Designing Finite State Machines with PEG

Gordon Hamachi

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

ABSTRACT

PEG is a finite state machine compiler. It translates high level language descriptions of finite state machines into the logic equations needed to implement state machine designs. Since the output format is compatible with *eqntott*, PEG may be used as a front end for Berkeley PLA tools.

1. Introduction

PEG (PLA Equation Generator) is a design tool for finite state machines. It compiles high level language descriptions of finite state machines into the logic equations needed to implement a design.

PEG programs are isomorphic to Moore machine state diagrams. There is a one-to-one correspondence between states in a state diagram and state definitions in the corresponding *PEG* program. The translation from state diagrams to *PEG* programs is simple and straightforward.

Designing with PEG provides a number of advantages over the traditional pencil-and-paper approach method of FSM design. PEG's high level language enables designs and design changes to quickly be implemented. PEG programs provide easy-to-understand documentation with clear control flow. PEG does the tedious and error-prone bookkeeping task of generating *output* and *next state* bits as a function of current state bits. It checks for design errors and eliminates redundant terms in logic equations.

As output PEG generates logic equations in the *eqn* format accepted by *eqntott* [Cmelik], another Berkeley design tool. By piping the output of PEG

through *eqntott*, PEG may be used as a front end for Berkeley PLA tools such as *mpla* [Mayo], and *espresso* [Rudell]. As an option, *PEG* will also print the unminimized truth table from which the logic equations are derrived.

2. A Simple Example

Figure 1 shows the state diagram for a four-state finite state machine implementing a 2-bit binary counter. The *PEG* description of this design appears in Figure 2. The program has no inputs besides an implicit clock. The outputs of the state machine are its *next state* bits, which are automatically generated by *PEG*.

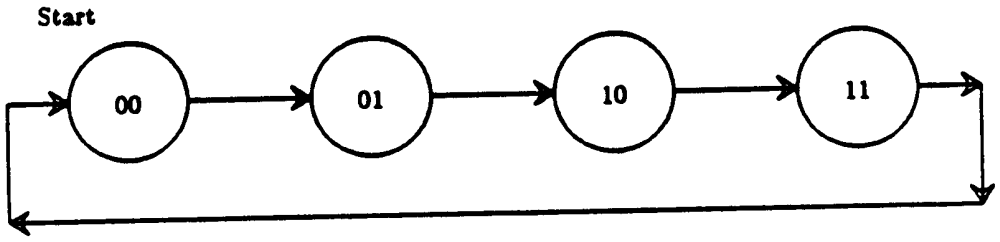

Figure 1: State Diagram for Example 1

In its most simple form, a *PEG* program consists of a list of state descriptions. The sample program has four states. Each state has four parts: an optional label, a colon, an optional signal assertion part, and and optional control part.

```
--Simple PEG program for 2-bit counter
--State transition on every clock
--No reset => starts in a random state

Start       :       --This is state 0
            :       --This is state 1
            :       --This is state 2
            :       --This is state 3
                      GOTO Start;
```

Figure 2: *PEG* Program for Example 1

The first state in the example is labeled with the identifier *Start*. The label is necessary only because of the GOTO from state 3 back to state 0.

States 1 and 2 are examples of the minimal state description. These states are completely defined by a colon, which acts as a place holder for the state. Empty states, in which no branching or signal assertions occur, are sometimes used to introduce necessary delays in FSMs.

Flow of control in *PEG* programs is sequential unless otherwise specified. Since no control information is present for states 0, 1, and 2, the program steps sequentially through the states 0, 1, 2, and 3. State 3 has control information specifying a jump back to the state labeled *Start*.

Since it has no sequential *next state*, control must always be defined for the last state in the program. *PEG* generates an error message and quits if control is not defined for the last state.

Although state transitions are performed on clock ticks, no clock is mentioned in the program. It is the user's responsibility to implement the state machine with synchronous logic to latch input and output signals.

Comments begin with a double dash "–" and terminate at the end of the line on which they appear. The first three lines of the program are comments. Comments also appear on lines 5 through 8.

Input is free-format. White space may appear anywhere in a program to enhance readability.

3. Interpreting the Output

Assuming that the *PEG* program for example 1 is in a file called *counter*, the following Unix command line may be used to invoke *PEG*:

peg counter

The resulting output is shown in figure 3. Generating a PLA from the same input file is accomplished with the command line:

peg counter | eqntott | mpla –I –O

Mpla will not automatically connect *next-state* outputs to *current-state* inputs. After generating the PLA the state outputs must be manually wired to the state inputs.

```
INORDER     =     InSt0* InSt1*;
OUTORDER    =     OutSt1* OutSt0*;
OutSt1*     =     (!InSt1*);
OutSt0*     =     ( InSt0*&!InSt1*)| (!InSt0*& InSt1*);
```

Figure 3: PLA Equations for Example 1.

3.1. Equations

PEG generates the two input variables *InSt0** and *InSt1** which are the state inputs for the finite state machine. It also generates two output variables *OutSt0** and *OutSt1**, the next-state outputs. Any signal name ending with an

asterisk was generated by *PEG*.

The INORDER and OUTORDER statements specify that the resulting PLA inputs and outputs, from left to right, are InSt0*, InSt1*, OutSt1*, and OutSt0*.

Following the OUTORDER statement are the logic equations for the two output variables, OutSt1* and OutSt0*. The exclaimation mark "!" indicates logical negation. The ampersand "&" signifies the logical *AND*, while the vertical bar "|" signifies a logical *OR*.

3.2. Truth Table

The -t option generates a truth table for the finite state machine. This truth table is written to the file *peg.summary*. The truth table for example 1 is shown in figure 4.

INPUTS:		s00: s01:	InSt0* (msb) InSt1* (lsb)	
OUTPUTS:		n01: n00:	OutSt1* (lsb) OutSt0* (msb)	
State Table	s 0	s 1	n 1	n 0
	0	0	1	0
	0	1	0	1
	1	0	1	1
	1	1	0	0

Figure 4: Truth Table for Example 1.

Labels across the top of the truth table identify its columns. The mapping from column labels to actual signal names is given in the lists of input and output signals which preceed the truth table. To the right of the truth table are the names of the states described by the rows of the table.

4. Another Example

The second and more complex example shows the state diagram and corresponding *PEG* program for a FSM which recognizes the regular expression (1|0)*100. The state diagram for this FSM is shown in figure 5.

The PEG program which implements this design is given in figure 6. Figure 6 describes a state machine with four states. The state machine has two inputs, *RESET* and *in*, and one output, *accept*.

Assume the text of figure 2 is in a file called *prog*. Logic equations for the state machine are generated by running the command

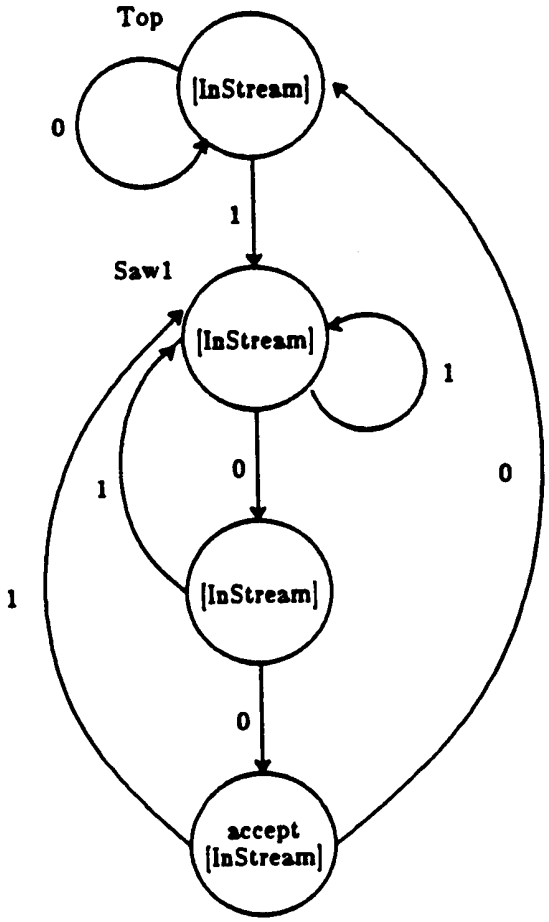

Figure 5: State Diagram Recognizing (1|0)*100

peg prog

Since this program has two inputs, they are declared in the *INPUTS* statement. If a *PEG* program has any inputs they must be declared in an *INPUTS* statement which must be the first statement in the program. The input *RESET* is a special keyword input. The other program input, *InStream*, is used to generate the *next state* for the FSM.

RESET indicates that when the *RESET* signal is asserted the state machine jumps to the top of the program, which in this case is named *Top*. When this keyword is present, conditional branches to the first state are automatically added to the *next state* expressions for each state. If *RESET* is not listed as an input, the program initializes in a random state.

IF the FSM designer does not want to pay the penalty of a larger and slower finite state machine, *RESET* may be omitted as it was in example 1. In this case

```
-Simple FSM example:  Accepts the regular expression (1|0)*100

INPUTS      :     RESET InStream;
OUTPUTS     :     accept;

Top         :     IF NOT InStream THEN LOOP;          --0*

Saw1        :     IF InStream THEN LOOP;              --1

            :     IF InStream THEN Saw1;              --10

            :     ASSERT accept;
                  IF InStream THEN Saw1 ELSE Top;     --100
```

Figure 6: PEG Program Recognizing (1|0)*100

```
INORDER     =     RESET InStream InSt0* InSt1*;
OUTORDER    =     OutSt1* OutSt0* Accept;
OutSt1*     =     (!RESET& InStream) |
                  (!RESET&!InStream& InSt0*&!InSt1*);
OutSt0*     =     (!RESET&!InStream& InSt0*&!InSt1*) |
                  !InStream&!InSt0*& InSt1*);
Accept      =     ( InSt0*& InSt1*);
```

Figure 7: Equations for Example 2.

the reset function may be external to the *PEG* program by implementing the FSM in such a manner that the *next state* feedback lines are pulled low when the *RESET* signal is asserted. This method will work because the top state in a *PEG* program is always assigned to state zero.

The *OUTPUTS* statement declares that this program has a single output called *accept*. The FSM asserts this signal high if a string in the given grammar is recognized. If any outputs are generated by a *PEG* program, they must be declared in an *OUTPUTS* statement which immediately follows the *INPUTS* statement. If no *INPUTS* statement is present, then the *OUTPUTS* statement is the first program statement.

INPUTS:	i00: RESET i01: InStream s00: InSt0* (msb) s01: InSt1* (lsb)
OUTPUTS:	n01: OutSt1* (lsb) n00: OutSt0* (msb) o00: Accept

State Table	i 0	i 1	s 0	s 1	n 1	n 0	o 0
1	-	0	0	0	0	-	Top
0	0	0	0	0	0	-	Top
0	1	0	0	1	0	-	Top
1	-	0	1	0	0	-	Saw1
0	0	0	1	0	1	-	Saw1
0	1	0	1	1	0	-	Saw1
1	-	1	0	0	0	-	Saw1+1
0	0	1	0	1	1	-	Saw1+1
0	1	1	0	1	0	-	Saw1+1
1	-	1	1	0	0	1	Saw1+2
0	0	1	1	0	0	1	Saw1+2
0	1	1	1	1	0	1	Saw1+2

Figure 8: Truth table for Example 2.

Example 2 introduces the *IF-THEN-ELSE* control construct. This construct is used to provide two-way branches ***based only on a single input signal.*** Branches based on more than one input signal are handled by the *CASE* statement which has not yet been presented. *IF* statements do not nest: Statements of the form *IF-THEN-ELSE-IF* are not allowed. The syntax of the *IF* is:

IF [NOT] <signal> THEN <state name> [ELSE <state name>] ;

The *ELSE* clause is optional: If it is omitted, the *ELSE* defaults to the next sequential state in the program. Thus, in state *Top*, if *InStream* is high, then the condition in the *IF* is false and the program takes the default branch to state *Saw1*.

The alert reader will have noticed that the state name *LOOP* is used but not defined. This is intentional. *LOOP* is a keyword which means to stay in the current state. It is an error to define a state with the label *LOOP*.

The final state in example 2 shows the first use of the *ASSERT* statement. The *accept* signal is asserted only in the accepting state of the FSM. If an *ASSERT* statement is present in the definition of a state, it must preceed the state's control statement.

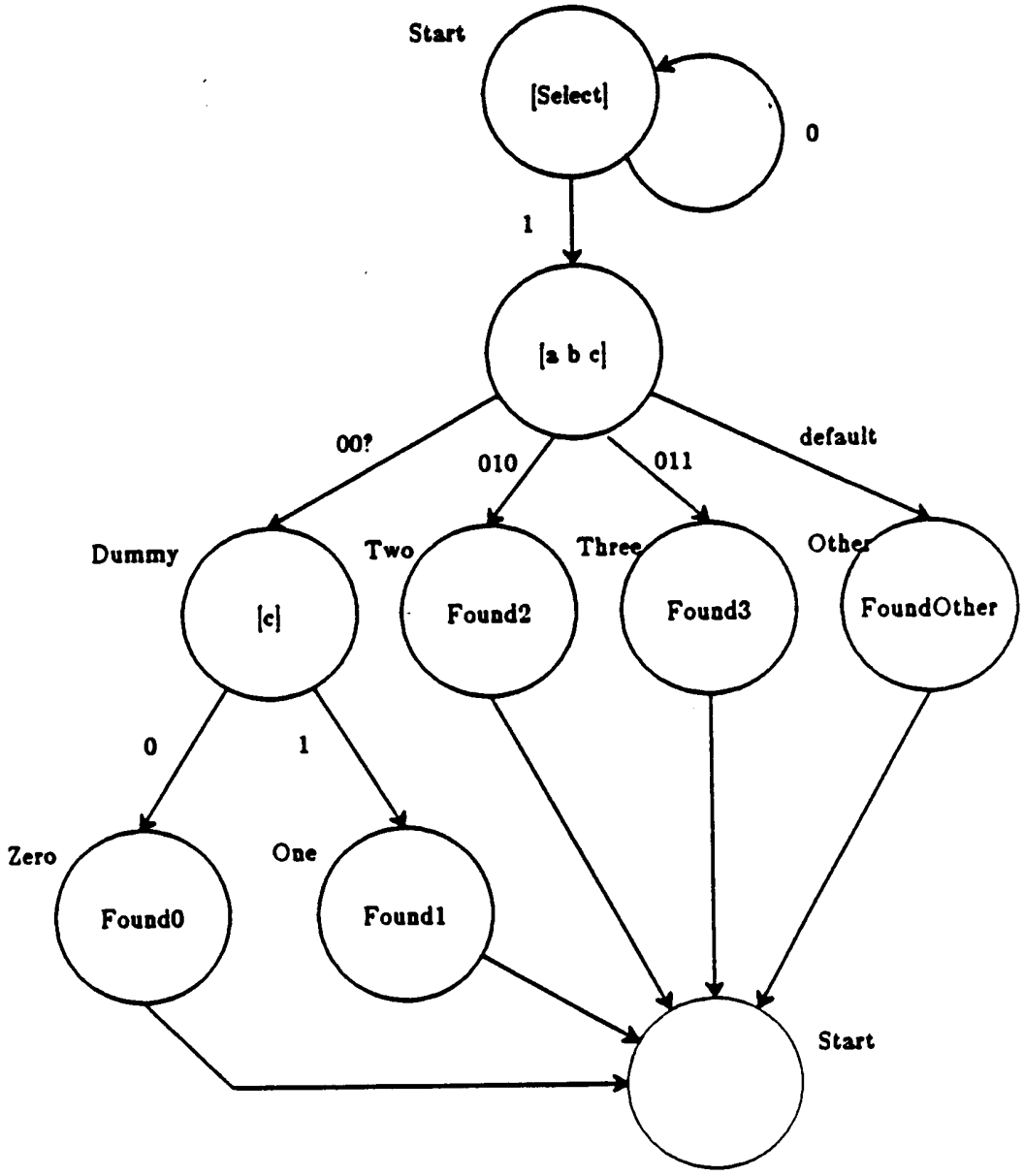

Figure 9: State Diagram for Example 3

Figure 11 shows an ambiguous case specifier. It is ambiguous because more than one case selector applies to the input (0 1 0). In such cases *PEG* processes

```
--Decode inputs a, b, and c into
--0, 1, 2, 3, or "other".

INPUTS      :   RESET Select a b c;
OUTPUTS     :   Found0 Found1 Found2 Found3 FoundOther;

Start       :   --This is the reset state
                IF NOT Select THEN LOOP;

            :   CASE (a b c) --Second state
                  0 0 ? ==> Dummy; --A don't-care
                  0 1 0 ==> Two;
                  0 1 1 ==> Three;
                ENDCASE==>Other;

Dummy       :   IF c THEN One;

Zero        :   ASSERT Found0; GOTO Start;

One         :   ASSERT Found1; GOTO Start;

Two         :   ASSERT Found2; GOTO Start;

Three       :   ASSERT Found3; GOTO Start;

Other       :   ASSERT FoundOther; GOTO Start;
```

Figure 10: PEG Program for Example 3

the list of case selectors from top to bottom, using the first one that applies to the inputs. Since the case specifier for State2 comes first, it defines the next state for inputs (0 1 0) and (0 1 1). The case specifier for State3 defines the next-state only for the case (1 1 0).

```
State1      :   CASE (a b c)
                  0 1 ? ==> State2;
                  ? 1 0 ==> State3;
                ENDCASE==>State4;
```

Figure 11: Ambiguity Resolution in Don't-Cares

5. Final Example

Figures 9 and 10 show the state diagram and *PEG* program for a state machine which decodes 3 bits into 0, 1, 2, 3, and "other". Example 3 shows the use of multiple inputs, multiple outputs, and multi-way branches.

Multi-way branches and branches based on two or more inputs are handled by the *CASE* statement. The *CASE* statement consists of the keyword *CASE* followed by an input signal list, a list of case selectors, and an *ENDCASE*.

A case selector specifies two things: a bit pattern corresponding to the input signals, and a *next-state* for that combination of inputs. Bit patterns are strings composed of the characters '0', '1', and signals in the input signal list. Don't-cares are specified with *?*.

The *ENDCASE* statement optionally specifies the default next-state if none of the other case selectors applies to the input. In keeping with the model of sequential execution, if the *ENDCASE* does not specify a next-state, the next-state defaults to the state following the one in which the *CASE* statement appears.

6. References

[CADMan]
CAD Manual, Online Unix documentation.

[Danford]
Peggy Danford, Private communication with author, June 1982.

[Unix]
Unix Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, University of California at Berkeley, November 1980.

7. Peg Syntax

```
<program>      : <InputList> <OutputList> <StateList>

<InputList>    : INPUTS : <IdentList> ; | /*NULL*/

<OutputList>   : OUTPUTS : <IdentList> ; | /*NULL*/

<StateList>    : <State> | <StateList> <State>

<IdentList>    : <Identifier> | <IdentList> <Identifier>

<State>        : <Identifier> : <Signals> <Control> | : <Signals> <Control>

<Signals>      : /*null*/ | <ASSERT> <IdentList> ;

<Control>      : CASE ( <IdentList> ) <Cases> <DefaultCase>
                 | IF <Identifier> THEN <Identifier> ;
                 | IF <Identifier> THEN <Identifier> ELSE <Identifier> ;
                 | IF <NOT> <Identifier> THEN <Identifier> ;
                 | IF <NOT> <Identifier> THEN <Identifier> ELSE <Identifier> ;
                 | GOTO <Identifier>
                 | /*NULL*/

<Cases>        : <Cases> <CaseStmt> | <CaseStmt>

<CaseStmt>     : <BitList> => <Identifier> ;

<Bit>          : 0 | 1 | ?

<BitList>      : <BitList> <Bit> | <Bit>

<DefaultCase>  : ENDCASE => <Identifier> ; | ENDCASE ;

<NOT>          : "!" | "NOT" | "-"

<Comment>      : "--".*$

<Identifier>   : [A-Za-z][A-Za-z0-9._]*
```