Concurrent properties of BSV methods and rules

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

One-rule-at-a-time semantics

Given a program with a set of rules \( \{ \text{rule } r_i \ a_i \} \) and an initial state \( S_0 \), \( S \) is a legal state if and only if there exists a sequence of rules \( r_{j1}, \ldots, r_{jn} \) such that \( S = a_{jn}(\ldots(a_{j1}(S_0))\ldots) \)
Concurrent execution of two rules

Concurrent execution of two rules, rule \( r_1 \) \( a_1 \) and rule \( r_2 \) \( a_2 \), means executing a rule whose body looks like \( (a_1; a_2) \), that is a rule which is a parallel composition of the actions of the two rules with the following restrictions to preserve the one-rule-at-a-time semantics:

- Either \( \forall S. (a_1; a_2)(S) = a_2(a_1(S)) \)
- or \( \forall S. (a_1; a_2)(S) = a_1(a_2(S)) \)

Concurrent scheduling of rules

Rule \( r_1 \) \( a_1 \) to rule \( r_n \) \( a_n \) can be scheduled concurrently, preserving one-rule-at-a-time semantics, if and only if there exists a permutation \( (p_1, \ldots, p_n) \) of \( (1, \ldots, n) \) such that

\[ \forall S. (a_1; \ldots; a_n)(S) = a_{p_n}(\ldots(a_{p_1}(S))) \]

How does a compiler decide which rules can be scheduled concurrently

Related question: what is a legal rule?
Well formed actions (rules)

Informally

- No possibility of \textit{double write error}. In general, no double use of a method
  - The only exception is a value method without arguments, e.g., register read, fifo.first
- No \textit{combinational cycles}. In general it means that it should be possible to put all the method calls in a sequential order consistent with their module definitions and data dependences

Are these actions legal?

- \[ x \leftarrow e_1; x \leftarrow e_2; \]
- \[ x \leftarrow e_1; \text{if}(p) x \leftarrow e_2; \]
- \[ \text{if}(p) x \leftarrow e_1; \text{else} x \leftarrow e_2; \]
- \[ x[0] \leftarrow x[1] \]
- \[ x[0] \leftarrow y[1]; y[0] \leftarrow x[1] \]
- \[ \text{if}(x[1]) x[0] \leftarrow e; \]
A critical example

Example 1: Rule exchange \( x \leq y; y \leq x \)
Example 2: Rule exchange’ \( f(); g() \)
   where Module foo
      register \( x, y \) etc
      method \( f() = (x \leq y); \)
      method \( g() = (y \leq x); \)

Example 3:
   Rule \( fr; \ x \leq y; \)
   Rule \( gr; \ y \leq x; \)

Is exchange legal?
Is exchange’ legal?
Can rules \( fr \) and \( gr \) be executed together?

Primitive module: Register

- read and write can happen in the same atomic action and don’t affect each other; the effect of write is not visible until the atomic action has completed
- Legality of an action depends upon the permitted intra-rule behaviors
- Concurrent scheduling depends upon the inter-rule behavior

Legality of an action depends upon the permitted intra-rule behaviors

Legality of an action depends upon the permitted inter-rule behaviors

Intra-rule behavior

\[
\begin{array}{c|c|c}
\text{reg.r} & \text{reg.w} \\
\hline
\text{reg.r} & \text{CF} & \text{CF} \\
\text{reg.w} & \text{CF} & \text{C} \\
\end{array}
\]

Inter-rule behavior

\[
\begin{array}{c|c|c}
\text{reg.r} & \text{reg.w} \\
\hline
\text{reg.r} & \text{CF} & < \\
\text{reg.w} & > & \text{C} \\
\end{array}
\]
**Intra-rule behavior**

<table>
<thead>
<tr>
<th></th>
<th>EHR.r0</th>
<th>EHR.w0</th>
<th>EHR.r1</th>
<th>EHR.w1</th>
</tr>
</thead>
<tbody>
<tr>
<td>EHR.r0</td>
<td>CF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EHR.w0</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EHR.r1</td>
<td></td>
<td>CF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EHR.w1</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

**Inter-rule behavior**

<table>
<thead>
<tr>
<th></th>
<th>EHR.r0</th>
<th>EHR.w0</th>
<th>EHR.r1</th>
<th>EHR.w1</th>
</tr>
</thead>
<tbody>
<tr>
<td>EHR.r0</td>
<td>CF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EHR.w0</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EHR.r1</td>
<td></td>
<td>CF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EHR.w1</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

---

**Intra-rule analysis**

**“Happens before” (\(<\) relation**

- “happens before” relation between the methods of a module governs how the methods behave when called by a rule, action, method or exp
  - \( f < g \) : f happens before g
    - (g cannot affect f within an action)
  - \( f > g \) : g happens before f
  - C : f and g conflict and cannot be called together
  - CF : f and g are conflict free and do not affect each other

- This relation is defined as a conflict matrix (CM) for the methods of primitive modules like registers and EHRs and derived for the methods of all other modules
Inter-rule analysis

“Happens before” (\(<\)) relation

- “happens before” relation between the methods of a module governs how the methods behave when called by two rules or methods
  - \(f < g\) : parallel: but \(g\) may affect \(f\) in a seq execution
  - \(f > g\) : parallel: but \(f\) may affect \(g\) in a seq execution
  - \(C\) : not parallel: \(f\) and \(g\) conflict
  - \(CF\) : parallel: \(f\) and \(g\) are conflict free and unrelated

- This relation is defined as a conflict matrix (CM) for the methods of primitive modules like registers and EHRs and derived for the methods of all other modules

Conflict ordering

\[
\begin{align*}
\text{CF} &= \{<,>\} \\
\{<\} & \cap \{<,>\} = \{>\} \\
\{>\} & \cap \{<\} = \{\}
\end{align*}
\]

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

- \( \text{mcalls}(x) \) is the set of method called by \( x \)
- \( \text{mcalls}(x) <_s \text{mcalls}(y) \) means
  \[
  a \in \text{mcalls}(x), b \in \text{mcalls}(y) \Rightarrow (a < b) \mid (a \ CF b) \mid (a \ ME b)
  \]
- we often overload \( < \) and use it in place of \( <_s \)

Deriving the Conflict Matrix (CM) of a module

- Let \( g_1 \) and \( g_2 \) be the two methods defined by a module, such that
  \[
  \text{mcalls}(g_1) = \{g_{11}, g_{12} \ldots g_{1n}\}
  \]
  \[
  \text{mcalls}(g_2) = \{g_{21}, g_{22} \ldots g_{2m}\}
  \]
- \( \text{conflict}(x, y) = \) if \( x \) and \( y \) are methods of the same module then CM\([x, y]\) else CF

Derivation

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

Compiler can derive the CM for a module by starting with the innermost modules in the module instantiation tree.
Data-dependence constraints

- if (e) a ⇒ NewDependences(mcalls(e), mcalls(a))
- m.g(e) ⇒ NewDependences(mcalls(e), {m.g})
- t = e ; a ⇒ NewDependences(mcalls(e),
  \{f \mid f \in mcalls(a) \& f \text{ uses } t\})

An action is legal if the data-dependence imposed constraints together with method definition constraints can be placed in a total order (no cycles)

Real legal-rule analysis is more complicated: Predicated calls

- The analysis we presented would reject the following rule because of method conflicts
  \text{if (p) m.g(e1) ; if (!p) m.g(e2)}
- We need to keep track of the predicates associated with each method call
  m.g is called with predicates p and !p
  which are disjoint – therefore no conflict
Mutually exclusive actions

- a1 and a2 are *mutually exclusive* if in all possible states the effect of one of them is “no action”
  - example: a1 = if (p) b1           a2 = if (!p) b2
- Mutual exclusivity of actions and methods reduces the number of conflicts at the cost of complicating the analysis
- In computing the conflict matrix (CM) one can ignore entries corresponding to mutually exclusive methods
- In determining the legality of an action (rule) one can ignore the ordering constraints between mutually exclusive sub-actions

Some examples to intra-rule and inter-rule analysis
CM for One-Element FIFO

module mkPipelineFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
  Reg#(t) d <- mkRegU;
  Reg#(Bool) v <- mkReg(False);
method Action enq(t x) if (!v);
  d <= x;
  v <= True;
endmethod
method Action deq if (v);
  v <= False;
endmethod
method t first if (v);
  return d;
endmethod
endmodule

Notice enq and deq are mutually exclusive

mcalls(enq) = {v.r, d.w, v.w}
mcalls(deq) = {v.r, v.w}
mcalls(first) = {v.r, d.r}

Intra-rule CM

\[
\begin{array}{ccc}
\text{enq} & \text{deq} & \text{first} \\
C & ME & ME \\
ME & C & CF \\
ME & CF & CF \\
\end{array}
\]

Inter-rule CM

\[
\begin{array}{ccc}
\text{enq} & \text{deq} & \text{first} \\
C & ME & ME \\
ME & C & > \\
ME & < & CF \\
\end{array}
\]
**CM for One-Element Pipeline FIFO**

```verilog
module mkPipelineFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
    Reg#(t) d <- mkRegU;
    Ehr#(2, Bool) v <- mkEhr(False);
    method Action enq(t x) if (!v[1]);
        d <= x;
        v[1] <= True;
    endmethod
    method Action deq if (v[0]);
        v[0] <= False;
    endmethod
    method t first if (v[0]);
        return d;
    endmethod
endmodule
```

mcalls(enq) = \{v.r1, d.w, v.w1\}
mcalls(deq) = \{v.r0, v.w0\}
mcalls(first) = \{v.r0, d.r\}

CM\[enq,deq\] = \conflict[v.r1,v.r0] \land \conflict[v.r1,v.w0] \land \conflict[d.w,v.r0] \land \conflict[d.w,v.w0] \land \conflict[v.w1,v.r0] \land \conflict[v.w1,v.w0]

**Intra-rule derivation**
```verilog
cm for One-Element Bypass FIFO

module mkBypassFifo(Fifo#(1, t)) provisos(Bits#(t, tSz));
  Ehr#(2, t) d <- mkEhr(?);
  Ehr#(2, Bool) v <- mkEhr(False);
  method Action enq(t x)
    if !v[0];
      d[0] <= x;
      v[0] <= True;
  endmethod
  method Action deq
    if v[1];
      v[1] <= False;
  endmethod
  method t first
    if v[1];
      return d[1];
  endmethod
endmodule

mcalls(enq) = {d.w0, v.w0, v.r0}
mcalls(deq) = {v.r1, v.w1}
mcalls(first) = {v.r1, d.r1}

CM[enq,deq] =

Intra-rule derivation
```

**CM for One-Element Bypass FIFO**

mcalls(enq) = \{d.w0, v.w0, v.r0\}
mcalls(deq) = \{v.r1, v.w1\}
mcalls(first) = \{v.r1, d.r1\}

CM[enq,deq] =

Intra-rule derivation
module mkCFFifo(Fifo#(2, t)) provisos(Bits#(t, tSz));
  Ehr#(2, t) da <- mkEhr(?);
  Ehr#(2, Bool) va <- mkEhr(False);
  Ehr#(2, t) db <- mkEhr(?);
  Ehr#(2, Bool) vb <- mkEhr(False);

  rule canonicalize(vb[1] && !va[1]);
    da[1] <= db[1];
    va[1] <= True; vb[1] <= False; endrule

  method Action enq(t x) if va[0];
    db[0] <= x; vb[0] <= True; endmethod
  method Action deq if !vb[0];
    va[0] <= False endmethod
  method t first if !vb[0];
    return da[0]; endmethod
endmodule

CM for Two-Element Conflict-free FIFO

延展CM到规则
使用inter-rule CMs

- CM between two rules is computed exactly the same way as CM for the methods of a module
- Given rule $r_1$ a1 and rule $r_2$ a2 such that
  \[
  \text{mcalls(a1)} = \{g_{11}, g_{12}, \ldots, g_{1n}\}
  \text{mcalls(a2)} = \{g_{21}, g_{22}, \ldots, g_{2m}\}
  \]
- Compute
  - Conflict\(x, y\) = if \(x\) and \(y\) are methods of the same module then CM[\(x, y\)] else CF
  - CM[r1, r2] = conflict(g11, g21) \&\& conflict(g12, g21) \&\& conflict(g11, g22) \&\& conflict(g12, g22) \&\& ... \&\& conflict(g1n, g21) \&\& conflict(g12, g22) \&\& ... \&\& conflict(g1n, g2m)
- Conflict relation is not transitive
  - $r_1 < r_2, r_2 < r_3$ does not imply $r_1 < r_3$
Using CMs for concurrent scheduling of rules

Two rules that are conflict free can be scheduled together without violating the one-rule-at-a-time semantics. In general, use the following theorem:

Theorem: Given a set of rules \{rule \( r_i \), \( a_i \)\}, if there exists a permutation \{\( p_1, p_2 \ldots p_n \)\} of \{1..n\} such that

\[
\forall \ i < j . \ CM(a_{p_i}, a_{p_j}) \text{ is CF or } < \text{ or ME}
\]

then the rules \( r_1, r_2 \ldots r_n \) can be scheduled concurrently with the effect \( \forall \ i, j . \ r_{p_i} < r_{p_j} \)

A compiler can perform the analysis needed for concurrent scheduling of rules. James Hoe, 2000

Scheduling constraints due to multiple modules

Can ra and rb be scheduled concurrently?

<table>
<thead>
<tr>
<th></th>
<th>aTob</th>
<th>bToa</th>
<th>Concurrent scheduling?</th>
</tr>
</thead>
<tbody>
<tr>
<td>fifo</td>
<td>CF</td>
<td>CF</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>pipeline</td>
<td>pipeline</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>bypass</td>
<td>bypass</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>CF</td>
<td>CF</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>pipeline</td>
<td>pipeline</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>bypass</td>
<td>bypass</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>CF</td>
<td>CF</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>pipeline</td>
<td>pipeline</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>bypass</td>
<td>bypass</td>
<td></td>
</tr>
</tbody>
</table>

// assume both fifos are not empty

February 22, 2016
http://csg.csail.mit.edu/6.375
L07-27

February 17, 2016
http://csg.csail.mit.edu/6.375
L06-28