." We first traverse the hierarchies to the lowest level and apply the techniques described in the two preceding sections to handle the processes at this level. If the module contains a sequential process (SEQ_P), we label the module SEQ_M; otherwise, we label it COM_M. All information concerning this module is lumped into this label to reduce complexity. Consequently, when we go up one level, the loop detection algorithm can operate in the same way without repeatedly traversing the lower hierarchies.

FSM recognition algorithm

The overall recognition flow, shown below, and the pseudocode for the FSM recognition algorithm, shown in Figure 3, summarize the techniques.

Recognition flow:

1. Read the RTL HDL description and build the hierarchical PM graph G.

2. Set the initial *label(v)* for each node v in G.

3. Call FSM_recognition(G_M) to recognize FSMs, where M is the top module (see Figure 3).

4. Report each FSM found by listing its constituent processes.

In phase 1, the algorithm calls itself recursively to traverse the hierarchies in a depth-first search (DFS) fashion. It starts to recognize the two-process patterns in phase 2. The operations in phase 2 are similar to those in a DFS. Therefore, the complexity is the same as when performing a DFS, that is, $O(V+E)$, where V and E are the numbers of nodes and edges in the PM graph, respectively. In phase 3, the algorithm searches the sequential processes left after phase 2 for the one-process patterns, that is, the self-loop pattern. The complexity of this step is only $O(S)$, where S is the number of sequential nodes in the PM graph. After all pattern recognition is finished, phase 4 determines the output logic. The operations in this step resemble those in a breadth-first search (BFS). Therefore, the complexity is the same as when performing a BFS, that is, $O(V+E)$. The computational complexity of all steps in our algorithm is only linear, so the incurred overhead can be very small.

Controlling FSM selection

Since almost all controllers consist of smaller FSMs and their communication hardware, the proposed FSM-finding technique can find the possible locations of controllers. However, not all FSMs are part of a controller. The use of general FSM patterns in the recognition process means that some special designs may be recognized as general FSMs even though they are only data operators. For example, the program counter is a typical FSM often used in the processor-based design to indicate the next instruction's memory address, but it is often not included in the controllers. If the program counter is included in the controllers, the control part's state space will become extremely large. Therefore, we propose a way to select only the FSMs needed for controllers so that the controlling behavior can be captured and the control part's state space can be reduced.

Definition 1: An FSM *f* is called a controlling FSM if and only if it is part of the controllers of a given design; otherwise, it is called a noncontrolling FSM.

```
FSM_recognition (process_module_graph G) {
// phase 1 : traverse the hierarchy by DFS
  for each node a with label SEQ_M in G
    FSM_list ← FSM_recognition(G_a);

// phase 2 : find 2-process patterns
  Loop_list ← find_topological_loops_with_one_sequential_node(G);
    /* self loops are not included */
  for each loop x in Loop_list {
    Ans ← check_functional_loop(x);
    if (Ans == TRUE) {    /* find valid loop */
      if (the state variables of x has belonged to some FSM i)
            /* multi-loop FSM */
        Add_FSM_list(FSM_list, i, x);
      else     /* new FSM */
        Add_new_FSM_list(FSM_list, x);
      change_node_label_as_FSM(x);
    }
  }

// phase 3 : find 1-process patterns
  for each node p with label SEQ_P in G {
    Ans ← check_self_loop(p);
    if (Ans == TRUE) {    /* find 1-process pattern */
        Add_new_FSM_list(FSM_list, p);
        label(p) ← FSM_P;
    }
  }

// phase 4 : find output logics
  for each FSM f in G {
    node_list ← copy_list(FSM_list, f);
    for each node n in node_list
      color(n) ← f;    /* give nodes in each FSM a different color */
    for each node m in node_list {
      for each node n ∈ fanout(m) {
        if (label(n) == COM_P or COM_M) {
          if (color(n) == NULL)  color(n) ← color(m);
          else color(n) ← DEAD_COLOR;   /* touched twice */
          Add_list(node_list, n);   /* propagate the color out */
        }
      }
    }
  }
  for each node c with label COM_P or COM_M in G {
    if (color(c) != DEAD_COLOR) {    /* belong to only one FSM */
      Add_FSM_list(FSM_list, color(c), c);
      change_node_label_as_FSM(c);
    }
  }

  return FSM_list;
}
```

Figure 3. The FSM recognition algorithm.

Definition 2: A control statement corresponds to a branch point in the dataflow graph. In other words, a control statement is an if statement, an elseif statement, a case statement, or the condition list of a conditional assignment.

If designers don't provide hints, it's hard to determine which FSMs they intended as controllers. However, we can still make some reasonable guesses by analyzing each FSM's usage

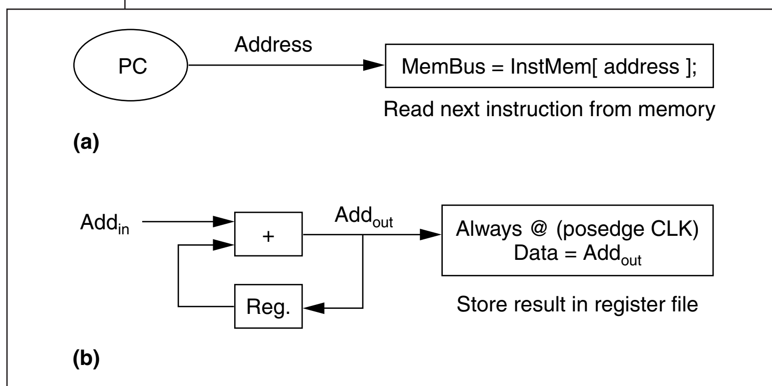

Figure 4. Common usage of noncontrolling FSMs: (a) program counter, (b) accumulator.

at the RTL. The HDL provides several conditional constructs that enable users to easily describe the design's control flow. Typically, we can often find the outputs of a controlling FSM used in the control statements. If an FSM doesn't control the circuit's behavior, its outputs are often used as the operands of another statement, as shown in Figure 4a, or as the input of another register, as shown in Figure 4b. Using this observation, we can define a controlling score for each FSM according to its usage and thereby classify FSMs as controlling or noncontrolling.

Definition 3: If a variable v is functionally dependent on the output *out* of an FSM f through some combinational statements, we say v depends on f_{out}. We define the controlling score of an FSM f as the sum of the output bits on which some variables in the control statements depend.

By definition 3, the controlling scores of the FSMs in Figure 4 are zero. Thus, they are noncontrolling FSMs. Only FSMs with a nonzero controlling score can be controlling FSMs. Therefore, after the FSMs are extracted, a simple DFS-like search analyzes their usage and calculates their controlling scores according to definition 3. With the controlling scores, we can easily extract a design's controllers from the controlling FSMs. These operations have only linear complexity, so the incurred overhead is also very small.

Experimental results

Our algorithm has been implemented in C++. Table 1 shows experimental results on five real designs and provides information about those designs. We obtained the running results shown in the table on a 300-MHz UltraSparc II. PCPU is a simple 32-bit DLX CPU. MEP is a block-matching motion estimation processor used in the MPEG-II system. PTME is a processor for 3D graphics perspective texture mapping. MPC is a programmable MPEG-II system controller. The last circuit, DCT, is a 2D discrete cosine transform/inverse discrete cosine transform design.

Typical designs often include many registers; however, only a few registers belong to the controllers. The experimental results show that we can find this small portion of registers quickly by using our approach. The number of selected control registers is relatively small in every case, and the state register reduction ratio (the number of registers in the design divided by the number of registers included in controlling FSMs) is very significant. Because of the exponential relationship between the state registers and the number of states, the state space can be dramatically reduced. In other words, the time and memory space required to verify the designs are also greatly reduced. In addition, the five designs in the table were obtained from different designers, reflecting five different writing styles. This demonstrates the algorithm's generality.

Table 1. Experimental results of the controller extraction algorithm.

Designs	PCPU	MEP	PTME	MPC	DCT
Lines in HDL code	913	2,062	3,980	4,183	5,629
Nodes in PM graph	101	2,585	3,590	336	7,370
Edges in PM graph	136	3,749	3,577	595	3,092
Registers in the design	1,380	10,494	1,387	67,377	1,071
FSMs found in PM graph	3	24	12	19	20
Nodes included in FSMs	21	126	119	89	1,042
Registers included in FSMs	67	1,063	92	151	242
Controlling FSMs (CFSMs)	1	5	10	12	10
Nodes included in CFSMs	5	59	38	60	286
Registers included in CFSMs	3	130	51	117	62
State register reduction ratio	460	80.7	27.2	575.9	17.3
User satisfaction	100%	100%	100%	100%	100%
Extraction time (seconds)	0.52	13.74	6.10	3.31	5.73

To assess the correctness of our

algorithm, we asked the designers to review the lists of control registers our tools generated and report the matching percentage in the user satisfaction row of Table 1. According to the reported data, we found all the intended control registers in the five designs. Moreover, the operations can be completed within a few seconds even for the largest design, making computation overhead very small.

ALL FSMS DETECTED by our method represent typical styles with one latching edge. There are other, extended FSMs such as pipelined FSMs or double-latching FSMs. Future work may include extending the algorithm to handle those cases. In addition, there remains the interesting problem, after FSMs have been identified, of generating their state transition graphs from the HDL code without any limitation on writing styles. A practical solution to this problem is another of our goals. ■

Acknowledgments

This work was supported in part by Novas Software Inc. and the R.O.C. National Science Council under grant NSC89-2215-E-009-009. We also thank the SI2 group in the Department of Electronics Engineering at the National Chiao Tung University for kindly providing their designs for our experiments.

■ References

1. A. Evans et al., "Functional Verification of Large ASICs," *Proc. 35th ACM/IEEE Design Automation Conf.*, ACM, New York, June 1998, pp. 650-655.
2. R.C. Ho and M.A. Horowitz, "Validation Coverage Analysis for Complex Digital Designs," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design,* IEEE Computer Soc. Press, Los Alamitos, Calif., Nov. 1996, pp. 146-151.
3. D. Moundanos, J.A. Abraham, and Y.V. Hoskote, "Abstraction Techniques for Validation Coverage Analysis and Test Generation," *IEEE Trans. Computers*, Vol. 47, No. 1, Jan. 1998, pp. 2-14.
4. T.-H. Wang and T. Edsall, "Practical FSM Analysis for Verilog," *Proc. IEEE Int'l Verilog HDL Conf. and VHDL Users Forum*, IEEE Computer Soc. Press, Los Alamitos, Calif., Mar. 1998, pp. 52-58.
5. S.-T. Cheng et al., "Compiling Verilog into Timed Finite State Machines," *Proc. IEEE Int'l Verilog HDL Conf.*, IEEE Computer Soc. Press, Los Alamitos, Calif., Mar. 1995, pp. 32-39.
6. Y.V. Hoskote et al., "Automatic Verification of Implementations of Large Circuits Against HDL Specifications," *IEEE Trans. Computer-Aided Design*, Vol. 16, No. 3, Mar. 1997, pp. 217-228.
7. C.-N. Liu and J.-Y. Jou, "A FSM Extractor for HDL Description at RTL Level," *Proc. Fifth Asia-Pacific Conf. Hardware Description Languages*, Society of CAD and VLSI Design Research, IEEK, Seoul, Korea, July 1998, pp. 33-38.
8. D.I. Cheng et al., "New Hybrid Methodology for Power Estimation," *Proc. 33rd ACM/IEEE Design Automation Conf.*, ACM, New York, June 1996, pp. 439-444.

Chien-Nan Jimmy Liu is a PhD candidate in the Department of Electronics Engineering at the National Chiao Tung University, Taiwan. His primary research interest is functional verification of designs written in HDL at the register transfer level. He holds a BS in electronics engineering from the National Chiao Tung University.

Jing-Yang Jou is a professor in the Department of Electronics Engineering at the National Chiao Tung University, Taiwan. His research interests include behavioral and logic synthesis, VLSI designs and CAD for low power, design verification, synthesis and design for testability, and hardware/software co-design. Jou received a BS in electrical engineering from the National Taiwan University and an MS and a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of Tau Beta Pi.

■ Direct comments and questions about this article to Jing-Yang Jou, Department of Electronics Engineering, National Chiao Tung University, 1001 Ta-Hsueh Rd., 300 HsinChu, Taiwan, R.O.C.; jyjou@bestmap.ee.nctu.edu.tw.