BSV Training

Section: Controlling Rule Scheduling

Attributes, rules scheduling attributes

www.bluespec.com

Controlling rule scheduling

Recall:

Pure rule semantics are non-deterministic (recall from lecture L2):

```
while (at least one rule’s condition is ready)
    choose any such rule
    perform its body action(s)
```

When bsc compiles BSV rules to Verilog, it produces control logic that allows multiple enabled rules to fire, in a specific order:

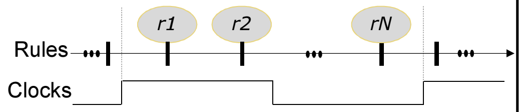
- i.e., bsc eliminates non-determinism by making *static choices in rule ordering*, and generating control logic accordingly

The user can force *bsc*’s choices with various *attributes* to control scheduling

- BSV attribute syntax (same as SystemVerilog) : (* *attribute* = “*rule and method names*” *)
- These are written in a module, typically just before the rules mentioned in the attribute.
- Since methods are just rule fragments, these attributes can mention methods as well

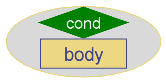
A refinement on rule ordering: urgency and earliness

Consider a set of concurrent rules (rules enabled because their rule conditions are true, and that fire in a clock in a logical order):

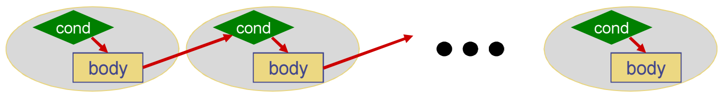


Executing each rule involves two separate activities:

- evaluate its condition
- evaluate its body



Thus, when executing a rule sequence, we're conceptually alternating these activities:



But note:

- Rule conditions are pure boolean expressions. They have no side effects. Thus, just evaluating the rule condition of rule rA can never affect another rule rB (neither evaluating rB's condition, nor what rB's body does).
- Many rule bodies do not affect other rule conditions

Thus, we can reorder condition and body evaluations (provided we preserve required orderings)

Thus, rule orderings can be refined into two orderings:

- *Urgency*: order in which we evaluate rule conditions
- *Execution*: order in which we evaluate rule bodies (technically this is original rule ordering), also called *Earliness*

Controlling scheduling: rule urgency

```
(* descending_urgency = "r1, r2" *)

rule r1 (c1);
   fifo.enq (e1);  // one enq per cycle
endrule

rule r2 (c2);
   fifo.enq (e2);  // one enq per cycle
endrule
```

- Urgency is the order/priority in which WILL_FIRE conditions are computed
- In this example, r1 and r2 conflict because of 'fifo.enq()'
  The scheduler will not even consider r2's condition if r1 WILL_FIRE
- If the user does not specify urgency between conflicting rules, the compiler will pick an urgency order and notify the user
  - Note: This makes execution completely deterministic (non-determinism in the logical semantics is eliminated)

Controlling scheduling: rule execution order

```
(* execution_order = “r1, r2" * )

rule r1;
   x <= 5;
endrule

rule r2;
   y <= 6;
endrule
```

- Forces an ordering in the logical semantics
  - Execution order is the same as the ordering in the original pure rule semantics
  - Execution order is also called “earliness”

- In this example, forces “r1 < r2”

Urgency and Execution orders may be different

```
(* descending_urgency="enq_item, enq_bubble" *)
rule enq_item;
  outfifo.enq(infifo.first); infifo.deq;
  bubbles <= 0;
endrule

rule inc_bubbles;
  bubbles <= bubbles + 1;
endrule

rule enq_bubble;
  outfifo.enq(bubble_value);
  max_bubbles <= max (max_bubbles, bubbles);
endrule
```

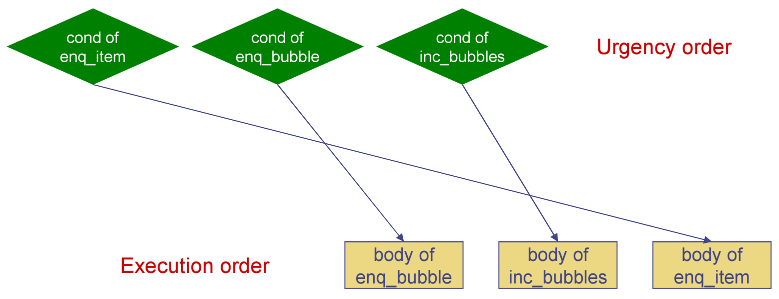
- This code enqueues items from the infifo into the outfifo, if available; otherwise, it enqueues a ‘bubble_value’
- It also computes the maximum stretch of bubbles

- The execution order is: enq_bubble < inc_bubbles < enq_item because reads of ‘bubbles’ must precede writes of ‘bubbles’

- However, we have forced the urgency to be: enq_item < enq_bubble

Urgency and Execution orders may be different

Pictorially:



Controlling scheduling: rule preempts

```
(* preempts = "r1, r2" *)

rule r1 (upA);
   x <= x + 3;
endrule

rule r2;
   y <= y + 1;
endrule
```

- This forces one rule's firing to suppress another rule's firing
- In this example, whenever r1 fires the scheduler will suppress r2 (even if its conditions are otherwise true)
  - For example, here y counts "idle cycles" of r1

Controlling scheduling: rules mutually exclusive

```
(* mutually_exclusive = "updateBit0, updateBit1" *)

rule updateBit0 (oneHotNumber[0] == 1);
  x[0] <= 1;
endrule

rule updateBit1 (oneHotNumber[1] == 1);
  x[1] <= 1;
endrule
```

- Asserts to the compiler that two rules' conditions are mutually exclusive (will never simultaneously be true in any clock)
- When rules are mutually exclusive, the compiler can generate better HW
  - E.g., simple muxes instead of priority muxes
- The compiler does sophisticated Boolean analysis to try to prove that two rule conditions are mutually exclusive
  - However, this question is undecidable in general. E.g.,
    - The conditions depend on external inputs
    - Mutual exclusivity depends on application-specific semantic knowledge (such as "one-hotness" of a bit-vector)
  - This assertion helps the compiler, when it can't decide
  - For simulation, the compiler also generates code to verify mutual exclusivity

bluespec

End

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