Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013
© 2013 The authors and Open Channel Publishing Ltd. All rights reserved.

Selective Choice 'Feathering' with XCHANs

Øyvind TEIG[1]
Autronica Fire and Security AS[2], Trondheim, Norway

Abstract. This paper suggests an additional semantics to **XCHAN**s, where a sender to a synchronous channel that ends up as a component in a receiver's selective choice (like **ALT**) may (if wanted) become signaled whenever the **ALT** has been (or is being) set up with the actual channel *not* in the active channel set. Information about this is either received as the standard return on **XCHAN**'s attempted sending or on the built-in feedback channel (called **x-**channel) if initial sending failed. This semantics may be used to avoid having to send (and receive) messages that have been seen as *uninteresting*. We call this scheme *feathering*, a kind of low level *implicit* subscriber mechanism. The mechanism may be useful for systems where channels that were not listened to while listening on some *other* set of channels, will not cause a later including of those channels to carry already declared uninteresting messages. It is like not having to treat earlier bus-stop arrival messages for the wrong direction after you sit on the first arrived bus for the correct direction. The paper discusses the idea as far as possible, since modeling or implementation has not been possible. This paper's main purpose is to present the idea.

Keywords. channels, synchronous, asynchronous, buffers, overflow, flow control, CSP, modeling, semantics, feathering

Introduction

The idea of the suggested semantics appeared after reading Tony Hoare's lecture "Concurrent programs wait faster" from 2003 [1]. After some pondering of a situation described there, we saw that it might be viable to use **XCHAN** [2] as a vehicle for a secondary problem not mentioned in Hoare's lecture. The problem is how to avoid having to relate to information of busses that would potentially stop at a bus stop but heading in the wrong destination. It was believed that this could map to a new software pattern: *feathering*.

The word *feathering* is used like "turning an oar parallel to the water between pulls" [3]. Metaphorically a pull is like an **ALT** selective choice. The oar is pushed into the water at one place: only one channel is taken. But we can hear the oar whip the top of the small waves on its way saying "was there, but not interested". So we take the step to name barely touching the small waves as *feathering*.

The author is not aware of **XCHAN** having been implemented in any language or runtime system. The time when a "channel was a channel" seems to be over; the plethora of channel types and channel usage seems to increase. Worth mentioning here is Go's channel usage and semantics [4], which are quite different from what we are used to in the **occam** tradition. The **XCHAN** is here perceived as being an expansion of the traditional channel (**CHAN**) type.

This *feathering* pattern has not been formally modeled or proven. The paper informally introduces the ideas. No literature search has been done to find similar or equal ideas.

---

[1] The author works with concurrent software for fire detection systems, but this "industrial paper" does not necessarily reflect views taken by the company. See: http://www.teigfam.net/oyvind/work/work.html
[2] A UTC Fire & Security company. http://www.autronicafire.com

The **occam** language [5] is used as a vehicle for the reasoning here because of its clean design. However, Listing 1 will show ANSI C macro usage, close to how we might have implemented it at Autronica.

1. Original XCHAN

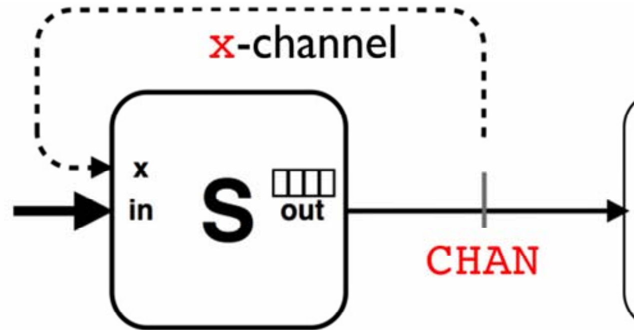

Figure 1. XCHAN is **CHAN** plus **x-**channel.

We repeat the basics from [2]. The whole idea with **XCHAN** is to merge asynchronous and synchronous traditions and allow application level (a sender or server) to handle overflow while it always is allowed to continue (like accept new input). **XCHAN** may be buffered or not; in the latter case a buffer could instead be inside the sender, to allow more control of overflow handling like flushing (see Figure 1). Also shown by Figure 1, an **XCHAN** is like a standard **CHAN** with a bundled **x-**channel feedback channel. This feedback originates in the scheduler[3]. The sending on **XCHAN** always returns with a status telling about *success* or *not success*; so sending never blocks. In the *success* case the message passes through like for a standard **CHAN**, and all is done. *Not success* means that there was no reader or buffer available. In this case the sender is required to listen on the **x-**channel and not send again. When **x-**channel arrives the sender must *commit to send* immediately - old, new or even data ignorantly uninteresting to the receiver. The receiver side does not know whether data arrived on an **XCHAN** or a **CHAN**; and the **ALT** as seen by the application in the receiver is not changed from a standard **ALT**. The discussion uses standard single channel output, not from a selective choice with output guards. The receiver side is a selective choice with input guards only, or a single channel input.

The semantics in summary:

1. **XCHAN** may be used in any type of input or output construct except sending from a selective choice with output guards (not in **occam** anyhow).
2. The sending call never blocks but returns a status value.
3. If the sending call returns *success* then the message has been taken by the channel (if buffered) or by a receiver (if non-buffered). This may be repeated any number of times, without use of the **x-**channel.
4. If the sending call returns *await_commit* then the sender is committed to listen on the **x-**channel amongst its other work. This may be **PRI ALT**ing on the **x-**channel

[3] We will mention the runtime *scheduler* several times in this paper, even if it would be invisible if **XCHAN** and *feathering* were burnt into a language. Still *scheduler* is seen as having some value for the reasoning.

along with other input sources or it may mean regular polling of the **x-**channel. When the **x-**channel is triggered, the receiver is committed to receive (if the **XCHAN** is unbuffered) or the channel has space (if buffered) and the sender must send (old or new data) as soon as possible.

To synchronise the sender and receiver when they are both committed, the receiver's input will be two phased. The first phase will accept the request and the second will take the data. This is an implementation detail invisible to the programmer.

To prepare **XCHAN** for *feathering* (discussed in the following sections):

a. We will extend the sending return value set to also include a *feathered* value, in addition to *success* and *not success*, the latter called *await_commit*.
b. We will let **x-**channel carry data with ***x-****committed*, ***x-****feathered* or ***x-****unfeathered* values (naming convention is to prefix with '***x_***' all that comes in on **x-**channel).

The **x-**channel usage may alternatively be implemented to provide the sender with a precondition and then become involved on every sending [6]. We have not used these semantics here.

2. XCHAN with Feathering Semantics

2.1 Problem: Busses in the Wrong Direction

Hoare's lecture [1] starts by describing the average waiting time for a bus that will take a passenger from A to B. Instead of taking a particular route and wait for it to arrive, Hoare shows that the standard practice in concurrent programming to instead take the *first* bus which could take the passenger to the same place (e.g. bus routes 0 and 2 in Figure 2, Section 2.2) would make the system "wait faster" in Hoare's wording. The waiting time may become shorter than one would assume at first thought, Hoare shows.

This author's first whim to this was to think of each bus route as a software process, and the bus stop as a multiplexer, containing a selective choice. In **occam** this is an **ALT**; in most modern embodiments this is a variant of **select**. Being used to thinking that any unnecessary communication should be avoided; we saw that messages of wrong busses arriving at the bus stop might be a problem. Entering just any first bus could be wrong, as having to rise from the bus stop's bench every time to inspect the line number on the bus perhaps would be unnecessary. Being near-sighted easily causes more work than necessary - and besides, entering a wrong bus could cause a meeting to be lost. So we would not want a paradigm where we would be fed and have to inspect every bus passage or message.

A process could enter an **ALT** and then switch off the set of non-interesting bus arrival messages, by having the Boolean expression in those guards evaluate to **FALSE**. This way the process could wait for only the correct set of buses. This is a standard *channel based programming* pattern.

What happens after the first correct bus has arrived is not of interest. However, if in fact uninteresting buses had arrived during the waiting time, the bus stop process would have to treat those messages of older arrivals *afterwards*, even with the pattern described. The bus processes may be blocking to send - or have sent and forgotten the message in a queue or buffered channel. There is no way to avoid having to flush these messages. *But we could have avoided sending them*.

2.2 Suggestion

It appeared that an **XCHAN** as described in [2] would not help; but might it be a possible vehicle on which to build a mechanism to avoid having to send the old bus arrival messages? We think so.

Unlike the traditional **CHAN**, the **XCHAN** sender already has a means to treat messages that were not sent immediately. As we have seen we suggest this will mostly be interesting when:

1. the receiver end is a pre-conditioned guard in an **ALT**
2. the **XCHAN** is not buffered (see above)

Events participating in an **ALT** are "enabled". When or if one is ready and selected, both it and all the enabled channels are "disabled"; the **ALT** is "torn down". So, with **CHAN** or **XCHAN** there already is a way to handle the *interesting* channels that did not win the **ALT**.

However, in the present implementation of an **ALT**, a guard's Boolean pre-condition (if present) is evaluated and if the result is **FALSE**, then the channel is not touched. To the channel it looks like the input side is not present, like no 'call' to the input[4].

We propose marking **XCHAN** guards that have **FALSE** pre-conditions in a blocked receiving **ALT** as *feathered* - i.e. that this process is not interested in them[5]. A sender attempting to send to an **XCHAN** in that state (or has already attempted to send and been told to wait) is now signalled (on the **x-**channel) about that state. The sender should abandon its message and should not send anything more until the **XCHAN** is *unfeathered*, which happens at the end of the **ALT** (and signalled on the **x-**channel).

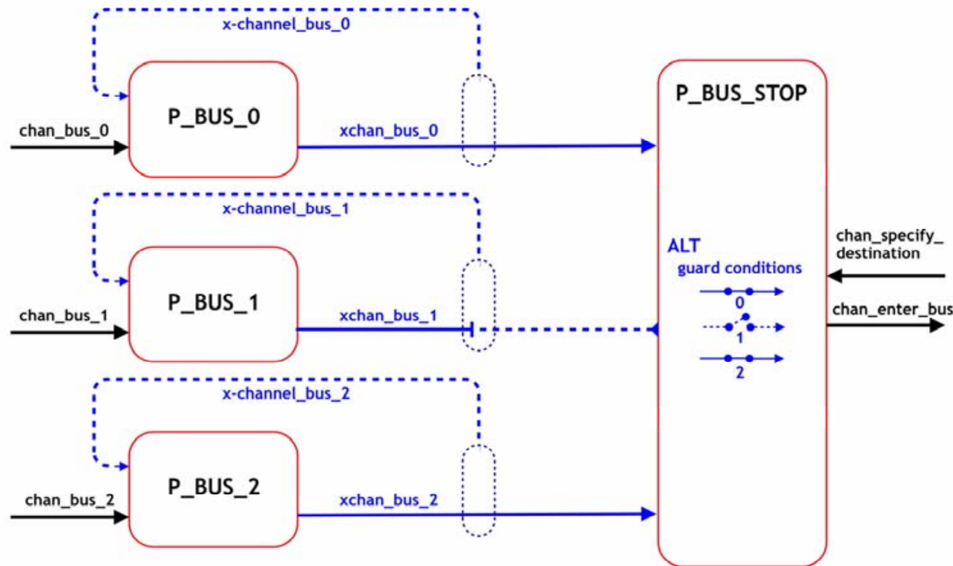

Figure 2. XCHAN (array of 3) and feathering, with only buses #0 and #2 possible to ride.

Figure 2 shows an example of this. The *bus stop* process is waiting on its **ALT** for either buses 0 or 2 (the channels from those buses have **TRUE** pre-conditions) – but is, on this occasion, not interested in bus 1 (whose channel is pre-conditioned **FALSE**).

[4] This author is not aware of any implementation where it is any other way.
[5] This is information that is 'lost' in the traditional semantics of a channel.

2.3 Feathering Semantics (Discussion)

With *feathered* semantics we are given a means not to have to send that unnecessary data. This section is a discussion. Section 2.4 tries to summarise.

A sending is done with an output attempt in the **XCHAN** the usual way. However, we need to declare that we want *feathering* semantics.

2.3.1 Successful immediately

Sending attempt will go through as successful if the receiver was able to receive. As mentioned, the **x-**channel is not involved.

2.3.2 Unsuccessful

If sending were unsuccessful there are two outcomes: either there was no receiver or the channel had been *feathered*, meaning that the receiver is already in an **ALT** waiting on other events and explicitly excluding the sender's **XCHAN**. In both cases the sender receives this status (*success*, *feathered* or *await_commit*) after the **XCHAN** output. It never blocks.

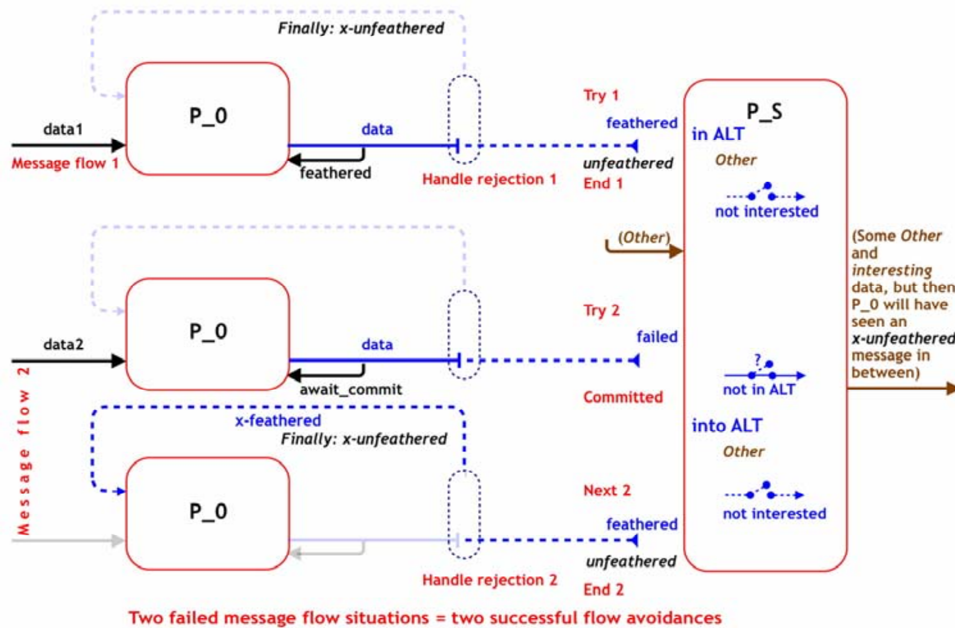

Figure 3. Two mind map scenarios that show message avoidance.

Figure 3 resembles Figure 2, but in addition to topology (process, data flow) it also attempts to overlay sequences. The diagram may be seen as an informal mind map showing message avoidance only. The first scenario (top) shows an initial *feathered* which is terminated with an ***x-**un**feathered***. The second scenario shows first an initial *await_commit* (centre) followed by an ***x-**feathered* and final ***x-**un**feathered*** (bottom).

The sender is finished if it receives *success*. It will handle *feathered* at application level, so it is per design what it would do with the message that seemed to be uninteresting. We shall see below that *feathered* is not a state in which the application has finished with its attempted send.

2.3.3 Towards feathering

If the sender gets an *await_commit* return then it is obliged to listen on the **x-**channel. If the receiver commits to the read (or buffer space in the **XCHAN** becomes available), then ***x*-*committed*** will be sent and the sender must send as soon as possible. If it receives ***x*-*feathered*** then it means that the receiver has already *feathered* the channel, and in this way told any potential sender (of which it is not aware) that it is not interested. The sender should treat this as it will with the *feathered* status return.

2.3.4 Ready for next with *x*-*unfeathered*

Observe that when a message has been "warned away" with *feathering* we propose that there should be no undo; that it shall not be possible not send *anyhow*. If the receiver end of the channel later should becomes ready that should in the default semantics be for a next message.

Feathered status is never returned when the receiver is not in the act of entering or being in an **ALT**, so it must be removed when the **ALT** is torn down.

It is important that there should be no divergence (livelock) between the sender and the receiving **ALT**. The allowable action for a sender after seeing a *feathered* receiver end should probably be the same for a *feathered* status return or an ***x*-*feathered*** value. In the first case the receiver is already in an **ALT** with that **XCHAN** guard pre-conditioned **FALSE** (we have already mentioned that we would never return *feathered* if the receiver is anywhere else in its code). Retrying to send would in that case cause divergence, as the same *feathered* status return would be repeated infinitively. The same would happen with an ***x*-*feathered*** value. Trying to send immediately would therefore also cause a repeated *feathered* status return. Therefore, after any *feathered* status we need a channel 'link level' intermediate state to avoid this. During this state it is not allowed to resend. To terminate this state we propose to wait for an ***x*-*unfeathered*** value.

This is sent when the **ALT** has been taken by one of the other inputs after having waited; on 'tearing down' the **ALT**. It will send this to all the *feathered* **x-**channels.

2.3.5 Observation

The described scheme certainly motivates a need to model *feathering* semantics and to see if there are any unseen pitfalls. Also, this pattern may be asserted with a proper rule for the compiler to (usage) check.

Section 2.4 summarises the proposed semantics of *feathering*. Section 2.5 gives a coding template for a process sending on an **XCHAN** with *feathering*. Further discussion is in Section 3.

2.4 Feathering Semantics (Summarised)

This is a summary of the suggested semantics, not a formal model.

1. *Feathering* semantics inherits **XCHAN** semantics (see Section 1)
   a. However, output and input constructs are limited by point 2 (here)
   b. This may not include buffered **XCHAN**; we are uncertain about the usability of *feathering* semantics in that case.
2. *Feathering* is possible if the receiver end of **XCHAN** being processed by an **ALT**, not a single channel input. However, the sending side is a single channel output, not part of an **ALT** with output guards (not possible in **occam**).
3. *Feathering* semantics is specified with a parameter in the **XCHAN** send call.
4. The *feathered* status is returned to a sender that is trying to send when a receiver is in an **ALT** and the requested channel has been tagged by the receiver as *not-interesting* (i.e. its pre-condition is **FALSE**).
5. The *feathered* status is sent to a sender on **x-**channel if it has been trying to send but got *await_commit* reply; when the receiver enters an **ALT** and the requested channel is being tagged as *not-interesting*.
   a. The **ALT** in the receiver sends these *synchronous* messages to all pre-condition excluded feathered **XCHAN** senders at the end of the **ALT** setup if the **ALT** blocks - i.e. it is not immediately taken by another guard (channel, timeout or **SKIP**).
   b. None of these will block indefinitely, as all the receivers will already have committed to listen on **x-**channel.
6. Whenever a sender knows that a channel is *feathered* it shall obey the rule not to resend before an *unfeathered* message has been received on **x-**channel.
7. The *unfeathered* status is delivered to a *feathered* **x-**channel when the **ALT** is later on taken (by another guard) and 'torn down', in the same synchronous scheme as described above (5.a-b)
8. The **x-**channel will only carry an **x-***unfeathered* after a *feathered* situation.
9. The **x-**channel will only carry **x-***feathered* or ***x-****committed* after an *await_commit* status return on the initial sending call.
10. A receiver could possibly do a system call to learn if a message in fact did get rejected. This information could alternatively be delivered on an "**n**-channel"[6] that could be "parallel" with the **XCHAN**'s input on the receiver side. This probably is a complicating matter since type of channel is transparent on the receiver side. We will not discuss this here.

[6] Like "**x**-channel" on the sender side of **XCHAN** and "**n**-channel" on the receiver side of **XCHAN**.

2.5 Code example

```
01 CP->Tag = READY; // READY, SUCCESS, AWAIT_READY, FEATHERED
02 while (true) {
03   PRIALT ();
04     ALT_CHAN_IN (X_CHANNEL, X_Tag); // X_COMMITTED,
05                                     // X_FEATHERED, X_UNFEATHERED
06     ALT_CHAN_IN (CHAN_DATA_IN, Value);
07   ALT_END (); // delivers ThisChannelId
08
09   switch (ThisChannelId) {
10     case X_CHANNEL: { // After CHAN_OUT ret AWAIT_READY or FEATHERED
11       if (X_Tag == X_FEATHERED) {
12         ...  handle not interested
13         CP->Tag = FEATHERED;                  // stop
14       } else if (X_Tag == X_COMMITTED){
15         CHAN_OUT (XCHAN_DATA_OUT,Value,NIL);  // will succeed
16         CP->Tag = READY;                      // finished
17       } else { // (X_Tag == X_UNFEATHERED)
18         CP->Tag = READY;                      // finished
19       }
20     } break;
21     case CHAN_DATA_IN: {
22       if ((CP->Tag == AWAIT_READY) or (CP->Tag == FEATHERED)) {
23         ...  handle overflow (decide what value(s) to discard)
24       } else { // (CP->Tag == READY)
25         CP->Tag = CHAN_OUT (XCHAN_DATA_OUT,Value,ALLOW_FEATHERING);
26         if (CP->Tag == SUCCESS) {
27           CP->Tag = READY;                    // finished
28         } else if (CP->Tag == FEATHERED) {
29           ...  handle not interested
30         } else { // (CP->Tag == AWAIT_READY)
31         }
32       }
33     } break;
34   }
35 }
```

Listing 1. Overflow and 'feathered' handling on an **XCHAN** (ANSI C and macros).

The listing above basically follows listing 1 in [2]. **CP** is the Context Pointer to process data that has been allocated on the heap. It shows an example of a server that always can input new data since output is non-blocking. Data arriving on **CHAN_DATA_IN** is always accepted (line 21) within a bounded time (since no code in listing blocks and an **X_CHANNEL** signal can only happen at most twice by a failed send on **XCHAN_DATA_OUT**, in response to the previous input from **CHAN_DATA_IN**). If there is no overflow situation then line 25 attempts to send. If this succeeds immediately with **SUCCESS**, it is finished. If the result is **FEATHERED** then handle the fact that the receiver was "not interested" immediately (e.g. abandon processing of this input). If the result is **AWAIT_READY** or **FEATHERED** then loop around to wait for an **X_CHANNEL** signal; or more from **CHAN_DATA_IN**. Observe that in the example we always include **X_CHANNEL** in the **ALT**; this is ok since it will not be signaled out of order (though it might be more efficient to exclude it, by pre-conditioning, when not expected). If the **X_CHANNEL** signal is **X_FEATHERED**, then handle this "not interested" state (responses in lines 12 and 29 should be the same). If it is **X_COMMITTED**, send the latest received value (line 15). Once **FEATHERED**, it cannot retry sending before **X_UNFEATHERED** is received. Observe that we now use a **PRIALT** in 03, so **CHAN_DATA_IN** cannot jam **X_CHANNEL**. The code has no explicit error handling and there are no compiler usage checks.

3. Discussion

3.1 Feathering is "Low Level"

The receiver is able to tell to the sender at a potential synchronisation point that it is '*not interested*'. However, it is not able to give any *reason*. It is a basic, low-level type of rejection. This way the **ALT** construct need not be changed. It may also be discussed whether alternatively 'sending' some reason across is counter-productive. One must probably take care that this does not become an opposite direction channel hidden in the channel. We therefore propose only the lowest level: the implicit "not interested". (However, a higher level channel with these properties could potentially be part of some other software pattern.)

There is no other type of state exchange either. The sender may not know the receiver's state; like a priori information whether a channel would have become *feathered* or if the receiver is just doing something else; like being in a session with a third party and that it certainly would want to receive the pending message later. Coupling between processes should be kept at a minimum, and *feathering* does not seem to increase it – rather the opposite – with its implicit subscriber mechanism and, after all, less point-to-point messages between the parties.

Instead of sending on **XCHAN** with a parameter, we could have defined a new type of channel called **FXCHAN**, ("**F**eathering **XCHAN**") with *feathering* as standard semantics. In that case the receiver would know about its *feathering* capability. However, a generic type would make the semantics selectable at run-time (although with some cost: **FALSE** pre-conditioned **XCHAN** guards always need processing). Usage rules would be a little more complex with **XCHAN** plus parameter, certainly also depending on whether this parameter is a variable or an invariable in the actual scope.

With the suggested scheme the receiver will not know if it will implicitly stop a message by *feathering*, or if *feathering* has any impact. It would only know that this new **XCHAN** or **FXCHAN** has a potential to be *feathered*.

Sending outdated messages across would also need a design criterion as to decide whether the message is outdated. This also implies some degree of coupling between the processes (like a common clock).

Like any language primitive there may be uses for *feathering* and cases where it is not applicable or plainly wrong.

As mentioned we have not studied the alternative precondition semantics of **x-**channel [6].

3.2 Anti-Deadlock

We argue that the anti-deadlocking property of **XCHAN** is kept by this additional semantics. We have shown that the extra synchronisation needed to feed **x-***feathered* or **x-***unfeathered* back on the **x-**channels is not blocking, asserted by rules and contracts. However, the associated scheduling may cause the sender in a 'new round' to block on other channel outputs, but this will be part of another communication graph. Therefore **XCHAN** in itself still breaks any cycle (and 'removes' deadlock).

Setting of the *feathered* status in the **ALT** construct is done by calls to scheduler code. It has full control of all parties (no pre-emption), so there should be no race between the two sides of **XCHAN**. However, this needs to be further studied for multicore or distributed systems.

3.3 Run-Time Penalty

As we see, there would have to be some reporting back to the sender on every channel that is *feathered*. One might argue that the goal not to send unnecessary data is compromised when some other (data or not) will be sent instead. Not having to send lots of, for example, 1500 bytes long messages would probably save bandwidth. However, the language design to offer the receiver a pattern of an implicit subscriber mechanism is perhaps even more interesting. It is this idea that has driven the writing of this paper.

Some communication and associated scheduling will be removed. Since nothing is free, there is a cost associated with supplying the information enabling this removal. The extra two communications for **x-***feathered* and **x-***unfeathered* on the **x-**channels will after all interfere with state in the sender, which is then allowed to take proper action, like reporting back to some other component.

There is no busy-polling in any of the algorithms.

3.4 Feathering Requires "Visible" Pre-Conditions on **ALT** Guards

An **occam ALT** with conditional guards may be transformed to an **IF** structure:

```
01 ALT                           -- feathering semantics hidden
02   condition.0 & in.0 ? x.0
03     ...  response 0
04   condition.1 & in.1 ? x.1
05     ...  response 1

10 IF                            -- no feathering
11   condition.0 AND condition.1
12     ALT
13       in.0 ? x.0
14         ...  response 0
15       in.1 ? x.1
16         ...  response 1
17   condition.0                 -- condition.1 must be FALSE
18     SEQ
19       in.0 ? x.0
20       ...  response 0
21   condition.1                 -- condition.0 must be FALSE
22     SEQ
23       in.1 ? x.1
24       ...  response 1
```

Listing 2. *Feathering* loss of semantic equivalence (**occam**).

The programmer must be aware that, for *feathering* semantics, the pre-condition on the guard must be "visible" in the code. Without *feathering*, the two blocks of code (lines 1-5 and 10-24) above are equal. With *feathering*, the **ALT** in line 12 will never take part in any *feathering*; neither will the two **SEQ** blocks starting at lines 18 and 22. Since *feathering* means to have a "tap to turn off", all should be obvious: there is no "tap" or boolean expression. The sender does not know which of the methods are used; but the block starting in line 10 will return neither *feathered* status nor **x-***feathered* nor **x-***unfeathered*.

The receiver may use this loss in semantic equivalence to ensure that *feathering* is off (although the **IF**-version gets exponentially more complex to program as the number of guards in the **ALT** increases). Expressions evaluated to **FALSE** are not so easily spotted. The code on the sender side should probably not be implemented with this knowledge: an

XCHAN's receiver may at any time decide to engage in *feathering*. The compiler or linker could inform about this. This would probably also go for visible invariant conditionals. But there is nothing wrong with coding that at a certain time *all* components are hard coded as *interesting*.

3.5 Feathering Adds Non-Determinism

Whether a message gets sent or not depends on when the attempt is made. This adds non-determinism. However, *feathering* should only be used when this non-determinism is deliberate. Just as using **XCHAN** opens for application control of message handling (like data loss) on the sender side of the **XCHAN**, *feathering* takes the receiver's face value for throwing a message. "Not listening on that ear" really means "please, drop it" – by design.

By hiding this non-determinism inside a block that will equally allow different behaviour, *feathering* should not hinder any formal proofs.

4. Alternative

We have not discussed whether *feathering* solves real problems where programmers have longed for a solution.

An alternative to this mechanism could be an *explicit* subscriber mechanism. Mixing real life and software we could say that the bus-stop subscribes to the busses, on behalf of the passenger, which of the busses that are interesting. When the correct bus has been taken, all busses are unsubscribed to. This way there are no arrival messages to flush, provided there are no possible race conditions in the algorithm used. Some of us may have missed a bus if it were so close to the bus in front that we did not see the badge.

Feathering may also be seen as an implicit embodiment of the *publish-subscribe pattern* [7]. *Feathering* status would be an a *not publish* status handled to the publisher (sender) when the subscriber (receiver) implicitly has decided not to *subscribe*. However, differently from the *publish-subscribe pattern* there is no explicit message or calls done by the subscriber to subscribe to publishments or to unsubscribe. Examples for the usability of *feathering* may probably be collected from the *publish-subscribe pattern* examples.

Since *feathering* may be considered a low-level implicit subscriber mechanism, it has no messages to flush. The algorithm we have described here should be without race condition, but as mentioned – this is not formally proven.

5. Summary

The proposed **XCHAN** [2] already provides actionable feedback to the sender on the state of the receiver. This proposal extends that information (beyond success/fail/ready) to include *feathered* (i.e. not interested, do not resend) and *unfeathered* (which ends a period of feathering). *Feathered* signals are sent (to relevant potential senders) if the receiver's **ALT** blocks (and the relevant **XCHAN** guard is excluded by a **FALSE** pre-condition). Unfeathered signals are sent upon exit from the **ALT**.

6. Conclusion

This paper takes the rather drastic attempt to change the state of an **XCHAN** when, in the standard **occam** semantics of **CHAN**, there would be no state change when *not* touching a

guard of an **ALT**. The suggested *feathering* semantics exploits information in the running code that a channel's receiving end has been excluded from an **ALT** – and propagates this information to a potential sender. This was our starting point. We have suggested a usage of this idea that could be of interest in some systems.

The sum of the benefits of **XCHAN** and *feathering* may justify adding these features to a concurrent language.

This paper is a study of usage and discussion of the possible semantics, as well as some implementation issues. We have not formally proven that *feathering* is possible, but the reasoning above should be a starting point for such proofs.

We hope that the ideas will cause language designers to take *feathering* further. Perhaps the pull of the oar will take the ideas safely to port?

Acknowledgements

I am grateful to Professor Sverre Hendseth of NTNU, Trondheim, for his encouragement and enthusiasm about the theme. I also thank the peer review readers and the editor for thoroughly pointing out inconsistencies and more subtle matters.

References

[1] Tony Hoare, Concurrent programs wait faster. Microsoft Research, 2003. See http://research.microsoft.com/en-us/people/thoare (ref. notes on the first two pages)
[2] Øyvind Teig, XCHANs: Notes on a New Channel Type. In: P.H. Welch et al., *Communicating Process Architectures 2012 (CPA-2012), Proceedings of the 34th WoTUG Technical Meeting*, ISBN 978-0-9565409-5-9. Open Channel Publishing Ltd., 2012, pp. 155-170. See http://www.teigfam.net/oyvind/pub/pub_details.html#XCHAN
[3] 'Feathering', defined in OneLook, see http://www.onelook.com/?w=feathering&ls=a
[4] Google Go programming language by Robert Griesemer, Rob Pike, and Ken Thompson. See "The Go Programming Language Specification" at http://golang.org/ref/spec
[5] INMOS Limited, Occam 2 Programming Manual, Prentice-Hall International, ISBN 0-13-629312, 1988. See http://www.wotug.org/documents.shtml
[6] Peter H. Welch. An occam Model of XCHANs, https://www.cs.kent.ac.uk/research/groups/plas/wiki/An_occam_Model_of_XCHANs , 2013.
[7] Publish-subscribe pattern. See http://en.wikipedia.org/wiki/Publish–subscribe_pattern