"XCHANs: Notes on a New Channel Type"
\[ \text{XCHAN} = \text{x-channel} + \text{CHAN} \]
Background

• From discussions at Autronica
• Not implemented
• Goal for me was to try to merge asynchronous and synchronous "camps"..
• ..to arrive at a common methodology
• To make it "easier" to comply to SIL (Safety Integrity Level) approving according to IEC 61508 standard for safety critical systems
• Assumed implementation *loosely* based on implemented ideas with EGGTIMER and REPTIMER. ([9] CPA-2009 paper)
XCHAN OF BYTE my_xchan:
XCHAN (100) OF BYTE my_xchan:
asynchronous, buffered
XCHAN (...) OF BYTE my_xchan:

Sender is notified as to its success or "failure"
XCHAN (...) OF BYTE my_xchan:

Sender is notified as to its success on return of send:
- data moved to buffer
- data moved to receiver
XCHAN (...) OF BYTE my_xchan:

Sender is notified as to its "failure" on return of send:
- buffer full
- receiver not present
XCHAN (...) OF BYTE my_xchan:

Sender is notified as to its "failure" on return of send:
- buffer full
- receiver not present

It always returns!
If "failed" to send on XCHAN:
If "failed" to send on XCHAN:

"Not sent" is no fault!
If "failed" to send on XCHAN:

"Not sent" is no fault!

But a contract to send later
If not sent on XCHAN:

- **listen to x-channel** (in an ALT or select)
- **resend** old or fresher value when it arrives
- this send will always succeed
If not sent:

- listen to x-channel (in an ALT or select)
- resend old or fresher value when it arrives
- this send will always succeed
If not sent:

- listen to x-channel (in an ALT or select)
- resend old or fresher value when it arrives
- this send will always succeed

This contract (design pattern) between sender and receiver must be adhered to.
All said!

The rest is really rationale and
Tradition

Send and *forget*
New

Send, if not sent then *don't forget x-channel*
Send and *forget*

*Asynchronous*

Send, if not sent then *don't forget* x-channel

*Asynchronous*
Send and *forget*

Asynchronous

Restart if buffer overflow (bridge metaphor: collapse)

Send, if not sent then *don't forget* x-channel

Asynchronous

Full flow control (bridge only ever full)
Send and *forget*

Asynchronous

Restart if buffer overflow (bridge metaphor: collapse)

.. finding "enough" buffer size..

Send, if not sent then *don't forget* x-channel

Asynchronous

Full flow control (bridge only ever full)
Send and *forget*

Asynchronous

Restart if buffer overflow (bridge metaphor: collapse)

*Forget means* no application handling

Send, if not sent then *don't forget* x-channel

Asynchronous

Full flow control (bridge only ever full)

*Full application handling* (but don't *forget* x-channel)
Send and *forget*

Asynchronous

Restart if buffer overflow (bridge metaphor: collapse)

*Forget means* no application handling

Those programmers..
Send and *forget*

Asynchronous

Restart if buffer overflow (bridge metaphor: collapse)

*Forget means* no application handling

Those programmers... *..could love this..*

Send, if not sent then *don't forget x-channel*

Asynchronous

Full flow control (bridge only ever full)

*Full* application handling (but *don't forget x-channel*)
Send, if not sent then don't forget x-channel

Send and forget

Restart if buffer overflow (bridge metaphor: collapse)

Full flow control (bridge only ever full)

Forget means no application handling

Full application handling (but don't forget x-channel)

Those programmers.. could love this..

..merging asynchronous and synchronous traditions
XCHAN is a new tool (in the *not* empty toolbox!)
Buffering (or not)

1. buffering on-the-way:
   a. after send-and-forget (asynchronous only, **no flow control**)
   b. inside a buffered channel (asynchronous until full, then blocking)
   c. inside a buffered XCHAN (asynchronous until full, then wait for ready)

2. buffering inside a process (task, thread, …) combined with:
   a. no buffering on-the-way with zero-buffered channel
      (blocking synchronous, communication by synchronisation)
   b. buffering on-the-way, see bullets 1a or 1b above
   c. no buffering on-the-way with zero-buffered XCHAN
      (ready synchronous or wait for ready)

3. no explicit buffering at all (with zero-buffered channels)
"If further events are to be possible (such as a channel which can report on whether or not the channel is empty) …" Schneider [10]
"If further events are to be possible (such as a channel which can report on whether or not the channel is empty) …" Schneider [10]
## Output guard and/versus XCHAN

<table>
<thead>
<tr>
<th><strong>XCHAN</strong></th>
<th><strong>Output guard</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Never blocks</td>
<td>May block</td>
</tr>
<tr>
<td>Would have blocked is explicit</td>
<td>One taken, which others could have?</td>
</tr>
<tr>
<td>Next is <em>sure</em></td>
<td>Next is <em>attempt</em> and part of ALT</td>
</tr>
<tr>
<td>Commit to send, not what to send</td>
<td>ALT commits to what to send</td>
</tr>
<tr>
<td>Commitment is state</td>
<td>No such state</td>
</tr>
</tbody>
</table>
## Output guard and/versus XCHAN

<table>
<thead>
<tr>
<th><strong>XCHAN</strong></th>
<th><strong>Output guard</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Never blocks</td>
<td>May block</td>
</tr>
<tr>
<td>Would have blocked is explicit</td>
<td>One taken, which others could have?</td>
</tr>
<tr>
<td>Next is sure</td>
<td>Next is <em>attempt</em> and part of ALT</td>
</tr>
<tr>
<td>Commit to send, not what to send</td>
<td>ALT commits to what to send</td>
</tr>
<tr>
<td>Commitment contains <em>state</em></td>
<td>No such state</td>
</tr>
<tr>
<td>A priori = &quot;first order&quot;</td>
<td>A posteriori = &quot;second order&quot;</td>
</tr>
</tbody>
</table>
Do observe

the source of the x-channel is the run-time system, as for a "timeout-channel"
Architectural leak

is when application code is added (made more complicated) to compensate for missing features at link level

Extra processes
Extra channels
Busy polling
Shared state
...

34
Fast producer, slow consumer and XCHAN

When Server S cannot get rid of *this* data, it can still input more, and finally send *newer* data
"Traditional" solution

Figure 1. Example of an overflow buffer (OBUF)
An XCHAN solution

Figure 2. Buffered XCHAN, as shown in Listing 1 (below)
Listing 1. Overflow handling and output to buffered channels (ANSI C and macros)
Another XCHAN solution

Figure 3. Zero buffered XCHAN
XCHANs as tool to break deadlock cycles
"Knock-come"

**Figure 4.** Traditional "knock-come" pattern
Knock-come(?) with XCHAN

Figure 5. Same pattern with XCHAN
Knock-come(?) with XCHAN

No need to think about knock-come, it comes for free!
Extending XCHANs

- XCHAN sending could return more than "sent" / "not sent"
- x-channel could deliver more than "ready"
Extending XCHANs

- XCHAN sending could return more than "sent" / "not sent" (like "percentage full")
- x-channel could deliver more than "ready" (like "closed")
Extending XCHANs

- XCHAN sending could return more than "sent" / "not sent" (like "percentage full")
- x-channel could deliver more than "ready" (like "closed")

From runtime system, (not process)?
Semantics and implementation

- XCHAN that sends immediately has standard channel semantics
- x-channel has standard channel semantics
- Triggering of x-channel and intermediate blocking in receiver before sender do send, probably cannot be modeled in CSP, and needs help from runtime system. That was at paper time. We now know better: stay tuned
Appendix

Code courtesy of golang–
Appendix

Go has output guards

**Figure 6.** Go example (right channel capacity irrelevant)
Listing 2. Managing without xchan in Go with goroutines

```go
01 func Server (in <-chan int, out chan<- int) {
02     value := 0     // Declaration and assignment
03     valid := false // --"--
04     for {
05         outc := out // Always use a copy of "out"
06         // If we have no value, then don't attempt
07         // to send it on the out channel:
08         if !valid {
09             outc = nil // Makes input alone in select
10         }
11         select {
12         case value = <-in: // RECEIVE?
13             // "Overflow" if valid is already true.
14             valid = true
15         case outc <- value: // SEND?
16             valid = false
17         }
18     }
19 }
```

Appendix
Another code example also shown in paper

There, sender empties receiver end!
– if channel is seen to be full,
Send to itself?

We have not studied whether buffered XCHAN could be wrapped into the sending process, enabling the process to send to itself – but we think this is possible.
Modeling XCHAN

---

PROC xchan.zero (CHAN BOOL ready!, CHAN STUFF in?, CHAN STUFF out!!)

WHILE TRUE
  STUFF s:
  out !!
  SEQ
  ready ! TRUE
  in ? s
  !! s

-- extended output (not recommended)
-- let the writer know when the buffer is full
-- the writer delivers data only when the buffer is empty
-- reader is committed to reading the data

---

This is a zero-buffered version of xchan.

---

Finall
Modeling XCHAN

- Model of buffered XCHAN in occam-pi
- Model of unbuffered XCHAN in occam-pi
- Done as a feasibility study by the editor of the paper, Peter H. Welch, during and after the editing
Thank you!

¿questions?
First page: Drammen signal box (1910), now at the Railway Museum at Hamar

Three pictures: Vemork hydroelectric plant Rjukan (1911), now the Norwegian Industrial Workers Museum (2012)

Small toolbox that I made for Isac