The need for failures specifications

The traces specification of a buffer is that its alphabet is $left.T \cup right.T$ for some type T, and that for any trace s, $s \downarrow right \leq s \downarrow left$.This is as much as we can usefully say with a trace specification.

Like any trace specification, this passes some unlikely processes, such as

- $STOP$ which never does anything
- $\mu\, p.left?x \to p$ which accepts inputs, but never gives them back
- $\mu\, p.left.0 \to right.0 \to p$ which transmits only 0's
- $STOP \sqcap COPY$, which may behave perfectly, but may also deadlock.

These all fail on some aspect of *liveness*, which is what failures/divergences specifications are for.

Buffer desires

(i) All a buffer does is input on $left$ and output on $right$. It correctly copies all its inputs to its output channel, without loss or reordering.

(ii) Whenever it is empty (i.e., it has output everything it has input) then it must accept any input.

(iii) Whenever it is nonempty, there is something that it cannot refuse to output.

Formally, $divergences(B) = \{\}$, and

(i) $s \in traces(B) \Rightarrow \quad s \in (left.T \cup right.T)^* \wedge s \downarrow right \leq s \downarrow left$

(ii) $(s, X) \in failures(B) \wedge s \downarrow right = s \downarrow left \Rightarrow X \cap left.T = \{\}$

(iii) $(s, X) \in failures(B) \wedge s \downarrow right < s \downarrow left \Rightarrow right.T \not\subseteq X$

Note that the refusal clause (ii) for "input" is different from clause (iii) for "output".

Characteristic process

As with other *behavioural* specifications, there is a most nondeterministic buffer. The following definition follows the specification closely:

$$\begin{array}{rcl} BUFF_{\langle\rangle} & = & left?x \to BUFF_{\langle x\rangle} \\ BUFF_{s\frown\langle a\rangle} & = & ((left?x \to BUFF_{\langle x\rangle\frown s\frown\langle a\rangle}) \sqcap STOP) \\ & & \Box\ right!a \to BUFF_s \end{array}$$

In general, a process B is a buffer if, and only if, $BUFF_{\langle\rangle} \sqsubseteq_{FD} B$.

Unfortunately, $BUFF_{\langle\rangle}$ is infinite state and therefore of limited use with FDR.

It does, however, have interesting approximations both from above and from below.....

A conservative approximation

Any process which you will be able to prove a buffer on FDR (via a direct check) necessarily has some maximum capacity. Therefore it will refine the following for sufficiently large N: the most nondeterministic buffer that does not accept more than N objects.

$$\begin{array}{rcl} BUFF^N_{\langle\rangle} & = & left?x \to BUFF^N_{\langle x\rangle} \\ BUFF^N_{s\frown\langle a\rangle} & = & (((left?x \to BUFF^N_{\langle x\rangle\frown s\frown\langle a\rangle}) \sqcap STOP) \\ & & \quad \lessdot \#s < N{-}1 \gtrdot STOP) \\ & & \Box\ right!a \to BUFF^N_s \end{array}$$

If $BUFF^N_{\langle\rangle} \sqsubseteq B$ fails, this will either be because B fails the buffer specification or because it accepts more than N things. *You can tell which by inspecting the counter-example – but should remember that the second does not imply it is a buffer.*

Weak buffers

$BUFF^N_{\langle\rangle}$ is a *stronger* specification than $BUFF_{\langle\rangle}$. There is a dual, weaker specification which is always passed then the implementation gets overfull:

$$
\begin{array}{rcl}
WBUFF^N_{\langle\rangle} & = & left?x \to WBUFF^N_{\langle x\rangle} \\
WBUFF^N_{s\hat{}\langle a\rangle} & = & ((left?x \to (\ WBUFF^N_{\langle x\rangle\hat{}s\hat{}\langle a\rangle} \\
& & \quad\quad \lessdot \#s < N{-}1 \gtrdot \\
& & \quad\quad \mathbf{div})) \\
& & \quad \sqcap STOP) \\
& & \Box\ right!a \to WBUFF^N_s
\end{array}
$$

Any buffer will refine all $WBUFF_{\langle\rangle}$, and $BUFF_{\langle\rangle}$ is unique in satisfying

$$
\begin{array}{l}
\ldots \sqsubseteq_{FD} WBUFF^N_{\langle\rangle} \sqsubseteq_{FD} WBUFF^{N+1}_{\langle\rangle} \sqsubseteq_{FD} \ldots \\
\sqsubseteq_{FD} BUFF_{\langle\rangle} \sqsubseteq_{FD} \\
\ldots \sqsubseteq_{FD} BUFF^{N+1}_{\langle\rangle} \sqsubseteq_{FD} BUFF^N_{\langle\rangle} \sqsubseteq_{FD} \ldots
\end{array}
$$

Any finite-state process that *is* a buffer will refine some $BUFF^N_{\langle\rangle}$ and any that is not will fail to refine some $WBUFF^N_{\langle\rangle}$.

Buffer laws

The following laws show the buffer specification's close relationship with $\gg$.

- BL1. If P and Q are buffers, then so is $P \gg Q$. Note (via montonicity) that this statement is equivalent to the fact that

$$BUFF_{\langle\rangle} \sqsubseteq_{FD} BUFF_{\langle\rangle} \gg BUFF_{\langle\rangle}$$

- BL2. If Q uses only the events $left.T \cup right.T$, and P and $P \gg Q$ are both buffers, then Q is a buffer.
- BL3. If P uses only the events $left.T \cup right.T$, and Q and $P \gg Q$ are both buffers, then P is a buffer.
- BL4. If $P \gg Q$ is a buffer and x is not free in either P or Q, then the following process is also a buffer:

$$left?x \to (P \gg right!x \to Q)$$

The most commonly used of the above is, of course, BL1.

Inductive buffer laws

BL5. If x is free in neither P nor Q, which are such that

$$P \gg Q \sqsupseteq_{FD} left?x \to (P \gg right!x \to Q)$$

then $P \gg Q$ is a buffer.

This basically shows that if a piped combination must handle the first item correctly and then return to its initial state, then it is a buffer *with a bit of extra asynchrony introduced by* $\gg$.

But it may not return to its initial state, rather another equally good one:

BL5′. If x is free in neither P_λ nor Q_λ for any λ in Λ, a non-empty indexing set, and for each $\lambda \in \Lambda$ $P_\lambda \gg Q_\lambda \sqsupseteq_{FD} left?x \to \bigsqcap_{\lambda\in\Lambda}(P_\lambda \gg right!x \to Q_\lambda)$
then each $P_\lambda \gg Q_\lambda$ is a buffer.

Simple use of BL5

BL5 and BL5′ are powerful tools for the manual analysis of supposed buffers.

The following is a trivial example: if

$$
\begin{array}{rcl}
P & = & left?x \to right!(2 \times x) \to P \\
Q & = & left?x \to right!(x/2) \to Q
\end{array}
$$

then $P \gg Q$ is a buffer by BL5 because of the following simple calculation:

$$
\begin{array}{rcl}
P \gg Q & = & left?x \to (right!(2 \times x) \to P \gg Q) \\
& = & left?x \to (P \gg right!((2 \times x)/2) \to Q) \\
& = & left?x \to (P \gg right!x \to Q)
\end{array}
$$

See the book for a substantial example of BL5 (majority voting to overcome message corruption).

Example for BL5′

A pair of processes IP and CP, transmitting bits, insert an extra parity bit after each group of 8, and respectively check and discard this bit.

$$\begin{array}{rcl} IP(b,n) & = & \mathit{left}?x \to \mathit{right}!x \to IP(b \oplus x, n+1) \\ & & \lhd n < 8 \rhd \\ & & \mathit{right}!b \to IP(0,0) \\ CP(b,n) & = & \mathit{left}?x \to \mathit{right}!x \to CP(b \oplus x, n+1) \\ & & \lhd n < 8 \rhd \\ & = & \mathit{left}.b \to CP(0,0) \end{array}$$

The combination will deadlock if CP is offered the wrong parity bit at any stage.

Using BL5′

To prove $IP(0,0) \gg CP(0,0)$ is a buffer, we have to deal with all the combinations

$$\{IP(b,n) \gg CP(b,n) \mid b \in \{0,1\}, n \in \{0,1,\ldots,8\}\}$$

If $n < 8$, it is easy to show that

$$\begin{array}{l} IP(b,n) \gg CP(b,n) \\ = \mathit{left}?x \to (IP(b \oplus x, n+1) \gg \\ \qquad\qquad \mathit{right}!x \to CP(b \oplus x, n+1)) \end{array}$$

while $IP(b,8) \gg CP(b,8) = IP(0,0) \gg CP(0,0)$ which in turn equals

$$\mathit{left}?x \to (IP(x,1) \gg \mathit{right}!x \to CP(x,1))$$

It follows trivially that, for each b and $n \leq 8$,

$$\begin{array}{l} IP(b,n) \gg CP(b,n) \sqsupseteq \\ \mathit{left}?x \to \\ \bigsqcap\{IP(b',m) \gg \mathit{right}!x \to CP(b',m) \mid \\ \qquad b' \in \{0,1\}, m \in \{0,1,\ldots,8\}\} \end{array}$$

Buffer laws *versus* FDR

BL5 and especially BL5′ have something in common with th UFP rule: they are a tool for structuring manual state analysis.

By and large FDR is much better at proving results about finite state systems than we are, even with such results!

However,

- It can only decide fixed, finite instances of problems *though see the work on data-independence in Section 15.2.*
- The presence of large buffers in systems, especially over a non-trivial data-type, can lead to overwhelming state explosions uncomfortably fast!
- Results such as BL1 can frequently be used to *combine* a series of smaller results derived from FDR.

Fixed point induction

The following *very general* principle is related to UFP and BL5:

If $P = F(P)$ is any recursive definition such that either *it is guarded/constructive* or *it defines a divergence-free process, and Q is such that $Q \sqsubseteq_{FD} F(Q)$, then we may conclude $Q \sqsubseteq_{FD} P$. In the cases of $\sqsubseteq_F$ and $\sqsubseteq_T$ this principle holds without the need to assume anything about the function.*

Note that, since F is always monotone, $Q \sqsubseteq F(Q)$ is equivalent to the statement

$$Q \sqsubseteq P \Rightarrow Q \sqsubseteq F(P)$$

'If P satisfies (the specification) Q, then so does $F(P)$'. *See the book for the mutual recursion version.*

Fixed point induction and buffers

Hence, if $F(BUFF_{\langle\rangle}) \sqsupseteq_{FD} BUFF_{\langle\rangle}$ for a well-behaved function $F(\cdot)$, then the process defined by $P = F(P)$ is a buffer. Consider

$$B^{+} = left?x \rightarrow (B^{+} \gg right!x \rightarrow COPY)$$

This can be proved constructive via an *ad hoc* argument (see Chapter 4).

Suppose P is a buffer, then

- $P \gg COPY$ is a buffer by BL1, which implies
- $left?x \rightarrow (P \gg right!x \rightarrow COPY)$ is a buffer by L4.
- This proves what is required for fixed-point induction; we can infer that B^{+} is a buffer.

Note that you could perform this proof for any $WBUFF^{N}_{\langle\rangle}$ on FDR, but not the complete one because of infinite state problems.

Buffer tolerance

Let's consider the behaviour of internal (i.e., hidden) channels between processes in a CSP network.

CSP has *handshaken* communication: outputs on a channel can't happen until the inputter is ready. This can have a tremendous effect on process behaviour.

But sometimes it seems irrelevant: it would not matter if we were to put a buffer on the channel. If one can, without meaning that the system fails its specification, we say the channel is *buffer tolerant*.

This can be desirable, because the handshake mechanism may be expensive – especially on long-distance channels.

It can also provide useful links with non-CSP models where buffering of communications is allowed or implicit.

Absolute and relative buffer tolerance

A channel has *absolute* buffer tolerance if putting a buffer on the channel doesn't change the overall semantics of the process at all.

More common is buffer tolerance *relative* to some specification: the system still satisfies the specification when a buffer is added.

For example, the system

$$COPY \gg COPY$$

is not absolutely buffer tolerant, but is (thanks to BL1) relative to the buffer specification.

Other possibilities that can arise are

- Systems tolerant (in either sense) only of buffers up to some given size.
- Systems tolerant of some *weak* buffer $WBUFF^{N}_{\langle\rangle}$, meaning (assuming the specification forbids divergence) that even running asynchronously, no buffer on that channel never gets more than N things in.

Another buffer law?

The following looks as though it ought to be true.......

> If $P \gg R$ and Q are both buffers, with the type of Q matching the 'internal' type of $P \gg R$, then $P \gg Q \gg R$ is also a buffer.

Unfortunately, not: suppose $COPY_2$ and $COPY_3$ are respectively defined over the types $T_2 = \{0, 1\}$ and $T_3 = \{0, 1, 2\}$.

Then setting

$$\begin{array}{rcl} P & = & COPY_2 \,\square\, (right!2 \rightarrow STOP) \\ Q & = & COPY_3 \\ R & = & COPY_2 \end{array}$$

gives a counter-example to the above (the internal type being T_3):

putting a buffer between P and R allows P to deadlock.

The difficulty is clearly *selective input*.

Another buffer law!

If we take away the possibility of R controlling what P outputs to it, the law on the last slide becomes true:

BL6. Suppose $P \gg R$ is a buffer, where the internal type is T (i.e., the type communicated by *right* in P and *left* in R), and that

(a) Q is a buffer of type T, and
(b) If $s\hat{}\langle left.x\rangle \in traces(R)$ and $y \in T$, then $s\hat{}\langle left.y\rangle \in traces(R)$.
(c) If $(s, X) \in failures(R)$ then $\{| \; right \; |\} \subseteq X$ implies $\{| \; left \; |\} \cap X = \{\}$.

Then $P \gg Q \gg R$ is a buffer.

Notice that you would probably *never* expect to find buffer tolerance when you did have selective input.

What the above law says is that any network of the form $P \gg Q$ satisfying the side condition is buffer tolerant relative to the buffer specification!

Layered protocols

If we have designed some pairs of processes (T_i, R_i) such that each $T_i \gg R_i$ $(i > 0)$ is a buffer, and it can be shown that $T_0 \gg M \gg R_0$ is a buffer for some communication medium M, then BL6 and induction imply that

$$T_n \gg T_{n-1} \gg \ldots \gg T_1 \gg T_0 \gg M$$
$$\gg R_0 \gg R_1 \gg \ldots \gg R_{n-1} \gg R_n$$

is one too provided the R_i satisfy the no-selective-input criterion of BL6 and the types match up in the obvious way.

Thus BL6 is fundamental to the sensible design of layered communications protocols.

Communications protocols

One frequently uses communications protocols to factor out errors that may be present in some message transport system.

To create a reliable buffer of the form

$$S \gg E \gg R$$

from an erroneous medium E you need:

- some bound on the worst-case performance of E, and
- to allow for this worst case all the time.

(See the example file `protocols.csp` for many illustrations.)

Communications protocols II

In practice

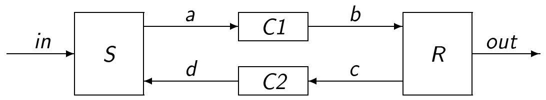
- we may not want to assume a worst case,
- the network may behave reliably most of the time
- so you get a system which is not only not provably correct to the desired degree, but also unacceptably inefficient.
- However, there will probably be an acknowledgement mechanism available (over the same unreliable medium).

Assumptions

Let's assume we have a medium that can lose or duplicate messages, but cannot corrupt or reorder them. It can send structured (i.e., not just bits) messages and acknowledgements,

Note that corruption can be turned into loss via checksums.

The simplest of a family of protocols for this situation is the *alternating bit protocols*, whose network is



Messages and acknowledgements are tagged alternately with 0 and 1 to distinguish new ones from repetitions and duplications.

This is usually described using real-time features such as time-outs, but it is possible to construct a version whose correctness is independent of timing details.

Error-prone channels

In the following there is no limit on loss or duplication

$$\begin{array}{rcll} C(in, out) & = & in?x \to C'(in, out, x) & \\ C'(in, out, x) & = & out!x \to C(in, out) & \text{(correct transmission)} \\ & \sqcap & out!x \to C'(in, out, x) & \text{(potential duplication)} \\ & \sqcap & C(in, out) & \text{(loss of message)} \end{array}$$

The channels for the protocol would be $C1 = C(a, b)$ and $C2 = C(c, d)$.

No matter how many errors one can tolerate, it is clear we could not tolerate an infinite unbroken sequence of them: the system would almost certainly diverge. It is useful to have a version with bounded error behaviour.

$$\begin{array}{rcl} C_N(in, out, r) & = & in?x \to C'(in, out, x, r) \\ C'_N(in, out, x, r) & = & out!x \to C_N(in, out, N) \\ & & \lhd r = 0 \rhd \\ & & (out!x \to C_N(in, out, N) \\ & & \sqcap\ out!x \to C'_N(in, out, x, r-1) \\ & & \sqcap\ C_N(in, out, r-1)) \end{array}$$

Basic sender and receiver

Each has a parameter saying which bit the next *new* acknowledgement/message will have.

$$\begin{array}{rcl} S(s) & = & in?x \to S'(s, x) \\ S'(s, x) & = & a.s.x \to S'(s, x) \\ & & \square\, d.s \to S(1{-}s) \\ & & \square\, d.(1{-}s) \to S'(s, x) \\ R(s) & = & b.s?x \to out!x \to R(1{-}s) \\ & & \square\, b.(1{-}s)?x \to R(s) \\ & & \square\, c!(1{-}s) \to R(s) \end{array}$$

Note that each is always willing both to send and receive.

Divergence *via* unfairness

Even if the channel processes behave perfectly, the system built from the above can diverge: there is nothing to stop (repeated) messages or (repeated) acknowledgements infinitely excluding the other.

This can be avoided by restricting R so that it never does infinitely many of one without the other. Namely, it is made *fair*.

You can either do this by putting a regulator process in parallel with R, or by redesigning it: perhaps making channels b and c alternate.

Divergence could still occur if $C1$ and $C2$ could commit an infinite sequence of errors, but we have assumed they do not......

Correctness

Under these assumptions divergence (livelock) is impossible: if it could

- The divergence would necessarily include infinitely many communications on each of a, b, c, and d.
- All must have the same bit (or else R would output).
- But then S gets its message acknowledged, and so communicates externally.

It can't *deadlock* because S never allows a *cycle of ungranted requests* to appear.

The partial correctness can be deduced from the inequalities

$$\#\overline{a} \geq \#\overline{b} \geq \#\overline{c} \geq \#\overline{d} \geq \#\overline{a} - 1$$

which are derived from the various components.

See the book for more details of all these arguments!

The resulting system is failures/divergences equivalent to $COPY$ (the one-place buffer).