Stmt FSM

Arvind (with the help of Nirav Dave)
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

March 10, 2010
http://csg.csail.mit.edu/6.375

Motivation

❤ Some common design patterns are tedious to express in BSV

❤ Testbenches
❤ Sequential machines (FSMs)
  ♦ especially sequential looping structures

These are tedious to express in Verilog as well (but not in C)
Testing the IP Lookup Design

- Input: IP Address
- Output: Route Value
- Need to test many different input/output sequences

Testing IP Lookup

- Call many streams of requests responses from the device under test (DUT)

**Case 1**
- `dut.enter(17.23.12.225)`
- `dut.getResult()`
- `dut.enter(17.23.12.25)`
- `dut.getResult()`

**Case 2**
- `dut.enter(128.30.90.124)`
- `dut.enter(128.30.90.126)`
- `dut.getResult()`
- `dut.getResult()`
- `dut.getResult()`

March 10, 2010  
http://csg.csail.mit.edu/6.375
But we usually want more counters, display, ...

```verbatim
function Action makeReq(x);
    action
        reqCnt <= reqCnt + 1;
        dut.enter(x);
        $display("[Req #:",fshow(reqCnt),"] = ",fshow(x));
    endaction
endfunction

function Action getResp();
    action
        resCnt <= resCnt + 1;
        let x <- dut.getResult();
        $display("[Rsp #:",fshow(resCnt),"] = ",fshow(x));
    endaction
endfunction
```
A more complicated Case:
Initializing memory

C

```c
int i; Addr addr=addr0;
bool done = False;
for(i=0; i<nI; i++){
    mem.write(addr++,f(i));
}
done = True;
```

BSV

```bsv
Reg#(int) i <= mkReg(0);
Reg#(Addr) addr <= mkReg(addr0);
Reg#(Bool) done <= mkReg(False);

rule initialize (i < nI);
    mem.write (addr, f(i));
    addr <= addr + 1;
    i <= i + 1;
    if (i+1 == nI) done<=True;
endrule
```

Initialize a memory with a 2-D pattern

BSV

```bsv
Reg#(int) i <= mkReg(0);
Reg#(int) j <= mkReg(0);
Reg#(Addr) addr <= mkReg(addr0);
Reg#(Bool) done <= mkReg(False);

rule loop ((i < nI) && (j < nJ));
    mem.write (addr, f(i,j));
    addr <= addr + 1;
    if (j < nJ-1) j <= j + 1;
    else begin
        j <= 0;
        if (i+1 == nI) i <= i + 1;
        else      done <= True;
    end
endrule
```
An imperative view

It is easy to write a sequence in C

```c
void doTest(){
    makeReq(17.23.12.225);
    getResp();
    makeReq(17.23.12.25);
    getResp();
    exit(0);
}
```

Writing this in rules is tedious:

Can we just write the actions and have the compiler make the rules?

```c
seq
    makeReq(17.23.12.225);
    getResp();
    makeReq(17.23.12.25);
    getResp();
$finish();
endseq;
```

From Action Lists to FSMs

▶ FSM interface

```c
interface FSM:
    method Action start();
    method Bool done();
endinterface
```

▶ Creating an FSM

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

March 10, 2010
http://csg.csail.mit.edu/6.375
The Stmt Sublanguage

Stmt =
  <Bluespec Action>
  | seq s1..sN endseq
  | par s1..sN endpar
  | if-then / if-then-else
  | for-, while-, repeat(n)-
    (w/ break and continues)

Translation Example:
Seq to FSM

Stmt s = seq
  makeReq(17.23.12.225);
  getResp();
  makeReq(17.23.12.25);
  getResp();
  $finish();
endseq;

FSM f <- mkFSM(s);

module mkFSM_s(FSM)
  Reg#(Bit#(3)) pos <- mkReg(0);
  rule step1(pos==1);
    makeReq(17.23.12.225); pos <= 2;
  endrule
  rule step2(pos==2);
    getResp(); pos <= 3; endrule
  rule step3(pos==3);
    makeReq(17.23.12.25); pos <= 4;
  endrule
  rule step4(pos==4);
    getResp(); pos <= 5; endrule
  rule step5(pos==5);
    $finish; pos <= 0; endrule
method Action start() if(pos==0);
  pos <= 1;
endmethod
method Bool done() return (pos == 0);
endmethod
endmodule
Parallel Tasks

seq
refReq(x);
refRes(rReg);
dutReq(x);
dutRes(dReg);
checkMatch(rReg,dReg);
endseq

We want to check
dut and ref have
same result

Do each, then
check results

But it doesn’t matter that ref finishes before dut starts...


Start ref and dut at the same
time

Seq. for each
implementation

Start together

Both run at own
rate

Wait until both
are done

seq
par

seq  refReq(x);
refRes(refv);endseq

seq  dutReq(x);
dutRes(dutv); endseq

endpar

checkMatch(refv,dutv);
endseq

March 10, 2010  http://csg.csail.mit.edu/6.375  L11-14
What exactly is the translation?

- The Stmt sublanguage is clearer for the designer; but, what FSM do we get?

- Let’s examine each Stmt Construction case and see how it can be implemented.

Base Case: Primitive Action: `a`

```plaintext
Reg#(Bool) doneR <- mkReg(True);
rule dowork(!doneR);
    a;
    doneR <= True;
endrule
method Action start() if (doneR);
    doneR <= False;
endmethod

method Bool done(); return doneR; endmethod
```
Sequential List - \textit{seq}

\textbf{seq} \textit{s1...sN} \textbf{endseq}: sequential composition

\begin{verbatim}
Reg#(int) s <- mkReg(0);
FSM s1 <- mkFSM (s1); ... ; FSM sN <- mkFSM (sN);
Bool flag = s1.done() && ... && sN.done();

rule one (s==1); s1.start(); s <= 2; endrule
rule two (s==2&& s1.done());
    s2.start(); s <= 3; endrule

rule n (s==n && sN-1.done());
    sN.start(); s <= 0; endrule

method Action start() if (flag); s <= 1; endmethod
method Bool done(); return flag; endmethod
\end{verbatim}

Implementation - \textit{par}

\textbf{par} \textit{s1...sN} \textbf{endpar}: parallel composition

\begin{verbatim}
FSM s1 <- mkFSM (s1); ... ; FSM sN <- mkFSM (sN);
Bool flag = s1.done() && ... && sN.done();

method Action start() if (flag);
    s1.start(); s2.start(); ...; sN.start();
endmethod
method Bool done(); return flag; endmethod
\end{verbatim}
Implementation - \textit{if}

\textit{if} \ p \ \textit{then} \ sT \ \textit{else} \ sF: \ conditional \ composition

\begin{verbatim}
FSM sT <- mkFSM (sT); FSM sF <- mkFSM (sF);

Bool flag = sT.done() && sF.done();

method Action start() if (flag);
    if (p) then sT.start() else sF.start();
endmethod

method Bool done(); return flag; endmethod
\end{verbatim}

Implementation - \textit{while}

\textit{while} \ p \ \textit{do} \ s: \ loop \ composition

\begin{verbatim}
s <- mkFSM(s);
Reg#(Bool) busy <- mkReg(False);
Bool flag = !busy;

rule restart_loop(busy && s.done());
    if (p) begin s.start(); busy <= True;
    else busy <= False;
endrule

method Action start() if (flag);
    if (p) begin s.start(); busy <= True;
    else busy <= False;
endmethod

method Bool done(); return flag; endmethod
\end{verbatim}
The StmtFSM library

- This is the Library (almost)
  - Some optimizations for seq/base case
  - Stmt syntax added for readability
- Good but not great HW (users can do better by handcoding)
  - state-encoding
    - Use a single wide register \((i,j)\) instead of two
    - Use 1 \(\log(n)\)-bit register instead of \(n\) 1-bit registers
    - See if state can be inferred from other data registers
  - Unnecessary dead cycles can be eliminated

FSM atomicity

- FSM Actions are made into rules
  - rule atomicity governs statement interactions

Stmt \(s_1 = \text{seq}
\begin{align*}
  \text{action } f_1.\text{enq}(x); & f_2.\text{enq}(x); \\
  & \text{endaction}
\end{align*}
\begin{align*}
  \text{action } f_1.\text{deq}(); & x=x+1; \\
  & \text{endaction}
\end{align*}
\begin{align*}
  \text{action } f_2.\text{deq}(); & y=y+1; \\
  & \text{endaction}
\end{align*}
\text{endseq;}

Stmt \(s_2 = \text{seq}
\begin{align*}
  \text{action } f_1.\text{enq}(y); & f_2.\text{enq}(y); \\
  & \text{endaction}
\end{align*}
\begin{align*}
  \text{action } f_1.\text{deq}(); & \$\text{display}("\%d", y); \\
  & \text{endaction}
\end{align*}
\begin{align*}
  \text{action } f_2.\text{deq}(); & \$\text{display}("\%d", x); \\
  & \text{endaction}
\end{align*}
\text{endseq;}

rule \(s_1(\ldots); f_1.\text{enq}(x); f_2.\text{enq}(x); \ldots; \text{endrule}
rule \(s_2(\ldots); f_1.\text{deq}(); x=x+1; \ldots \text{endrule}
rule \(s_3(\ldots); f_2.\text{deq}(); y=y+1; \ldots \text{endrule}

rule \(s_1(\ldots); f_1.\text{enq}(y); f_2.\text{enq}(y); \ldots \text{endrule}
rule \(s_2(\ldots); f_1.\text{deq}(); \$\text{display}("\%d", y); \ldots \text{endrule}
rule \(s_3(\ldots); f_2.\text{deq}(); y=y+1; \ldots \text{endrule}

FSM Atomicity

- We’re writing actions, not rules
  - Do they execute atomically?

- Seq. Stmt
  - Only one at a time
  \[ \Rightarrow \]

- Par. Stmt
  - all at once
  \[ \Rightarrow \]

What happens here?

FSM summary

- Stmt sublanguage captures certain common and useful FSM idioms:
  - sequencing, parallel, conditional, iteration

- FSM modules automatically implement Stmt specs

- FSM interface permits composition of FSMs

- Most importantly, same Rule semantics
  - Actions in FSMs are atomic
  - Actions automatically block on implicit conditions
  - Parallel actions, (in the same FSM or different FSMs) automatically arbitrated safely (based on rule atomicity)