Serializability of Concurrent Execution of Rules

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Linearizability

- *Rule atomicity* says that execution no other rule appears to be interleaved with the execution of a rule.
- This is also known as *linearizability*, i.e., other rules appear to execute before or after a given rule.
- The following example has no double write error but concurrent execution of `ra` and `rb` violates linearizability.

```plaintext
rule ra;
    x <= y+1;
endrule
rule rb;
    y <= x+2;
endrule
```

Initially `x=0, y=0`

Final value of `(x,y)`

- `ra ; rb` → `(1,3)`
- `rb ; ra` → `(3,2)` ≠
- Concurrent Execution → `(1,2)` ≠

We say rules `ra` and `rb` conflict and should not be executed concurrently.
Serializability

- We could say that the concurrent execution of rules or methods is allowed as long as no double write error is possible and the execution is linearizable.
- In fact, we impose the additional constraint of *serializability* on the concurrent execution of rules:

|
| **Serializability** means that a concurrent execution of rules must match some serial execution of rules, aka one-rule-at-a-time execution of rules |

- The serializability constrain is imposed to make it easier to analyze the behavior of concurrent systems; it is a common and well established practice all distributed systems and databases.

  |
  | **In the hardware domain the idea of serializability is new at the design level but it has been used extensively in proving properties of designs** |
An example to illustrate Serializability

**Rule ra**

\[
\begin{align*}
x & \leq y + 1; \\
\text{endrule}
\end{align*}
\]

**Rule rb**

\[
\begin{align*}
y & \leq z + 2; \\
\text{endrule}
\end{align*}
\]

**Rule rc**

\[
\begin{align*}
z & \leq x + 3; \\
\text{endrule}
\end{align*}
\]

Initially

\[
\begin{align*}
x = 0, y = 0, z = 0
\end{align*}
\]

Any two rules can be executed concurrently but not all three

\[
\begin{align*}
&\text{Sequential Executions} \\
&\begin{align*}
\text{ra} & ; \text{rb} & ; \text{rc} & \quad (1,0,0) \rightarrow (1,2,0) \rightarrow (1,2,4) \\
\text{ra} & ; \text{rc} & ; \text{rb} & \quad (1,0,0) \rightarrow (1,0,4) \rightarrow (1,6,4) \\
\text{rb} & ; \text{rc} & ; \text{ra} & \quad (0,2,0) \rightarrow (0,2,3) \rightarrow (3,2,3) \\
\text{rb} & ; \text{ra} & ; \text{rc} & \quad (0,2,0) \rightarrow (3,2,0) \rightarrow (3,2,6) \\
\text{rc} & ; \text{ra} & ; \text{rb} & \quad (0,0,3) \rightarrow (1,0,3) \rightarrow (1,5,3) \\
\text{rc} & ; \text{rb} & ; \text{ra} & \quad (0,0,3) \rightarrow (0,5,3) \rightarrow (6,5,3)
\end{align*}
\end{align*}
\]

**Parallel Executions**

\[
\begin{align*}
&\begin{align*}
\text{ra} & | \text{rb} & | \text{rc} & \quad (1,2,3) \\
(\text{ra} & | \text{rb}) & ; \text{rc} & \quad (1,2,0) \rightarrow (1,2,4) \\
\text{rc} & ; (\text{ra} & | \text{rb}) & \quad (0,0,3) \rightarrow (1,5,3) \\
\text{ra} & ; (\text{rb} & | \text{rc}) & \quad (1,0,0) \rightarrow (1,2,4) \\
(\text{rb} & | \text{rc}) & ; \text{ra} & \quad (0,2,3) \rightarrow (3,2,3) \\
\text{rb} & ; (\text{ra} & | \text{rc}) & \quad (0,2,0) \rightarrow (3,2,3) \\
(\text{ra} & | \text{rc}) & ; \text{rb} & \quad (1,0,3) \rightarrow (1,5,3)
\end{align*}
\end{align*}
\]

Not allowed

\[
\begin{align*}
\text{ra} & < \text{rb} \\
\text{rb} & < \text{rc} \\
\text{rc} & < \text{ra}
\end{align*}
\]

Notice \(\text{ra} < \text{rb}, \text{rb} < \text{rc}\) does not imply that \(\text{ra} < \text{rc}\)
Why is serializability important?

- As you have seen it is straightforward to build hardware so that ra and rb will execute concurrently. However, in general, it is difficult to derive the behavior of the resulting circuit.
- Serializability, lets us apply one rule at a time in some order to derive the behavior of the composite system.
- Without serializability, the atomicity of each rule has no meaning in a complex system.
- Even though serializability imposes an additional constraint, and will make us reject some RTL implementations for a Bluespec design, in practice its advantages far outweigh its disadvantages in debugging and verification.

```plaintext
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= x+2;
endrule
```

Initially (x,y) are (0,0)

Concurrent Execution

ra < rb (1,3)

rb < ra (3,2)
Conflict Matrix

- We can define a Conflict Matrix (CM), which specifies for a given pair of methods, or a pair of rules, or a method and rule, the effect of concurrent execution
  - $\text{ra} < \text{rb}$: $\text{ra}$ and $\text{rb}$ can be executed concurrently; the net effect is as if $\text{ra}$ executed before $\text{rb}$
  - $\text{ra CF rb}$: $\text{ra}$ and $\text{rb}$ can be executed concurrently; the net effect is the same as ($\text{ra} < \text{rb}$) and ($\text{rb} < \text{ra}$)
  - $\text{ra C rb}$: $\text{ra}$ and $\text{rb}$ Conflict; either the concurrent execution will cause a *double-write* error or the resulting effect is neither ($\text{ra} < \text{rb}$) nor ($\text{rb} < \text{ra}$)
  - $\text{ra ME rb}$: the guards of $\text{ra}$ and $\text{rb}$ are mutually exclusive and thus, $\text{ra}$ and $\text{rb}$ can never be *rdy* together
The derivation of CM

- There is a natural ordering between the values of CM entries

  \[ CF = \{<,>\} \]

  \[ \{<\} \quad \{>\} \]

  \[ C = \{\} \]

- This ordering permits us to take intersections of conflict information, e.g.,
  - \( \{>\} \cap \{<,>\} = \{>\} \)
  - \( \{>\} \cap \{<\} = \{\} \)

- We use the CM of primitive modules (register, EHR) to derive the CM for the interface methods of a module
Deriving the Conflict Matrix (CM) of a module interface

- Let \( g_1 \) and \( g_2 \) be the two methods defined by a module, such that
  
  \[
  \text{mcalls}(g_1) = \{g_{11}, g_{12}, \ldots, g_{1n}\}
  \]
  
  \[
  \text{mcalls}(g_2) = \{g_{21}, g_{22}, \ldots, g_{2m}\}
  \]

  \[
  \text{CM}[g_1, g_2] = \text{conflict}(g_{11}, g_{21}) \cap \text{conflict}(g_{11}, g_{22}) \cap \ldots \cap \text{conflict}(g_{12}, g_{21}) \cap \text{conflict}(g_{12}, g_{22}) \ldots \cap \text{conflict}(g_{1n}, g_{21}) \cap \text{conflict}(g_{1n}, g_{22}) \cap \ldots
  \]

  \[
  \text{conflict}(x, y) = \begin{cases} 
  \text{CM}[x, y] & \text{if } x \text{ and } y \text{ are methods of the same module} \\
  \text{CF} & \text{else}
  \end{cases}
  \]

Compiler can derive the CM for a module by starting with the innermost modules in the module instantiation tree
The conflict between two rules or a rule and a method can be derived in a similar manner by examining the CM properties of the constituent method calls.

### Example 1

```plaintext
rule ra;
    x <= x+1;
endrule
rule rb;
    y <= y+2;
endrule
```

<table>
<thead>
<tr>
<th></th>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>C</td>
<td>CF</td>
</tr>
<tr>
<td>rb</td>
<td>CF</td>
<td>C</td>
</tr>
</tbody>
</table>

### Example 2

```plaintext
rule ra;
    x <= y+1;
endrule
rule rb;
    y <= x+2;
endrule
```

<table>
<thead>
<tr>
<th></th>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>rb</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

### Example 3

```plaintext
rule ra;
    x <= y+1;
endrule
rule rb;
    y <= y+2;
endrule
```

<table>
<thead>
<tr>
<th></th>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>C</td>
<td>&lt;</td>
</tr>
<tr>
<td>rb</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>
Example 1: Compiler Analysis

```
rule ra;
  x <= x+1;
endrule
rule rb;
  y <= y+2;
endrule
```

mcalls(ra) = \{x.w, x.r\}
mcalls(rb) = \{y.w, y.r\}

CM(ra, rb) =
  conflict(x.w, y.w) ∩ conflict(x.w, y.r)
  ∩ conflict(x.r, y.w) ∩ conflict(x.r, y.r)
  = CF ∩ CF ∩ CF ∩ CF
  = CF

Rules ra and rb can be scheduled together without violating the one-rule-at-a-time-semantics. We say rules ra and rb are CF
Example 2: Compiler Analysis

```plaintext
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= x+2;
endrule
```

\[
\text{mcalls(ra)} = \{x.w, y.r\} \\
\text{mcalls(rb)} = \{y.w, x.r\}
\]

\[
\text{CM(ra, rb)} = \text{conflict(x.w, y.w)} \cap \text{conflict(x.w, x.r)} \\
\cap \text{conflict(y.r, y.w)} \cap \text{conflict(y.r, x.r)} \\
= \text{CF} \cap \{>\} \cap \{<\} \cap \text{CF} \\
= C
\]

Rules ra and rb cannot be scheduled together without violating the one-rule-at-a-time semantics. Rules ra and rb Conflict
Example 3: Compiler Analysis

**Rule ra**;

\[ x \leq y+1; \]

**endrule**

**Rule rb**;

\[ y \leq y+2; \]

**endrule**

\[
\text{mcalls}(ra) = \{x.w, y.r\} \\
\text{mcalls}(rb) = \{y.w, y.r\}
\]

\[
\text{CM}(ra, rb) = \\
\text{conflict}(x.w, y.w) \cap \text{conflict}(x.w, y.r) \\
\cap \text{conflict}(y.r, y.w) \cap \text{conflict}(y.r, y.r) \\
= \text{CF} \cap \text{CF} \cap \{<\} \cap \text{CF} \\
= \{<\}
\]

Rules ra and rb can be scheduled together without violating the one-rule-at-a-time-semantics.

**Rule ra < rb**
Two-Element FIFO
Deriving the CM

method Action enq(t x) if (!vb);
    if (va) begin db <= x; vb <= True; end
    else begin da <= x; va <= True; end
endmethod
method Action deq if (va);
    if (vb) begin da <= db; vb <= False; end
    else begin va <= False; end
endmethod

We can derive a conservative CM by ignoring the conditionals

mcalls(enq) = {vb.r, va.r, db.w, vb.w, da.w, va.w}
mcalls(deq) = {va.r, vb.r, da.w, db.r, vb.w, va.w}

\[
\text{CM[enq, deq]} = \text{CM[vb.r, va.r]} \cap \text{CM[vb.r, vb.r]} \cap \text{CM[vb.r, db.w]} \cap \text{CM[vb.r, db.r]} \cap \text{CM[vb.r, vb.w]} \cap \text{CM[vb.r, va.w]} \cap \text{CM[va.r, va.r]} \cap \text{CM[va.r, vb.r]} \cap \text{CM[va.r, da.w]} \cap \text{CM[va.r, db.w]} \cap \text{CM[va.r, db.r]} \cap \text{CM[va.r, vb.w]} \cap \text{CM[va.r, va.w]} \cap \text{CM[db.w, va.r]} \cap \text{CM[db.w, vb.r]} \cap \text{CM[db.w, da.w]} \cap \text{CM[db.w, db.w]} \cap \text{CM[db.w, db.r]} \cap \text{CM[db.w, vb.w]} \cap \text{CM[db.w, va.w]} \cap \text{CM[vb.w, va.r]} \cap \text{CM[vb.w, vb.r]} \cap \text{CM[vb.w, da.w]} \cap \text{CM[vb.w, db.w]} \cap \text{CM[vb.w, db.r]} \cap \text{CM[vb.w, vb.w]} \cap \text{CM[vb.w, va.w]} \cap \text{CM[da.w, va.r]} \cap \text{CM[da.w, vb.r]} \cap \text{CM[da.w, da.w]} \cap \text{CM[da.w, db.w]} \cap \text{CM[da.w, db.r]} \cap \text{CM[da.w, vb.w]} \cap \text{CM[da.w, va.w]} \cap \text{CM[va.w, va.r]} \cap \text{CM[va.w, vb.r]} \cap \text{CM[va.w, da.w]} \cap \text{CM[va.w, db.w]} \cap \text{CM[va.w, db.r]} \cap \text{CM[va.w, vb.w]} \cap \text{CM[va.w, va.w]} \cap \text{CF} \cap \{<\} \cap \text{CF} \cap \{<\} \cap \{>\} \cap \{>\} \cap \text{C} \cap \text{C} \cap \text{C} \cap \{>\} \cap \text{C} = \text{CF}
\]

September 23, 2019
http://csg.csail.mit.edu/6.375
Two-Element FIFO

More accurate analysis

For more accurate analysis we should consider the conditions under which both rules will be ready, i.e., va = True and vb = False

Thus, enq and deq do not conflict but the BSV compiler is unable to deduce this
Using the CM to Enforce Serializability

- Suppose we are given a “rule ordering”, that is, our preference about the behavior we would like to see given a set of rules
- We can keep scheduling the rules in that order, and if we find a conflict with an already scheduled rule we skip that rule and go to the next one
- Mathematically, given the list \( \{r_1, r_2, \ldots, r_n\} \)
  
  \[
  \text{will-fire}(r_i) = \text{can-fire}(r_i) \&\& \\
  \forall_{k<i} \{\text{if will-fire}(r_k) \text{ then } (\text{CM}[r_k, r_i] \cap \{<\}) = \{<\}\} \\
  = \text{can-fire}(r_i) \&\& \\
  \forall_{k<i} \{\neg \text{will-fire}(r_k) \text{ || } (\text{CM}[r_k, r_i] \cap \{<\}) = \{<\}\}
  \]
An example schedule

```plaintext
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= z+2;
endrule
rule rc;
  z <= x+3;
endrule

- CM = {(ra<rb), (rb<rc), (rc<ra)}
- Scheduling priority = {ra,rb,rc}

will-fire(r_i) = can-fire(r_i) &&
\forall_{k<i} \{!will-fire(r_k) || (CM[r_k, r_i] \cap \{<\}) = \{<\}\}

- can-fire(r_k) is true for all the rules every cycle

- will-fire(ra) = can-fire(ra)
- will-fire(rb) = can-fire(rb) &&
  (!can-fire(ra) || ((CM[ra, rb] \cap \{<\}) = \{<\}))
  = can-fire(rb) &&((! can-fire(ra)) || True)
  = can-fire(rb)
- will-fire(rc) = can-fire(rc) &&
  {(!can-fire(ra)) || (CM[ra, rc] \cap \{<\}) = \{<\})
  && (!can-fire(rb) || (CM[rb, rc] \cap \{<\}) = \{<\}))
  = can-fire(rc) &&
  {(!can-fire(ra))||False)) && (!can-fire(rb)||True)}
  = can-fire(rc) && !can-fire(ra)
```
Synthesis of the scheduler

- \( CM = \{(ra<rb), (rb<rc), (rc<ra)\}\)
- Scheduling priority = \{ra,rb,rc\}

will_fire(ra) = can_fire(ra)
will_fire(rb) = can_fire(rb)
will_fire(rc) = can_fire(rc) && !can_fire(ra)

Since in Ex 4, all canFire signals are true, the scheduler gets simplified to (True,True,False)
Slightly modified example

```plaintext
rule ra if (y==1);
  x <= y+1;
endrule
rule r;
  y <= z+2;
endrule
rule rc;
  z <= x+3;
endrule

- CM = {(ra<rb), (rb<rc), (rc<ra)}
- Scheduling priority = {ra, rb, rc}

will-fire(r_i) = can-fire(r_i) &
\forall_{k<i} \{ !will-fire(r_k) || (CM[r_k, r_i] \cap \{<\}) = \{<\} \}

- can-fire(r_k) is true for all the rules every cycle

- will-fire(ra) = can-fire(ra)
- will-fire(rb) = can-fire(rb)
- will-fire(rc) = can-fire(rc) && !can-fire(ra)
```

Simplification for Ex 5 will lead to

- will-fire(ra) = (y==1)
- will-fire(rb) = True
- will-fire(rc) = !(y==1)
Preserving atomicity while preventing a rule from firing

```
rule ra;
  x <= e1; y <= e2;
endmethod
rule rb;
  x <= e3; z <= e4;
endmethod
```

- ra and rb conflict because of a double write in x
- Suppose we want to prevent rb from firing

What is wrong with this circuit?

The atomicity of rule rb is violated: y may be updated without x being updated!

fix?
Preserving atomicity while preventing a rule from firing

```plaintext
rule ra;
  x <= e1; y <= e2;
endmethod
rule rb;
  x <= e3; z <= e4;
endmethod
```

What is wrong with this circuit?

The atomicity of rule rb is violated: y may be updated without x being updated!

fix?

When we do not want a rule to fire all its state updates must be stopped

- ra and rb conflict because of a double write in x
- Suppose we want to prevent rb from firing
A general method for inhibiting rule execution

rule1
body
guard
rdy

rule2
body
guard
rdy

m1.f1

m2.f2
A general method for inhibiting rule execution
A general method for inhibiting rule execution

- We introduce a scheduler to control which rules among the ready rules should execute
  - We feed it the \textit{rdy} signals of all the rules
We introduce a scheduler to control which rules among the ready rules should execute.

- We feed it the `rdy` signals of all the rules.
- The scheduler lets only *non-conflicting* rules proceed.
- It turns off some of the “can fire” signals.
A general method for inhibiting rule execution

- We introduce a scheduler to control which rules among the ready rules should execute
  - We feed it the rdy signals of all the rules
- The scheduler lets only non-conflicting rules proceed
  - It turns off some of the “can fire” signals

Scheduler is a pure combinational circuit with a small number of gates
What is inside the scheduler

- Suppose rules ra and rb can be executed concurrently – no double write
  - Scheduler

```plaintext
ra.canFire  ra.willFire

rb.canFire  rb.willFire
```
What is inside the scheduler

Suppose rules ra and rb should not be executed concurrently

Schedule 1: rule ra has priority, i.e., if ra can fire rb will not fire

Schedule 2: rule rb has priority, i.e., if rb can fire ra will not fire

The choice is specified by scheduling annotations in the BSV program
Combinational cycles

- Rules containing following types of actions will be rejected by the BSV compiler because they are meaningless and will generate combinational cycles
  - x[0] <= x[1]
  - x[0] <= y[1]; y[0] <= x[1]
  - if (x[1]) x[0] <= e;
Takeaway

- One-rule-at-a-time semantics are important to understand the legal behaviors of a system.
- Efficient hardware for multi-rule system requires that many rules execute in parallel without violating the one-rule-at-time semantics.
- BSV compiler builds a scheduler circuit to execute as many rules as possible concurrently.
  - It takes user advice in scheduling conflicting rules.
- For high-performance designs we have to worry about the CM characteristics of our modules.