Behavioral description using message-passing

Rajit Manohar

Asynchronous VLSI and Architecture (AVLSI) Group
Computer Systems Lab, Yale University

https://avlsi.csl.yale.edu/
https://csl.yale.edu/~rajit/
https://avlsi.csl.yale.edu/act
Communicating Hardware Processes

• Behavioral language
  ❖ CHP = Communicating hardware processes
    ‣ Based on Tony Hoare’s CSP (Communicating Sequential Processes) language
  ❖ Simplified programming language
• Assignment-based language, but...
  ❖ No memory allocation
  ❖ No memory references
• Basic data types: Booleans and unsigned integers

```c
bool x;
int y;
int<8> z;
```
Basic language constructs

• Simple statements
  
  ❖ **skip**  
  *statement that does nothing*

  ❖ **x := E**  
  *assignment*

  ‣ “Evaluate expression on the right-hand side, assign it to the variable on the left-hand side” : just like a standard software programming language

  
  \[
  \begin{align*}
  y & := y + x*2 \\
  b & := b \mid \neg z \\
  \text{skip} & \\
  \text{b-} & \\
  \text{b+} & 
  \end{align*}
  \]

• **S₁;S₂**  
  *Execute statement 1 until it is finished. Then execute statement 2*

  
  \[
  \begin{align*}
  y & := y + x*2; \\
  w & := y - 3; \\
  y & := 0; \text{skip}
  \end{align*}
  \]
Arrays

• In hardware, an array results in an address-calculation mechanism

\[ x[i] := x[i] + 1 \]

• Array access is of two kinds
  ❖ Standard array, where array index requires run-time information, or
  ❖ Array index is a constant

\[ x[0] := x[0] + 1 \]

• Only use standard arrays when absolutely necessary!
Selection statements

- **Deterministic selection** statement: generalized if-statement

  ```
  [ x > 10  ->  y := 3
  [ ] x < 10  ->  y := 4
  ]
  ```

  - If some condition (guard) is true, execute corresponding statement
  - If all guards are false, then **wait**
  - If multiple guards are true, error!

- **Non-deterministic selection**

  ```
  [ | x > 10  ->  y := 3
  [ ] x < 10  ->  y := 4
  [ ] x > 8  ->  y := 7
  [ ]
  ```

  Note: in CHP, semi-colon is used as a separator (no trailing semi-colon)
Loops

- Deterministic loop statement

  ```
  *[ x > 10 -> y := 3; x:= x - 1
   [ ] x < 10 -> y := 4; x := x + 1
  ]
  ```

- If some condition (guard) is true, execute corresponding statement and go back to the loop start
- If all guards are false, **then exit**
- If multiple guards are true, error!

```
  i := 0; j := 0;
  *[ i < 10 ->
     j := j + i;
     i := i + 1
  ]
```
More constructs

- $S_1, S_2$  
  *Execute non-interfering statements 1 and 2 in parallel*

  ```
  *[ x > 10 -> y := 3, x := x - 1 ]
  [ ] x < 10 -> y := 4, x := x + 1
  ```

- Common short-hand
  - Infinite loop
    ```
    *[ true -> STMTS ]
    *[ STMTS ]
    ```
  
  - Wait for some condition
    ```
    [ COND -> skip ]
    [ COND ]
    ```
Communication

• Hardware modules exchange information via **communication channels**

• **Channel**
  
  ❖ single-sender, single-receiver
  ❖ a matching send and receive behaves as a *distributed assignment*

  ![X!e](image1)
  ![Y?x](image2)

  Evaluate “e” and send it on output port X
  Receive value from input port Y and assign it to variable “x”

  ❖ If these two ports are connected, then this has the net effect of

  ![x := e](image3)

  ❖ Channels are **blocking**: a send waits for matching receive, and a receive waits for a matching send.
Overall hardware description

• A parallel collection of communicating hardware processes
  ❖ By default, no shared state

• Connections between processes via channels to exchange information
  ❖ (General shared variables possible; ignoring for this summer school!)

• For this summer school, syntax for connections, type declarations, etc. in the ACT language
  ❖ There are other examples of CSP-like languages (e.g. Occam)
Example: buffer

• One-place buffer, initially empty

  ❖ Empty state
    ‣ Only operation that is valid: read next input
      \[ \text{L?}x \]
    ‣ Final state: full
  ❖ Full state
    ‣ Only operation that is valid: send value on output
      \[ \text{R!}x \]
    ‣ Final state: empty
  ❖ Empty state to empty state:
    \[ \text{L?}x; \text{R!}x \]

• Buffer repeats this forever:
  \[ *\{ \text{L?}x; \text{R!}x \} \]
Example: adder

- Read two operands
- Add them together
- Send the result on the output

Parallel read

\[
*\left[ \begin{array}{c}
A?x; B?y; \\
O!(x+y)
\end{array} \right]
\]

Local state in variables \(x, y\)

Input port A

Input port B

Output port O
Synchronization

- Communication actions synchronize different parallel processes
  - Knowing where one process is in its local program can give you information about what other processes in the system are doing.

\[
\begin{align*}
&*[ \text{A}?x;\text{B}?y; \text{O}!(x+y) ] \\
&*[ \text{A}p!1 ] \\
&*[ \text{B}p!2 ] \\
&\text{Input port A} \quad \text{Local state in variables } x, y \quad \text{Output port O} \\
&\text{Input port B} \\
&*[ \text{O}p?z ]
\end{align*}
\]
Non-deterministic constructs

- Problem: two input ports A and B and one output Z
  - Receive the “next input” from either A or B
  - Send this value on the output Z

- We need some new syntax!
  - Probe: “is there a communication pending on this port?”

```plaintext
*[ [ | #A -> A?x ] [ | #B -> B?x ] ]; Z!x ]
```

- Use with care, and only when absolutely necessary
Example: add/subtract/multiply

Local state in variables x, y, c

Input port A

Input port B

Input port Op

Output port O

*[ A?x, B?y, Op?c;
   [ c=0  ->  O!(x+y)
   [ ]c=1  ->  O!(x-y)
   [ ]c=2  ->  O!(x*y)
   ]
]*