Lecture 13: Introduction to CSP (Communicating Sequential Processes)

Dr. Richard S. Hall
rickhall@inf.fu-berlin.de

This lecture is based on the book "The Theory and Practice of Concurrency" by A.W. Roscoe (Prentice-Hall 1998)

Communicating Sequential Processes (CSP)

- A language for describing processes that interact
- Invented by Tony Hoare
  - First version in late 1970's and the second version in the early to mid 1980's
  - Our discussion will focus on a CSP dialect of the second version presented in Roscoe's book
- We will see many similarities to FSP
Fundamental CSP Concepts

- A CSP process is completely described by the ways in which it can communicate with its external environment.
- The most important first step in a CSP process is choosing its \textit{alphabet} of event communication:
  - An appropriate set of \textit{atomic} interactions for the world we are modeling.
  - The alphabet of all events is written $\Sigma$.
  - In CSP, events are assumed to be instantaneous, i.e., the instant when an interaction is agreed.

Example Alphabets

- $\{ \text{up, down, iszero} \}$ for a simple counter.
- $\{ \text{in}.x, \text{out}.x \mid x \in T \}$ for a unit that inputs and outputs values (of type $T$) on one channel each.
- $\{ \text{pay}.x, \text{change}.x \mid x \in M \} \cup \{ \text{cheddar}.w, \text{gouda}.w, \text{parmesan}.w, \ldots \mid w \in W \}$ where $M$ is the set of money amounts, $W$ is the set of weights for a cheese shop.
STOP Process

- The simplest process is **STOP**
  - Just like in FSP
  - **STOP** performs no actions at all
  - Can be convenient in specifications and also provides a simple model of a deadlocked system

Prefixing

- If \( P \) is a process and \( a \in \Sigma \) is any communication, then \( a \rightarrow P \) says that
  - \( a \) is offered until the environment accepts it and then behaves like \( P \)
- So just like in FSP, we can build processes like
  - \( \text{up} \rightarrow \text{down} \rightarrow \text{STOP} \)
  - \( \text{pay.$5} \rightarrow \text{gouda.500g} \rightarrow \text{cheddar.1kg} \rightarrow \text{change.$1.23} \rightarrow \text{STOP} \)
**Recursion**

- We can create processes that communicate forever by having them return to previous states
  - $P_1 = \text{up} \rightarrow \text{down} \rightarrow P_1$
  - $P_2 = \text{up} \rightarrow \text{down} \rightarrow \text{up} \rightarrow \text{down} \rightarrow P_2$
  - $P_u = \text{up} \rightarrow \text{down}$
  - $P_d = \text{down} \rightarrow \text{up}$
  - This last one obviously creates two processes

**Choice**

- If $A \subseteq \Sigma$ is any set of events and $P(a)$ is a process for each $a \in A$, then $?x : A \rightarrow P(x)$ says that
  - The environment is offered the choice of $A$ and then behaves like the appropriate $P(a)$

- Examples,
  - $\text{RUN}_A = ?x : A \rightarrow \text{RUN}_A$
  - $\text{REPEAT} = ?x : \Sigma \rightarrow x \rightarrow \text{REPEAT}$
Guarded Alternative

- Using the guarded alternative construct, just like in FSP, we can write
  \((a \rightarrow P(a) \mid b \rightarrow P(b) \mid ... \mid z \rightarrow P(z))\)

- Example
  - \(\text{COUNT}_0 = \text{up} \rightarrow \text{COUNT}_1\)
  - \(\text{COUNT}_{n+1} = (\text{up} \rightarrow \text{COUNT}_{n+2} \mid \text{down} \rightarrow \text{COUNT}_n)\)

Channels and Input/Output

- An event in \(\Sigma\) consists [conceptually] of a channel name plus zero or more data components
  - e.g., up, cheddar.1kg, send.a.b.m
  - The data components are sometimes used to transmit data between processes and sometimes to create arrays of channels
- Often a process will want to allow all or some communication on one of its channels
  - \(c?x \rightarrow P(x)\) where \(x \in \Sigma\)
  - \(c?x : A \rightarrow P(x)\) where \(x \in A\)
- When output happens on a channel, it is written \(c!x \rightarrow P\) rather than \(c.x \rightarrow P\), these are almost synonymous (but they are different when inputs and outputs are mixed in same communication)
Channels and Guarded Alternative

- Provided they are one distinct channels, inputs and outputs are allowed in the guarded alternative construct

\[
\begin{align*}
CS(0) &= \text{pay}\!?:x \rightarrow CS(x) \\
CS(x) &= (\text{cheddar}\!?:w : \{z \in W | z \times V_c \leq x\} \rightarrow CS(x - w \times V_c) \\
&\quad | \text{gouda}\!?:w : \{z \in W | z \times V_g \leq x\} \rightarrow CS(x - w \times V_g) \\
&\quad | \text{parmesan}\!?:w : \{z \in W | z \times V_p \leq x\} \rightarrow CS(x - w \times V_p) \\
&\quad | \text{pay}\!?:y \rightarrow CS(x + y) \\
&\quad | \text{change}\!?!x \rightarrow CS(0))
\end{align*}
\]

External Choice Operator

- The external choice operator generalizes the guarded alternative construct

- \( P \square Q \) offers the environment the choice between the initial actions of \( P \) and \( Q \) and then behaves like the one whose action is picked

- Every guarded alternative can be replaced by \( \square \)
**External vs. Guarded Choice**

- Consider guarded alternative as a "stepping-stone" to understanding $\Box$, rather than actually having a proper place in CSP.

- It is obvious that if $A \cap B = \{\}$ then

\[
(?x : A \rightarrow P(x)) \Box (?x : B \rightarrow Q(x)) = ?x : A \cup B \rightarrow R(x)
\]

where $R(x)$ is $P(x)$ or $Q(x)$ depending on whether $x$ is in $A$ or $B$.

- What happens when $A \cap B \neq \{\}$?
  - If the environment selects an initial event that is common to $P$ or $Q$ in $P \Box Q$ then it is *non-deterministic*.

**Non-deterministic Choice**

- Since non-determinism can occur naturally, CSP models it with the non-deterministic operation $\Box$.
  - $P \Box Q$ can behave like $P$ or like $Q$.

- Examples
  - $(a \rightarrow b \rightarrow \text{STOP}) \Box (a \rightarrow c \rightarrow \text{STOP})$ or
  - $a \rightarrow (b \rightarrow \text{STOP} \Box c \rightarrow \text{STOP})$.
Conditional Choice

- Neither $\square$ nor $\parallel$ are found in "ordinary" programming languages
- Another more conventional "choice" construct is
  - if $b$ then $P$ else $Q$ also written as $P \leftarrow b \rightarrow Q$

Parallel Processes

- Parallel process interact by handshake communication (in which both parties have to agree)
- The simplest CSP parallel operator $P \parallel Q$ makes two processes agree on everything
  - This is different from what we see in FSP
  - $?x : A \rightarrow P \mid ?x : B \rightarrow Q = ?x : A \cap B \rightarrow (P \parallel Q)$
  - Example
    - $P = (a \rightarrow a \rightarrow \text{STOP}) \square (b \rightarrow \text{STOP})$
    - $Q = (a \rightarrow \text{STOP}) \square (c \rightarrow a \rightarrow \text{STOP})$
    - $P \parallel Q = a \rightarrow \text{STOP}$
    - because the processes only agree on $a$
Alphabetized Parallel Operator

- Parallel process will not generally agree on all communications
  - Some communications will be shared and some will not be shared
- If $X$ and $Y$ are subsets of $\Sigma$, $P_{X\|Y}Q$ is the combination where $P$ and $Q$ are assigned the alphabets $X$ and $Y$, respectively
  - $P$ must perform every communication in $X$ and
  - $Q$ must perform every communication in $Y$
- $X \cap Y$ are communications between $P$ and $Q$, with $X\setminus Y$ and $Y\setminus X$ their independent actions

Example

$$\langle a \rightarrow b \rightarrow b \rightarrow \text{STOP} \rangle_{\{a,b\}\|\{b,c\}} (b \rightarrow c \rightarrow b \rightarrow \text{STOP})$$

has the behavior

$$a \rightarrow b \rightarrow c \rightarrow b \rightarrow \text{STOP}$$

because initially the only possible event is $a$ (since the left hand side blocks $b$); then both sides agree on $b$ and so no.
Alphabets

- Since the alphabet of a process is simply the set of actions it can perform, why do we need them?
  - Because processes sometimes cannot perform all of the actions we think they can, therefore it is vital that we know clearly whether processes must agree on some action
  - Because sometimes it is useful to give a process a bigger alphabet so it can stop another one from performing some actions
  - We have seen this in FSP, right?

Pantomime Horse Example

- In a pantomime horse an actor plays the front half of the horse and an actor plays the back half
- Suppose we have
  - Front $\parallel_B$ Back
    - $F = \{\text{forward, backward, nod, neigh}\}$
    - $B = \{\text{forward, backward, wag, kick}\}$
    - Front = forward → Front' □ nod → Front
    - Back = backward → Back' □ wag → Back
  - The horse will never perform Front' and Back', it will simply wag and nod forever
  - This is summarized by the Step Law of $\chi\parallel_Y$
Step Law of $x\parallel Y$

- Suppose
  - $P = ?x : A \to P'$
  - $Q = ?x : B \to Q'$
  - $C = (A \cap (X \setminus Y)) \cup (B \cap (Y \setminus X)) \cup (A \cap B \cap X \cap Y)$

then
  - $P \parallel Y Q = ?x : C \to (P' \ll x \in X \gg P \parallel_Y Q' \ll x \in Y \gg Q)$

Dining Philosophers

- We know the example from a previous lecture...
- The fork process
  - $\text{FORK}_i = (\text{picksup}.i \cdot i \to \text{putdown}.i \cdot i \to \text{FORK}_i)$
  - $\Box (\text{picksup}.i \oplus 1 \cdot i \to \text{putdown}.i \oplus 1 \cdot i \to \text{FORK}_i)$
- The philosopher process
  - $\text{PHIL}_i = \text{thinks}.i \to \text{sits}.i$
    - $\to \text{picksup}.i \cdot i \to \text{picksup}.i \cdot i \oplus 1$
    - $\to \text{eats}.i \to \text{putdown}.i \cdot i \oplus 1$
    - $\to \text{putdown}.i \cdot i \to \text{getsup}.i \to \text{PHIL}_i$
- Alphabets are $\text{AF}_i$ and $\text{AP}_i$, respectively
Dining Philosophers

- The completed dining philosophers system is formed by composing these ten pairs
  \[ \{(\text{FORK}_i, \text{AF}_i), (\text{PHIL}_i, \text{AP}_i) \mid i \in \{0,1,2,3,4\}\} \]
  in parallel

Interleaving Operator

- \(\|\) and \(\|_x\) make all partners allowed to communicate a given event, synchronize on it, the opposite is true of parallel composition by interleaving, \(P \| Q\)
  - \(P\) and \(Q\) run independently of each other and any event of \(P \| Q\) occurs in exactly one of \(P\) and \(Q\)
  - If both perform event \(a\), then we get non-determinism
  - If \(P = ?x : A \rightarrow P'\) and \(Q = ?x : B \rightarrow Q'\) then
    \[ P \| Q = ?x : A \cup B \rightarrow \]
    \[ (P' \| Q) \cap (P \| Q') \]
    \[ \leftarrow x \in A \cap B \rightarrow \]
    \[ (P' \| Q) \leftarrow x \in A \rightarrow (P \| Q') \]
**Interleaving Examples**

- An array of printers
  
  \[
  \text{Printer}(n) = \text{input}\? x \rightarrow \text{print}\.n\!x \rightarrow \text{Printer}(n)
  \]

  \[
  \text{Printroom} = \|\|_{n=1}^{i} \text{Printer}(n)
  \]

  - This is non-deterministic because the user has no control over which printer prints his file

- Behavior of \text{COUNT\_0} with single recursion
  
  \[
  \text{Ctr} = \text{up} \rightarrow (\text{Ctr} \|\| \text{down} \rightarrow \text{Ctr})
  \]

  - This effectively "spawns" off capabilities that remain active while further calls are made
  - This is very subtle

**Using Interleaving**

- The previous examples of interleaving were pretty sophisticated and require that you really understand the behavior you want

- The most common use of \|\| is as a substitute for \$_{X}\|_{Y}$ in cases where \( X \) and \( Y \) are disjoint

  - This saves the effort of having to define alphabets, for example
    
    \[
    \text{FORKS} = \text{FORK}_0 \|\| \text{FORK}_1 \|\| \ldots \|\| \text{FORK}_4
    \]
    
    \[
    \text{PHILS} = \text{PHIL}_0 \|\| \text{PHIL}_1 \|\| \ldots \|\| \text{PHIL}_4
    \]
    
    \[
    \text{AFS} = \{ | \text{picksup, putsdown } | \}
    \]
    
    \[
    \text{SYSTEM} = \text{FORKS}_{\text{AFS}} \|_{\Sigma} \text{PHILS}
    \]
Generalized Parallel

- $\parallel_x \parallel_Y$, and $\parallel \parallel$ are all special cases of a single operator, $P \parallel_X Q$, called interface parallel

- This operator runs $P$ and $Q$, making them synchronize on events in $X$ and independently on others

- $P \parallel_X Y Q = P \parallel Q_{X \cap Y}$

- $P \parallel Q = P \parallel Q_{\Sigma}$

- $P \parallel Q = P \parallel Q_{\{}$

- $P \parallel_X Q = P \parallel_Y Y Z Q$ where $X = Y \cap Z$

Parallel Composition as Conjunction

- Can be used to build trace specifications

- Consider

- $\text{ROBOT}_{n,m} = \text{position.}(n,m) - \text{ROBOT}_{n,m}$
  - north $\rightarrow \text{ROBOT}_{n-1,m}$
  - south $\rightarrow \text{ROBOT}_{n+1,m}$
  - east $\rightarrow \text{ROBOT}_{n,m+1}$
  - west $\rightarrow \text{ROBOT}_{n,m-1}$

- If the "world" for the robot is a rectangle with the corners $\{ (0,0), (n,0), (n,m), (0,m) \}$ then we can constrict its movement with parallel composition

  - See next slide...
Parallel Composition as Conjunction

- Compose ROBOT with
  - $\text{CT}(\text{east, west})_0$ alphabet \{ east, west \}
  - $\text{CT}(\text{west, east})_m$ alphabet \{ east, west \}
  - $\text{CT}(\text{north, south})_n$ alphabet \{ north, south \}
  - $\text{CT}(\text{south, north})_0$ alphabet \{north, south \}

where

\[
\text{CT}(a, b)_r = a \rightarrow \text{CT}(a, b)_{r+1}
\]

\[\square \; b \rightarrow \text{CT}(a, b)_{r-1} \text{ if } r > 0\]