Concurrent Programming
19530-V (WS01)

Lecture 13: Introduction to CSP (Communicating Sequential Processes)

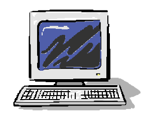
Dr. Richard S. Hall
rickhall@inf.fu-berlin.de

Communicating Sequential Processes

- 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 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 ***alphabet*** of event communication
  - An appropriate set of ***atomic*** interactions for the world we are modeling
  - The alphabet of all events is written Σ
  - In CSP, events are assumed to be instantaneous, i.e., the instant when an interaction is agreed.



Example Alphabets

- { up, down, iszero } for a simple counter
- $\{ in.x, out.x \mid x \in T \}$ for a unit that inputs and outputs values (of type T) on one channel each
- $\{ pay.x, change.x \mid x \in M\} \cup$
  $\{ cheddar.w, gouda.w, parmesan.w, ... \mid w \in W\}$
  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
  - up → down → STOP
  - pay.$5 → gouda.500g → cheddar.1kg → change.$1.23 → STOP

Recursion

- We can create processes that communicate forever by having them return to previous states
  - $P_1 = up \rightarrow down \rightarrow P_1$
  - $P_2 = up \rightarrow down \rightarrow up \rightarrow down \;\; P_2$
  - $P_u = up \rightarrow P_d$
    $P_d = down \rightarrow P_u$
    - 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,
  - $RUN_A = ?x : A \rightarrow RUN_A$
  - $REPEAT = ?x : \Sigma \rightarrow x \rightarrow REPEAT$

Guarded Alternative

- Using the *guarded alternative* construct, just like in FSP, we can write
  $(a \rightarrow P(a) \mid b \rightarrow P(b) \mid \ldots \mid z \rightarrow P(z))$
- Example
  - $COUNT_0 = up \rightarrow COUNT_1$
    $COUNT_{n+1} = (up \rightarrow COUNT_{n+2} \mid down \rightarrow COUNT_n)$

Channels and Input/Output

- An event in Σ 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

$$
\begin{aligned}
CS(0) &= pay?x \rightarrow CS(x) \\
CS(x) &= (cheddar?w : \{z \in W \mid z \times V_C \leq x\} \rightarrow CS(x - w \times V_C) \\
&\qquad \mid gouda?w : \{z \in W \mid z \times V_G \leq x\} \rightarrow CS(x - w \times V_G) \\
&\qquad \mid parmesan?w : \{z \in W \mid z \times V_P \leq x\} \rightarrow CS(x - w \times V_P) \\
&\qquad \mid pay?y \rightarrow CS(x + y) \\
&\qquad \mid change!x \rightarrow CS(0))
\end{aligned}
$$

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 $\square$, rather than actually having a proper place in CSP
- It is obvious that if $A \cap B = \{\ \}$ then $(?x : A \rightarrow P(x)) \ \square\ (?x : B \rightarrow Q(x)) = ?x : A \cup B \rightarrow R(x)$ 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 $P \ \square\ Q$ then it is *non-deterministic*

Non-deterministic Choice

- Since non-determinism can occur naturally, CSP models it with the non-deterministic operation $\sqcap$
  - $P \sqcap Q$ can behave like P or like Q
- Examples
  - $(a \rightarrow b \rightarrow STOP) \ \square\ (a \rightarrow c \rightarrow STOP)$ or $a \rightarrow (b \rightarrow STOP \sqcap c \rightarrow STOP)$

Conditional Choice

- Neither $\Box$ nor $\sqcap$ are found in "ordinary" programming languages
- Another more convential "choice" construct is
  - if b then P else Q also written as
    $P \nprec b \nsucc Q$

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
  - $?x : A \rightarrow P \parallel ?x : B \rightarrow Q = ?x : A \cap B \rightarrow (P \parallel Q)$
  - Example
    - $P = (a \rightarrow a \rightarrow STOP) \Box (b \rightarrow STOP)$
      $Q = (a \rightarrow STOP) \Box (c \rightarrow a \rightarrow STOP)$
      $P \parallel Q = a \rightarrow STOP$
      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 Σ, $P \;{}_{X}\|_{Y}\; Q$ 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

Alphabetized Parallel Operator

- Example
  $(a \rightarrow b \rightarrow b \rightarrow STOP) \;{}_{\{a,b\}}\|_{\{b,c\}}\; (b \rightarrow c \rightarrow b \rightarrow STOP)$
  has the behavior
  $a \rightarrow b \rightarrow c \rightarrow b \rightarrow STOP$
  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 $_F||_B$ Back

    F = {forward,backward,nod,neigh}

    B = {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* $_X||_Y$

Step Law of $_X\|_Y$

- Suppose
  $P = \; ?x : A \rightarrow P'$
  $Q = ?x : B \rightarrow Q'$
  $C = (A \cap (X \setminus Y)$ — P by itself
  $\quad \cup (B \cap (Y \setminus X))$ — Q by itself
  $\quad \cup (A \cap B \cap X \cap Y)$ — interactions
  then
  $P \; _X\|_Y \; Q = ?x : C \rightarrow (\; P' \not< x \in X \not> P$
  $\qquad _X\|_Y$
  $\qquad Q' \not< x \in Y \not> Q \;)$

Dining Philosophers

- We know the example from a previous lecture...
- The fork process
  $FORK_i = (picksup.i.i \rightarrow putsdown.i.i \rightarrow FORK_i)$
  $\Box \; (picksup.i \ominus 1.i \rightarrow putsdown.i \ominus 1.i \rightarrow FORK_i)$
- The philosopher process
  $PHIL_i = thinks.i \rightarrow sits.i$
  $\quad \rightarrow picksup.i.i \rightarrow picksup.i.i \oplus 1$
  $\quad \rightarrow eats.i \rightarrow putsdown.i.i \oplus 1$
  $\quad \rightarrow putsdowni.i \rightarrow getsup.i \rightarrow PHIL_i$
- Alphabets are AF_i and AP_i, respectively

Dining Philosophers

- The completed dining philosophers system is formed by composing these ten pairs $\{(FORK_i, AF_i), (PHIL_i, AP_i) \mid i \in \{0,1,2,3,4\}\}$ in parallel

Interleaving Operator

- $||$ and ${}_X||_Y$ make all partners allowed to communicate a given event, synchronize on it, the opposite is true of parallel composition by interleaving, $P \mathbin{|||} Q$
  - P and Q run independently of each other and any event of $P \mathbin{|||} Q$ 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

    $$P \mathbin{|||} Q = ?x : A \cup B \rightarrow$$
    $$(P' \mathbin{|||} Q) \sqcap (P \mathbin{|||} Q')$$
    $$\not< x \in A \cap B \not>$$
    $$(P' \mathbin{|||} Q) \not< x \in A \not> (P \mathbin{|||} Q')$$

Interleaving Examples

- An array of printers
  $Printer(n) = input?x \rightarrow print.n!x \rightarrow Printer(n)$
  $Printroom = |||^{4}_{n=1}\ Printer(n)$
  - This is non-deterministic because the user has no control over which printer prints his file
- Behavior of $COUNT_0$ with single recursion
  $Ctr = up \rightarrow (Ctr\ |||\ down \rightarrow Ctr)$
  - This effectively "spawns" off capabilities that remain active while further calls are made
  - This is very subtle

Using Interleaving

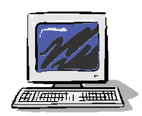
- The previous examples of interleaving were pretty sophisticated and require that you really understand the behavior you want
- The most common use of $|||$ is as a substitute for ${}_X||_Y$ in cases where X and Y are disjoint
  - This saves the effort of having to define alphabets, for example
    $FORKS = FORK_0\ |||\ FORK_1\ |||\ \ldots\ |||\ FORK_4$
    $PHILS = PHIL_0\ |||\ PHIL_1\ |||\ \ldots\ |||\ PHIL_4$
    $AFS = \{|\ picksup, putsdown\ |\}$
    $SYSTEM = FORKS\ {}_{AFS}||_{\Sigma}\ PHILS$

Generalized Parallel

- $\|$, ${}_X\|_Y$, and $|||$ are all special cases of a single operator, $P \underset{X}{\|} Q$, called *interface* parallel
  - This operator runs P and Q, making them synchronize on events in X and independently on others
  - $P \;{}_X\|_Y\; Q = P \underset{X \cap Y}{\|} Q$

    $P \| Q = P \underset{\Sigma}{\|} Q$

    $P \;|||\; Q = P \underset{\{\}}{\|} Q$
- $P \underset{X}{\|} Q = P \;{}_Y\|_Z\; Q$ where $X = Y \cap Z$



Parallel Composition as Conjunction

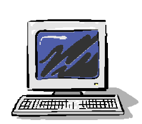
- Can be used to build trace specifications
- Consider
  - $ROBOT_{n,m} = position.(n,m) - ROBOT_{n,m}$

    $\square$ north $\rightarrow ROBOT_{n-1,m}$

    $\square$ south $\rightarrow ROBOT_{n+1,m}$

    $\square$ east $\rightarrow ROBOT_{n,m+1}$

    $\square$ west $\rightarrow ROBOT_{n,m-1}$
  - 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
  - $CT(east, west)_0$ alphabet { east, west }
    $CT(west, east)_m$ alphabet { east, west }
    $CT(north, south)_n$ alphabet { north, south }
    $CT(south, north)_0$ alphabet {north, south }
    where
    $CT(a, b)_0 = a \rightarrow CT(a, b)_1$
    $CT(a, b)_r = a \rightarrow CT(a, b)_{r+1}$
    $\Box\ b \rightarrow CT(a, b)_{r-1}$ if $r > 0$