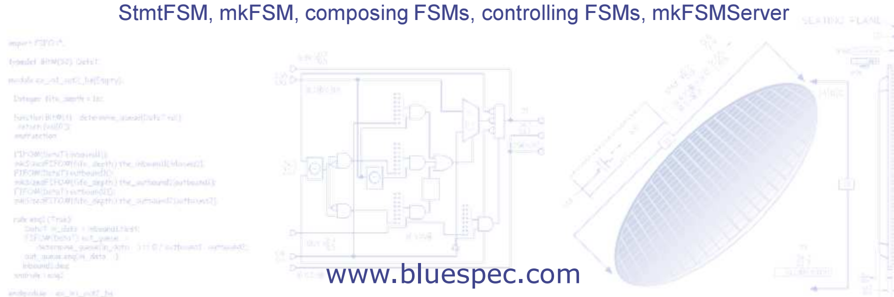
BSV Training

Section: FSM sub-language

StmtFSM, mkFSM, composing FSMs, controlling FSMs, mkFSMServer



www.bluespec.com

Motivations

Finite State Machines (FSMs) are very common in hardware design.

With BSV rules, you can encode arbitrary FSMs.
For example, a simple FSM involving a sequence of states and a loop:

```
typedef enum { S0, S1, S2, ... } State deriving (Bits, Eq);

module mkFoo (...);
   Reg #(State) state <- mkReg (S0);

   rule r0 (state == S0);
      ... do state S0 actions ...
      state <= S1;                         // next state
   endrule

   rule r1 (state == S1);
      ... do state S1 actions ...
      state <= (some cond ? S1 : S2);      // loop back to S1 state, or exit loop
   endrule

   rule r2 (state == S2);
      ... do state S2 actions ...
      ... transition to next state ....
   endrule

endmodule
```

Structured FSMs can be expressed more succinctly

There are common structured design patterns in FSMs:

- sequences, conditionals, loops, parallel threads, etc.

BSV provides a powerful FSM sub-language to express these more succinctly than having to write out the rules explicitly:

- But note that the semantics are *identical* to rules
- In fact, the compiler expands the FSM spec into the rules that you would have written by hand if you were to express them directly as rules

To use this facility:

- Import the “StmtFSM” package: `import StmtFSM :: *;`
- Create an FSM specification (an expression of type Stmt)
- Create an FSM module by giving it the specification
  - This returns an “FSM” interface with “start” and “done” methods
- Operate the FSM using the “start” and “done” methods

The FSM interface

```
interface FSM;
   method Action  start;
   method Bool    done;
   method Action  waitTillDone;
   method Action  abort;
endinterface: FSM
```

FSM interface

```
module mkFSM #( Stmt  s ) ( FSM );
```

mkFSM module

Note:

- ‘done’ and ‘waitTillDone’ are just alternative ways for knowing when the FSM is done. You can either test the boolean ‘fsm.done’, or you can execute the Action statement ‘fsm.waitTillDone’, whose implicit condition is the same as fsm.done.

  In different situations, one of the other may be more convenient.

- ‘abort’ allows an external agent to stop the FSM no matter what state it is in, and no matter how deeply nested it is (more about nesting later)

Example revisited, with FSMs instead of rules

```
import StmtFSM :: *;

module mkFoo (...);
   Stmt stmt = seq
                    ... do state S0 actions ...
                    ... do state S1 actions ...
                    while (some cond) ... do state S1 actions ...
                    ... do state S2 actions ...
                 endseq;

   FSM fsm <- mkFSM (stmt);

   rule init (...);
      fsm.start;   ...
   endrule

   rule done (! fsm.done);
      ...
   endrule
endmodule
```

- Import the “StmtFSM” package:
- Create an FSM specification (an expression of type Stmt)
- Create an FSM module by giving it the specification
  - This returns an “FSM” interface with “start” and “done” methods
- Operate the FSM using the “start” and “done” methods

```
while (some cond) ... do state S1 actions …
```

can also be written as

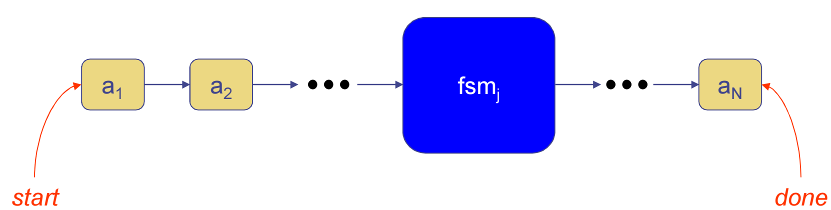
```
while (True) seq
   ... do state S1 actions ...
   if (some cond) break;
endseq
```


Composing FSMs

FSMs can be composed—individual FSMs can be combined systematically into larger FSMs.

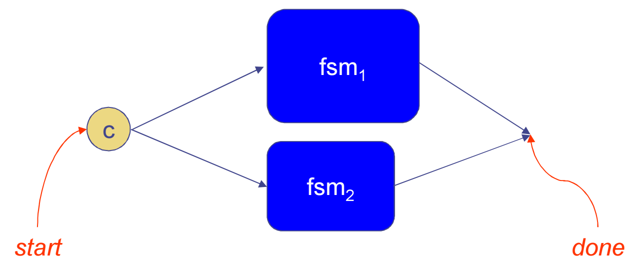
The compositional principle is:
- Every FSM has a *start* and *done* state, embodied in the FSM interface methods
- When composing a larger FSM, the *start* and *done* for the larger FSM is systematically derived from the *start* and *done* of the component FSMs

Example: Linear sequencing
- Syntax: **seq** a1; a2; ... ; aN; **endseq**
- Where each aJ is either an expression of type Action, or itself a sub-fsm (expression of type Stmt)



Conditionals

Syntax
- **if** (Bool expr) fsm1
- **if** (Bool expr) fsm1 **else** fsm2

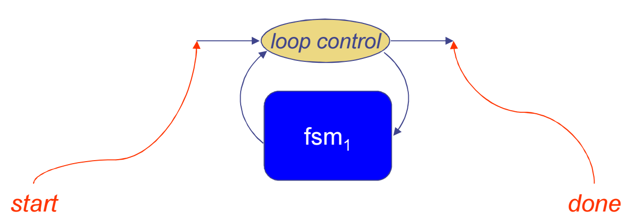


bluespec

Iteration (loops)

Syntax
- **for** (loop control) fsm1
- **while** (Bool expr) fsm1
- **repeat** (Integer expr) fsm1

Loop bodies can contain **break** and **continue** keywords, with the usual meaning



bluespec

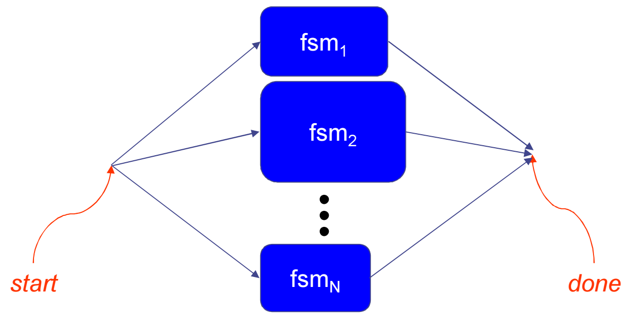
Parallel composition (fork-join)

Syntax

- **par** fsm1; fsm2; ...; fsmN; **endpar**

The sub-FSMs start at the same time and proceed independently.
The FSM is done when the all sub-FSMs are done (they need not complete simultaneously; we wait for the last one to be done).

Note: the sub-FSM can contain Actions that conflict with each other—these are resolved automatically using standard rule scheduling.



Another FSM example

```
Stmt specfsm =
  seq
    write( 15, 51 );
    read( 15 ) ;
    ack ;
    ack ;
    write( 16, 61 ) ;
    write( 17, 71 ) ;
    // a memory operation and an
    // acknowledge can occur
    // simultaneously
    action
      read( 16 ) ;
      ack ;
    endaction
    action
      read( 17 ) ;
      ack ;
    endaction
    ack ;
    ack ;
  endseq ;

FSM testfsm <- mkFSM (specfsm);
```

- action/endaction blocks compose larger entities of type Action. Since an Action is always within a rule, it guarantees that the sub-actions will be simultaneous (atomic).

```
rule run ( True );
  testfsm.start ;
endrule

rule done (testfsm.done);
  $finish(0);
endrule
```

FSMs are often used as testbench stimulus generators

Example:

```
// Specify an FSM generating a test seqence
Stmt test_seq =
  seq
    for (i <= 0; i < NI; i <= i + 1)              // each source
      for (j <= 0; j < NJ; j <= j + 1) action     // each destination
         let pkt <- gen_packet ();
         send_packet (i, j, pkt);                 // test i-j path in isolation
      endaction

    // then, test arbitration by sending packets simultaneously to same dest
    action
       send_packet (0, 1, pkt0);                  // to dest 1
       send_packet (1, 1, pkt1);                  // to dest 1 (so, collision)
    endaction
  endseq;

mkAutoFSM (test_seq);       // Generate the FSM and code to run it automatically
```

mkAutoFSM is another module provided in the library:

- It has an Empty interface
- Internally, it uses mkFSM to create the FSM, and it creates rules
  - to automatically start the FSM
  - to invoke $finish when the FSM is done

Revisiting our testbench from the EHRs lecture …

```
module mkTest (Empty);
   UpDownSatCounter_Ifc  ctr <- mkUpDownSatCounter;
   Reg #(int) step <- mkReg (0);
   Reg #(Bool) flag0 <- mkReg (False); Reg #(Bool) flag1 <- mkReg (False);

   function Action count_show (Integer rulenum, Bool a_not_b, Int #(4) delta);
      action
         let x <- (a_not_b ? ctr.countA (delta) : ctr.countB (delta));
         $display ("cycle %0d, r%0d: is %0d, count (%0d)", cur_cycle, rulenum, x, delta);
      endaction
   endfunction

   // Rules 0-9 are seqential, just testing one method at a time
   rule r0  (step == 0);  count_show (0, True,   3); step <= 1;  endrule
   rule r1  (step == 1);  count_show (1, True,   3); step <= 2;  endrule
                  ... and similarly, sequentially feed deltas of  3,3, -6,-6,-6,-6, 7, 3,
   // Concurrent execution
   rule r10 (step == 10 && !flag0); count_show (10,True,   6); flag0 <= True; endrule
   rule r11 (step == 10 && !flag1); count_show (11,False, -3); flag1 <= True; endrule

   // Show final value
   rule r12 (step == 10 && flag0 && flag1); count_show (12,True, 0); $finish; endrule
endmodule: mkTest
```

These parts just constitute a structured FSM.
On the next slide we use StmtFSM instead of explicit rules

Revisiting our testbench from the EHRs lecture …

```
module mkTest (Empty);
   UpDownSatCounter_Ifc  ctr <- mkUpDownSatCounter;

   function Action count_show (Integer rulenum, Bool a_not_b, Int #(4) delta);
      action
          let x <- (a_not_b ? ctr.countA (delta) : ctr.countB (delta));
          $display ("cycle %0d, r%0d: is %0d, count (%0d)", cur_cycle, rulenum, x, delta);
      endaction
   endfunction

   mkAutoFSM (
      seq
         count_show (0, True,   3);
         count_show (1, True,   3);
                  ... and similarly, sequentially feed deltas of  3,3, -6,-6,-6,-6, 7, 3,
         par
            count_show (10,True,   6);
            count_show (11,False, -3);
         endpar
         count_show (12,True, 0);
      endseq);
endmodule: mkTest
```

mkAutoFSM is just a BSV library function that
- *takes a Stmt argument (here, "seq…endseq"),*
- *creates an FSM from the Stmt,*
- *runs it once,*
- *and calls $finish*

```
module mkAutoFSM
#(Stmt)(Empty);
   FSM fsm <- mkFSM (Stmt);
   rule rA:
      fsm.start;
   endrule
   rule rB (fsm.done);
      $finish;
   endrule
endmodule: mkAutoFSM
```


Suspendable FSMs

The library provides another FSM constructor:

module mkFSMWithPred #(Stmt s, Bool b) (FSM);

An external agent can "asynchronously" start/stop the FSM by controlling the boolean predicate b.

[Language gurus: With parallel composition, nesting, abort and suspend, you get similar expressive power to FSMs in the language Esterel.   The Esterel literature characterizes this power—it allows FSM descriptions to be *<u>exponentially</u> smaller than descriptions without these capabilities.]*

FSM "servers"

Another useful composition facility is the FSM Server.
Normally, to perform a multi-cycle request to a server, your FSM would have to express it in a "split-phase" fashion:

```
Stmt s = seq
            mem.request.put (Req {op: Load, addr: a});
            let response <- mem.response.get;
         endseq
```

With FSM servers, you can express this in a more traditional "procedure call" style.

```
function RStmt #(Data) fn_memServer (Req req);
  seq
     ... do the work for reading mem data ...
     return data;       // RStmt is a generalization of Stmt with 'return' values
  endseq
endfunction

FSMserver #(Addr, Data) memServer <- mkFSMServer (fn_memServer);

Stmt s = seq
            response <- callServer (memServer, Req {op: Load, addr:a });
         endseq
```


More info on FSMs

Section C.6.1 in the Reference Guide goes into a lot more detail on FSMs,
including describing further functions, modules, etc.
and providing many more examples.

End

Questions?
Join online forums at www.bluespec.com, and ask your question,
or send an e-mail to support@bluespec.com