CSP Chapter 4: Communication

Andrzej Filinski
DIKU

Extreme Multiprogramming 8
December 13, 2013

Overview

- Today: channel-based communication.
  - No new concepts; just using existing primitives in a particularly structured way.
  - Multi-party participatory *events* $\rightsquigarrow$ two-party *communications*
  - In many CSP applications, *all* events are communications.
- Still useful to understand communication a special case of event-based synchronization.
  - All previous considerations about concurrency, deadlock, nondeterminism, divergence, etc. still apply.
  - Can understand fundamental issues in isolation, without getting tangled up in specifics of channel I/O.
- Will also introduce a bit of notation from CSP 5.1–5.2 (sequential composition).

Channels

- Channels impose *structure* on event sets.
  - So far, event names have generally been *atomic* (*coin*, *coke*, *sprite*, ...).
  - Though we have hinted that they *might* have internal structure (e.g, *coin*.10*kr*): "." suggests qualitative difference from "_".
  - Communication operators introduce specialized terminology for working with such events: *coin* is *channel*, 10*kr* is *message*.
  - **Def.** auxiliary functions: $chan(c.m) = c$, $msg(c.m) = m$.
  - Shorthand: writing C for $\{c.m \mid c \in C\}$ in $P \setminus C$, $STOP_C$, etc.
- Channels model *directed* events.
  - So far, no separation of *roles* played by participants in an event
  - *coin* means "accept a coin" to vending machine, but "insert a coin" to customer. No formal distinction!
  - Channels make a clear separation between roles of *sender* (initiator) and *receiver* (acceptor) of an event.
  - CSP channel I/O provides specialized syntax for working with communication events.

Communication topology and alphabets

- Most (frequently: all) of a process's event alphabet can be structured as:
  1. Two finite, disjoint sets of *incoming* and *outgoing* channel names: $\alpha_{\mathrm{i}} P = \{c_1, \ldots, c_n\}$, $\alpha_{\mathrm{o}} P = \{c'_1, \ldots, c'_m\}$, where $\alpha_{\mathrm{i}} P \cap \alpha_{\mathrm{o}} P = \{\}$.
  2. For each channel, a (generally infinite) *channel alphabet*: for $c \in \alpha_{\mathrm{i}} P \cup \alpha_{\mathrm{o}} P$, $\alpha c(P) = \{m \mid c.m \in \alpha P\}$.
- By convention, channels are directed, *one-to-one* connections between concurrent processes P and Q:
  - *Matching* channels (same name, but incoming in P and outgoing in Q or vice versa) are implicitly connected, and normally concealed from environment:

$$R = (P \parallel Q) \setminus C, \text{where } C = (\alpha_{\mathrm{o}} P \cap \alpha_{\mathrm{i}} Q) \cup (\alpha_{\mathrm{i}} P \cap \alpha_{\mathrm{o}} Q)$$

  - *Non-matching* channels are left unconnected and visible to environment:

$$\alpha_{\mathrm{i}} R = (\alpha_{\mathrm{i}} P \cup \alpha_{\mathrm{i}} Q) - C \qquad \alpha_{\mathrm{o}} R = (\alpha_{\mathrm{o}} P \cup \alpha_{\mathrm{o}} Q) - C.$$

Ill-formed compositions

- $R = (P \parallel Q) \setminus C$, where $C = (\alpha_{\mathsf{o}} P \cap \alpha_{\mathsf{i}} Q) \cup (\alpha_{\mathsf{i}} P \cap \alpha_{\mathsf{o}} Q)$.
- Two possibilities for *clashes*:
  - Different channel alphabets on matching channels: $\alpha c(P) \neq \alpha c(Q)$, for some c.
  - Multiple senders/receivers on a channel: $\alpha_{\mathsf{o}} P \cap \alpha_{\mathsf{o}} Q \neq \{\}$ or $\alpha_{\mathsf{i}} P \cap \alpha_{\mathsf{i}} Q \neq \{\}$.
- Clashes are considered *type errors*.
  - Note: underlying CSP concurrent composition still formally legal. (Recall: $\alpha(P \parallel Q) = \alpha P \cup \alpha Q$; no requirement that $C \subseteq \alpha P$ in $P \setminus C$).
  - So behavior of communication-ill-typed processes is still well defined *in principle*.
  - But: fewer well-typed processes $\Rightarrow$ more properties hold.
  - For well typed processes, can just write $\alpha c = \alpha c(P) = \alpha c(Q)$.

Output

- Let $c \in \alpha_{\mathsf{o}} P$.
- Then for any *expression* e of type αc, we write

  $$c\,!\,e \to P \quad \text{“on } c \text{ output } e\text{, then } P\text{”}$$

  for $c.m \to P$, where m is the value of e.
- Special syntax visually flags event as an output communication.
- Note: we allow m to be the result of a simple computation, possibly including variables from process's local state:

  $$EVENS_i \triangleq out\,!\,(2 \cdot i) \to EVENS_{i+1}$$

Input

- Let $c \in \alpha_{\text{¡}} P$.
- Then for any *variable* v, ranging over αc, we write

$$c\,?\,v \rightarrow P(v) \quad \text{“from } c \text{ input } v\text{, then } P\text{”}$$

for $(w\colon \{x \mid \mathit{chan}(x) = c\} \rightarrow P(\mathit{msg}(w))$.

- Here, dedicated syntax significantly simplifies notation!
- Variable v may be used in the continuation process P, just like a state variable:

$$\begin{aligned} \mathit{COPY} &\triangleq \mathit{left}\,?\,v \rightarrow \mathit{right}\,!\,v \rightarrow \mathit{COPY} \\ \mathit{ACCUM}_n &\triangleq \mathit{left}\,?\,v \rightarrow \mathit{right}\,!\,(n+v) \rightarrow \mathit{ACCUM}_{n+v} \end{aligned}$$

- **Caution:** When CSP is embedded in imperative language, inputs are normally represented as *assignment*, rather than *binding*, to the target variable.

Output vs. input

- Note asymmetry:
  - The prefix $c\,!\,e$ can participate in *exactly one* c-event: environment inputting from c must be prepared to accept any $m \in \alpha c$, or risk deadlock.
  - The prefix $c\,?\,v$ can participate in *all* c-events: environment outputting to c can supply any $m \in \alpha c$ without risking deadlock.
- Again: this is a *convention* specific to channel-based communication.
  - As opposed to general structured events of the form $c.m$.
- Engaging in channel events that cannot be expressed in terms of ! or ? also violates convention.
  - **Ex.** $(c.1 \rightarrow STOP \mid c.3 \rightarrow STOP)$ is wrong (for a process with c as an input channel with alphabet $\supsetneq \{1, 3\}$).
- **Communication law** (another special case of 2.3.1 L7): $(c\,!\,m \rightarrow P) \parallel (c\,?\,v \rightarrow Q(v)) = c.m \rightarrow (P \parallel Q(m))$

Communication in choices

- Convenient to generalize syntax of enumerated choice to allow *communication guards*.
- **Notation:**

$$P = (c_1 ? v_1 \to P_1(v_1) \mid \cdots \mid c_n ? v_n \to P_n(v_n) \mid$$
$$c'_1 ! e_1 \to P'_i \mid \cdots \mid c'_m ! e_m \to P'_m),$$

  where $n, m \geq 0$, $\{c_1, \ldots, c_n\} \subseteq \alpha_{\mathsf{i}} P$, $\{c'_1, \ldots, c'_m\} \subseteq \alpha_{\mathsf{o}} P$, and all channels pairwise distinct.
- Evident expansion into big parameterized choice over union of initial events of component communications.
- **Note:** still deterministic! Nondeterminism arises from *concealment* of internal communications, not from choice.
- **Beware:** some implementations of CSP do not allow *output* guards in choices.

Communication-guarded choices and concurrency

- Find $P \parallel Q$, where
  $P = (c_1 ? x \to P_1(x) \mid c_2 ! e \to P_2)$
  $Q = (c_2 ? y \to Q_1(y) \mid c_3 ! e' \to Q_2)$
- Simply use 2.3.1 L7:

$$
\begin{aligned}
A &= \{c_1.m \mid m \in \alpha c_1\} \cup \{c_2.e\} \\
B &= \{c_2.m \mid m \in \alpha c_2\} \cup \{c_3.e'\} \\
C &= (A \cap B) \cup (A - \alpha Q) \cup (B - \alpha P) \\
&= \{c_2.e\} \cup \{c_1.m \mid m \in \alpha c_1\} \cup \{c_3.e'\}
\end{aligned}
$$

- So $P \parallel Q = \big(c_1 ? x \to (P_1(x) \parallel Q)$
  $\mid c_2.e \to (P_2 \parallel Q_1(e))$
  $\mid c_3 ! e' \to (P \parallel Q_2)\big)$
  - Note: write *matched* communication events with ".", not "!".
- Next, will usually conceal channel c_2 in $P \parallel Q$.

Communication-guarded choices and concealment

- Find $R \setminus \{c_2.m \mid m \in \alpha c_2\}$, where
  $R = (c_1 ? x \rightarrow R_1(x) \mid c_2.e \rightarrow R_2 \mid c_3 ! e' \rightarrow R_3)$
- Use 3.5.1 L10: [**Misprint:** missing in book]
  - If $B \cap C$ *finite* and *non-empty*, then
    $(x : B \rightarrow P(x)) \setminus C = Q \sqcap (Q \,\square\, (x : (B - C) \rightarrow P(x) \setminus C))$,
    where $Q = \sqcap_{x \in B \cap C} P(x) \setminus C$
- For our R, we have $B = \{c_1.m \mid m \in \alpha c_1\} \cup \{c_2.e, c_3.e'\}$;
  $P(c_1.m) = R_1(m)$, $P(c_2.e) = R_2$, $P(c_3.e') = R_3$.
- Since $C = \{c_2.m \mid m \in \alpha c_2\}$, $B \cap C = \{c_2.e\}$, and so we simply have $Q = R_2 \setminus C$ for our R.
- Thus, writing $X^\star$ for $X \setminus C$, we get
  $R^\star = R_2{}^\star \sqcap (R_2{}^\star \,\square\, (c_1 ? x \rightarrow R_1(x)^\star \mid c_3 ! e' \rightarrow R_3{}^\star))$

Examples: streaming

- The copying process:

$$COPY \triangleq left?v \rightarrow right!v \rightarrow COPY$$

where $\alpha_{\mathsf{i}} COPY = \{left\}$ and $\alpha_{\mathsf{o}} COPY = \{right\}$.
Copies a stream from channel *left* to channel *right*.

- Generalization: mapping. Let $f : A \rightarrow B$ be any simple total function. Then

$$MAP^f \triangleq left?v \rightarrow right!f(v) \rightarrow MAP^f$$

(where $\alpha left = A$ and $\alpha right = B$) applies f to every element of the stream before passing it on.

- $COPY = MAP^{id}$, where $id(a) = a$.

Multi-input/output

- An (elementwise) addition process:

$$ADD \triangleq left_1 ? v_1 \rightarrow left_2 ? v_2 \rightarrow right ! (v_1 + v_2) \rightarrow ADD$$

- A replication process:

$$DELTA \triangleq left ? v \rightarrow right_2 ! v \rightarrow right_1 ! v \rightarrow DELTA$$

- A comparison/switch element (useful for parallel sorting):

$$\begin{aligned} COMP \triangleq\ & left_1 ? v \rightarrow left_2 ? w \rightarrow \\ & \quad \textbf{if } v \leq w \textbf{ then } (right_1 ! v \rightarrow right_2 ! w \rightarrow COMP) \\ & \quad \qquad \textbf{else } (right_2 ! v \rightarrow right_1 ! w \rightarrow COMP) \end{aligned}$$

- **Note:** all are *systolic*: on each iteration, input or output exactly one value on each channel.
- But open to potential deadlock! (Consider *DELTA* followed by *ADD*, with suitable renaming to connect each $right_i$ to $left_i$.)

Avoiding unnecessary sequentialization

- (Will use a bit of notation from Chapter 5, though we're not covering most of that chapter.)
- Brute-force solution: insert buffers (= *COPY*) on each channel.
  - Really only needed on channels between multi-output and multi-input processes.
- "Obvious" proper solution: perform independent inputs and outputs concurrently. But how?
- Note: $\mu X.\mathit{left}?v \rightarrow (\mathit{right}_1!v \parallel \mathit{right}_2!v) \rightarrow X$ is syntactically ill-formed: cannot prefix by non-atomic action.
- How about $\mu X.\mathit{left}?v \rightarrow ((\mathit{right}_1!v \rightarrow X) \parallel (\mathit{right}_2!v \rightarrow X))$? Is it well-formed? Well-typed? Does it "work"?

Properly performing concurrent outputs

- Possible approach: explicitly enumerate interleavings

$$
\begin{aligned}
DELTA \triangleq left?v \rightarrow (&right_1!v \rightarrow right_2!v \rightarrow DELTA \\
\mid\, &right_2!v \rightarrow right_1!v \rightarrow DELTA)
\end{aligned}
$$

  (Note: needs output guards in choice.)

- What if we had three output channels?

- Impractical to do transformation by hand, but could have special notation that conceptually expanded to the same thing.

- Alternative: allow concurrent processes to *join* after all have successfully completed their subtasks.

Sequencing

- Introduce distinguished event $\surd$ ("success").
- Special process $SKIP = \surd \rightarrow STOP$. Signals successful termination, then becomes inactive.
- General sequencing: $(P; Q)$, "P followed by Q". Like prefixing, but P need not be a simple event. (Cf. enumerated choice vs. general choice $P \,\Box\, Q$).
- Informal behavior: process that waits for P to signal successful completion ($\surd$), then continues with Q.
  - Note: the $\surd$ from P is concealed to environment of $(P; Q)$, but any $\surd$ from Q is not.
- Have usual sequential-programming laws: $(SKIP; P) = P$, $(P; SKIP) = P$, $(P; Q); R = P; (Q; R)$.
- $$\begin{aligned} DELTA \triangleq left?v \rightarrow ((&right_1!v \rightarrow SKIP) \parallel \\ &(right_2!v \rightarrow SKIP)); \\ &DELTA \end{aligned}$$

Non-systolic network elements

- Input merge:

$$\begin{aligned} MERGE \triangleq (&left_1 ? v \to right ! v \to MERGE \\ \mid\ &left_2 ? v \to right ! v \to MERGE) \end{aligned}$$

  Like (buffered) any2one channel.

- Output distribution:

$$DIST \triangleq left ? v \to (right_1 ! v \to DIST \mid right_2 ! v \to DIST)$$

  Like (buffered) one2any channel.

- Mutable global state:

$$VAR_v \triangleq (set ? w \to VAR_w \mid get ! v \to VAR_v)$$

  Obviously generalizable to more complex operations. Ultimate "object orientation": objects interact *only* by communication.

Buffering

- Simple *COPY* from before is actually a one-place buffer:

$$COPY \triangleq left?v \rightarrow right!v \rightarrow COPY$$

- Suppose $left \in \alpha_o PROD$, $right \in \alpha_i CONS$. Then in

$$PROD \parallel COPY \parallel CONS$$

  PROD can output one (but only one) message even before *CONS* is ready to accept it.

- (What if we actually don't want a built-in buffer, but a wire-like channel that forces synchronization between sender(s) and receiver(s)? *Acknowledgments*; see next time.)

Control vs. data state

- $COPY \triangleq left?v \rightarrow right!v \rightarrow COPY$
- $COPY$ is evidently (why?) equivalent to $BUFFER_{\langle\rangle}$ from the following pair of definitions:

$$\begin{aligned} BUFFER_{\langle\rangle} &\triangleq left?v \rightarrow BUFFER_{\langle v\rangle} \\ BUFFER_{\langle v\rangle} &\triangleq right!v \rightarrow BUFFER_{\langle\rangle} \end{aligned}$$

- Internal process state is always a combination of *control state* (position in code) and *data state* (value of local variables).
- What about multi-place buffers?

A two-place buffer

- Two approaches.
  1. Two single-place buffers, one after the other

$$
\begin{aligned}
COPY1 &\triangleq left\,?\,v \to mid\,!\,v \to COPY1 \\
COPY2 &\triangleq mid\,?\,v \to right\,!\,v \to COPY2 \\
BUFI^2 &= (COPY1 \parallel COPY2) \setminus \{mid\}
\end{aligned}
$$

     where $P \setminus \{c\}$ again abbreviates $P \setminus \{c.m \mid m \in \alpha c\}$.
     Buffer contents is *implicit* and distributed throughout system.
  2. One process with buffer contents as local state:

$$
\begin{aligned}
BUFE^2_{\langle\rangle} &\triangleq left\,?\,v \to BUFE^2_{\langle v\rangle} \\
BUFE^2_{\langle v_1\rangle} &\triangleq (left\,?\,v_2 \to BUFE^2_{\langle v_1,v_2\rangle} \mid right\,!\,v_1 \to BUFE^2_{\langle\rangle}) \\
BUFE^2_{\langle v_1,v_2\rangle} &\triangleq right\,!\,v_1 \to BUFE^2_{\langle v_2\rangle}
\end{aligned}
$$

- Should be obvious how to generalize both to n-place variants for arbitrary $n \geq 1$.
- Can we prove that $BUFI^2 = BUFE^2_{\langle\rangle}$? See next time.

Conditional communication guards

- Note that in $BUFE^2$, we actually treat all *left*-communications in the same way (add input to *end* of buffer), and likewise for *right*-communications (take output from *front* of buffer).
- But we only communicate on *left* when buffer is non-full, and only communicate on *right* when buffer is non-empty.
- Can express this more concisely:

$$
\begin{aligned}
BUF_s^n &\triangleq IN_s^n \,\square\, OUT_s^n \\
IN_s^n &= \textbf{if } \#s < n \textbf{ then } left?v \rightarrow BUF^n_{s\frown\langle v\rangle} \textbf{ else } STOP \\
OUT_s^n &= \textbf{if } \#s > 0 \textbf{ then } right!s_0 \rightarrow BUF^n_{s'} \textbf{ else } STOP
\end{aligned}
$$

(Remember notation: s_0 is head of sequence; s' is its tail.)

- Many CSP implementations have special syntax for expressing conditional willingness to communicate based on boolean flag.

Specifications (1): Stream values

- **Def.** $s \downarrow c = msg^*(s \restriction \{x \mid chan(x) = c\})$.
  - **Ex:** $\langle in.3, out.13, in.5, out.15, in.7\rangle \downarrow in = \langle 3, 5, 7\rangle$.
- **Recall:** P **sat** $S(tr) \iff \forall s \in traces(P).\, S(s)$.
  - Convenient (though potentially confusing) shorthand: all occurrences of c in $S(tr)$ stand for $tr \downarrow c$.
  - [Doubly confusing: writing $f(c)$ for $f^*(c)$.]
- **Ex:** Let $ADD10 = \mu X.in?v \to out!(v + 10) \to X$.
  Then $ADD10$ **sat** $(out \leq (x \mapsto x + 10)^*(in))$,
  where $\leq$ is prefix ordering on traces.
- Let $DELTAS \triangleq left?v \to right_1!v \to right_2!v \to DELTAS$.
  Then $DELTAS$ **sat** $(right_1 \leq left \wedge right_2 \leq right_1)$.
- But processes that only accepted input but never performed any output would satisfy those specs as well...

Specifications (2): Delays

- Also want to express that outputs don't lag too far behind input.
- Def. $s \leq^n t \iff s \leq t \wedge \#t \leq \#s + n$.
- s is a prefix of t, but at most n elements shorter.
- Can then show $ADD10$ **sat** $(out \leq^1 (x \mapsto x + 10)^*(in))$.
- Similarly,
  $DELTAS$ **sat** $(right_1 \leq left \wedge right_2 \leq right_1 \wedge right_2 \leq^1 left)$.
- Buffering may cause longer latency between input and output: $BUFF^n$ **sat** $(right \leq^n left)$, for $n \geq 1$.

Pipes

- Specialized notation for processes with exactly one input and one output channel.
- **Notation:** $P \gg Q$ ("P pipe Q"?)
  - $\alpha_\mathrm{i}(P \gg Q) = \alpha_\mathrm{i} P = \alpha_\mathrm{i} Q = \{\mathit{left}\}$
  - $\alpha_\mathrm{o}(P \gg Q) = \alpha_\mathrm{o} P = \alpha_\mathrm{o} Q = \{\mathit{right}\}$
- Output on P's *right* channel is connected to Q's *left*.
- Formally, $P \gg Q = (f_1(P) \parallel f_2(Q)) \setminus \{\mathit{mid}\}$, where $f_1(\mathit{left}.m) = \mathit{left}.m$, $f_1(\mathit{right}.m) = \mathit{mid}.m$, and $f_2(\mathit{left}.m) = \mathit{mid}.m$, $f_2(\mathit{right}.m) = \mathit{right}.m$.
  - Shorthand: $f_1(\mathit{left}) = \mathit{left}$, $f_1(\mathit{right}) = \mathit{mid}$, analogously for f_2.
- **Ex:** Could write $\mathit{BUFI}^2 = \mathit{COPY} \gg \mathit{COPY}$.
- Can show: $(P \gg Q) \gg R = P \gg (Q \gg R)$, so might as well write as $P \gg Q \gg R$.

Subordination

- Even more specialized notation: $P \mathbin{/\!/} Q$
- $\alpha(P \mathbin{/\!/} Q) = \alpha Q - \alpha P$, where $\alpha P \subseteq \alpha Q$.
- Could define by $P \mathbin{/\!/} Q = (P \parallel Q) \setminus \alpha P$.
- P is a "local server" for Q.
- Particularly useful in connection with process labeling:

$$x : VAR_0 \mathbin{/\!/} y : VAR_0 \mathbin{/\!/} (\cdots\ x.set\,!\,3 \to\ \cdots\ y.get\,?\,v \to\ \cdots)$$

Next time

- Tuesday, December 17: CSP Equivalences.
  - Last theory lecture in 2013; one more after Christmas.
- Nothing new from book, just examples of reasoning about CSP processes and channels.
- Be sure you're fully caught up on CSP book through 5.2.
- Assignment 1 due on Sunday; Assignment 2 out on Tuesday.