Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

XCHANs: Notes on a New Channel Type

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

(3 typos fixed, 31.Aug.2012)

Abstract. This paper proposes a new channel type, **XCHAN**, for communicating messages between a sender and receiver. Sending on an **XCHAN** is asynchronous, with the sending process informed as to its *success*. **XCHAN**s may be *buffered*, in which case a successful send means the message has got into the buffer. A successful send to an unbuffered **XCHAN** means the receiving process has the message. In either case, a failed send means the message has been discarded. If sending on an **XCHAN** fails, a built-in feedback channel (the ***x****-channel*, which has conventional channel semantics) will signal to the sender when the channel is ready for input (i.e., the next send will succeed). This **x**-channel may be used in a **select** or **ALT** by the sender side (only input guards are needed), so that the sender may passively wait for this notification whilst servicing other events. When the x-channel signal is taken, the sender should send as soon as possible – but it is free to send something other than the message originally attempted (e.g. some freshly arrived data). The paper compares the use of **XCHAN** with the use of output guards in **select**/**ALT** statements. **XCHAN** usage should follow a design pattern, which is also described. Since the **XCHAN** never blocks, its use contributes towards deadlock-avoidance. The **XCHAN** offers one solution to the problem of overflow handling associated with a fast producer and slow consumer in message passing systems. The claim is that availability of **XCHANs** for channel based systems gives the designer and programmer another means to simplify and increase quality.

Keywords. Channels, synchronous, asynchronous, buffers, overflow, flow control, CSP.

Introduction

With the advent of the Go programming language [1], channel communication based on the CSP paradigm [2] again seems to have a potential to becoming mainstream. A previous attempt was with the occam programming language [3-5], which gained significant industrial traction during the 1980s and early 1990s (and, of course, is still being developed and applied in academic research [6-8]). Whether new languages with concurrency based on CSP repeat this success remains to be seen.

Nevertheless, channels come in more flavours than the simple channels of occam. This paper suggests a new type of channel that could be added to CSP libraries or become a new primitive in future versions of CSP-based languages. We call this channel an **XCHAN**, which contains a communication channel and a built-in *channel-ready-channel* (the ***x-channel***) for flow control. An **XCHAN** may be sent to (asynchronously) and received from, but the sender must listen on the **x**-channel (usually in an **ALT**/**select**) when sending fails. An **XCHAN** may be buffered.

[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, see http://www.autronicafire.com.

The **XCHAN** idea came as a result of internal discussions at Autronica. As such it is still an idea, not having been implemented. We did this to try to merge the asynchronous and synchronous "camps", to arrive at a common methodology. This author believes that having a common tool is important, and the **XCHAN** is a suggestion. The author argues in this paper that it would work, partly based on similar thinking with **EGGTIMER** and **REPTIMER** as described in [9].

1. Not All New Ideas

Before we delve into the details, we show that the basic idea is not entirely new. The feedback channel is briefly mentioned in [10], page 211, in an example about buffers:

> *"If further events are to be possible (such as a channel which can report on whether or not the channel is empty) ..."*

However, the idea does not seem to be developed further in [10].

The Linux/Posix operating systems have a way to handle flow control in a pipe [11]. There, writing to a pipe may return **EAGAIN** or **EWOULDBLOCK** as error. When there is room in the pipe again, a **select** would then return a list containing the pipe's file descriptor. Observe that **select** may contain sending and receiving roles. However, we have in the literature failed to see the Linux primitives comparable to CSP channels. On the contrary, in *libcsp*[3] we see a CSP channel implemented on top of Posix needing two main mutexes, two queue mutexes and two condition variables [12]. The libcsp paper argues that adding this on top of Posix was done *"because the CSP approach is compositional, unlike the Posix Threads approach, allowing much clearer reasoning about program behaviour"*.

Also, observe that a Linux pipe is a byte-wise "data stream" and does not constitute "message passing". Bytes of messages are concatenated in the pipe. This implies that a receiver could read past a message, causing any percentage of a message to be seen. A channel delivers full messages and nothing more.

We will not address pipes any further.

Languages that do have **select** with input and output guards may be able to handle the situation with less "bricks" than for **ALT**/**select** with only inputs allowed. This is not a side effect, but by design. We will come back to this, and see proper examples with Go, in the Appendix.

Observe that the **XCHAN** takes the semantics further than mere *"testing for readiness"*, seen in several existing channel implementations. Testing implies busy polling. The **x**-channel delivers its information (that the XCHAN is ready) on a channel proper and the sender process does not need to poll.

2. The Problems

We show two problems. The first is to make proper application between a fast producer and a slow consumer, where the application must have full control of the overflow situation. The second is the continuous search for simpler software patterns that guarantee deadlock freedom.

[3] libcsp has later been expanded to libcsp2 by Bernhard H.C. Sputh.

3. Buffering (or not)

In this section, we discuss:

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)
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
3. no explicit buffering at all (with zero-buffered channels)

as distinct matters. A software mechanism that connects a fast producer (that we are not able to control) to a slow consumer may be implemented by buffering in a process, buffering on-the-way or a combination.

This author's industrial life experience is to add (data-less or programmed data loss) asynchronicity (bullets 1 or 2) when needed (most probably at the interface with the external systems) and full synchronicity (3) elsewhere (most probably inside a system). The author has, as a corollary, tried to avoid strict synchronicity at the edges and avoid asynchronicity inside.

For all cases, we would think of buffering as storing individual messages of dynamic lengths.

Pipes containing both full and parts of messages in the same byte stream (as we have said) are not treated. Therefore, building parsers that pick out the next message from a byte stream is also not discussed. Also, we do not consider physical buffering by bit streams propagating in a cable, where about one kilometer of a one Gbit/s stream contains some 625 bytes[4].

Observe that buffering is no guarantee for deadlock freedom. And a system built by non-buffered CSP type channels (3) may be designed deadlock free.

And on-the-way (1) is no panacea for high speed. A system built by non-buffered CSP type channels (3) may also be fast. Blocking on a channel when the channel is not ready may be just as good a paradigm as continuing doing the next thing after any send. It depends.

This paper adds 1c and 2c:

1. buffering on-the-way:
    c. inside a buffered **XCHAN** (asynchronous until full, then wait for ready)
2. buffering inside a process (task, thread, …) combined with
    c. no buffering on-the-way with zero-buffered **XCHAN**
    (ready synchronous or wait for ready)

*3.1 Why Not Buffer (*on-the-way*)?*

Asynchronous buffered communication between processes has been judged by some to be conceptually worse than synchronous non-buffered communication. Buffer and heap overflow handling have often caused run-time system exceptions and, therefore, the unsafe possibility of not knowing how much was really needed before it was too late. Buffering may also add memory moves: from one move (*process-to-process* context) for the synchronous case to two moves (into and out of buffer) for the asynchronous case.

[4] Electricty moves 20-23 cm per nanosecond in an electric conductor.

In a buffered system with send-and-forget without flow control, combined with a common message pool where several clients may request services from a server, there is additional complexity. In such a system, all messages would arrive in a common message queue. A session initiated by one client does not stop others from also trying to initiate their sessions. This may cause the server to have to store messages from other clients and, later on, process those messages when the original session ends. The server has to schedule its own jobs, based on the messages set aside, and "becomes its own scheduler". This again causes increased coupling between the clients and the server, since the agreements being made easily cause unnecessary knowledge about the internal coding of the other part.

*3.2 Why Buffer (*on-the-way*)?*

However, there may be reasons to have buffering. Examples could be:

- At terminal points, where speed of incoming data needs to be related to available memory and processing capacity. The terminal-side process in this case often is called a "driver".
- When a process writes to a file and wants to be decoupled from mechanical disk handling time.
- If context switches are expensive and one cannot afford the extra cycles that synchronous communication may cause. (However, if there is mostly one receive for each send, this collapses to the same amount of switching.)

Some of these reasons may be more or less in scope:

- When the communication libraries only offer buffered communication and this is the only option. Sometimes one does not have the choice.
- When tradition is rooted in asynchronous design. An example could be between communicating state machines, often implemented with asynchronous coupling. Basically, since a set of communicating state machines each only draws the next step from the set of global steps, it does not matter if a process state machine does *not* block on an outgoing transition when it in any case has to wait for the next incoming event that would cause the next transition. Therefore a set of communicating state machines may be realised with either asynchronous or synchronous coupling [13].

This paper suggests that making asynchronous communication *almost* as easy to use as send-and-forget *pretends* to be, and blocking *equally* easy - for the added benefit of full application process control and no busy polling, by using **XCHAN** with an integrated flow control feedback channel - may be a step forward in the art of concurrent programming. We will now go into some more detail.

3.3 Send-and-forget

Send-and-forget is a term where a process performs a send call, and the call returns immediately. In some systems there is a return value indicating whether there was room in the buffer.

One could also send-and-forget on a synchronous channel with no buffering in the channel, provided one knows that it would not block, that is the receiver is always ready. Attempting to send on a synchronous channel into a finite set of chained processes should only be done if there is some kind of feedback in the chain. This is done with one extra backwards channel for flow control ("c5" in Figure 1). The processes that **ALT** on this

"ready channel" will also contain the input channel in that **ALT**. Overflow handling is handled by the application, but there is no need for any busy poll timeout to retest. When there is space, the ready channel will inform.

In both cases the capacity of the buffer or the number of chained processes cannot be infinite. This asynchronous communication must sooner or later relate to overflow.

There are two ways to think about this, using some pattern to handle overflow or setting the capacity to "large enough".

If overflow, attempting to send into a buffer will then either cause the application process to drop that message (and hopefully send information about this when the buffer has space), or retry according to some scheme. Often a timer will do this active retry.

Many systems are designed such that all processes always do send-and-forget, with no channel concept. The fact that *"the world is asynchronous"* is believed to also mean that all the processes must therefore communicate asynchronously, instead of limiting asynchronicity to where it is needed, like at an interface with "the world". In those systems it seems inconvenient to handle a *"full buffer"* return, simply because one would not really know what to do with it (since any such overflow seems just as serious as any other). So, one tries to make sure that the buffer capacity is *"large enough"*, and the buffer system will then just crash and restart the system if it overflows.

Finding *"large enough"* is easier for a channel realised as a point-to-point pipe than for a common buffer pool. Many concurrent systems are designed with the latter. Using a metaphor: when more cars arrive on a bridge than the bridge is designed for, it collapses and all cars fall into the river. A millisecond after, the bridge has been rebuilt (the microcontroller has restarted), but the cars and people in them have been lost. Observe that there are no lights controlling the access to the bridge, and there are several roads connected to the bridge. Even worse, each car does not see the bridge and how crowded it may be, only an always-open (also when full) gate to it.

In a small system, memory for the pool is often limited. One would not initiate the buffer pool with 1000 elements when one thinks that only 13 are needed. One would set it with some margin, to f.ex. 15. But should the 16th message arrive, it will collapse by design. This practice should be approved by agencies if one can **prove** or **verify** that the 16th message indeed never arrives. Arguing should not be enough.

At this point we do not discuss the low "Office Mapping Factor" that often comes with send-and-forget. A programmer must, at application level, often know more about the other processes' internal doings than what might be inferred from the protocol between them. This is much discussed in an earlier paper [14] as well as in a myriad of papers for these WoTUG/CPA conferences.

3.4 Sending on a Synchronized Unbuffered Channel

This is the basic mechanism in CSP-based systems. The first process that is trying to perform an operation on the channel will be descheduled, and only rescheduled when the second process arrives and the communication is completed and data moved. The sender or receiver may be first. Observe that a channel here most often is unidirectional and point-to-point. Data is copied from the sender's context directly to the receiver's context. There is no intermediate buffer. In order to handle buffering and overflow, extra processes to manage this have to be added.

3.5 Selective Choice

Classical occam does not allow channel output guards in the **ALT** statement, only input guards, timeouts and **SKIP** (a guard that is always ready, like an **else**). Input guards

become ready when there is a sender at the other end. Output guards (which CSP allows) become ready when there is a receiver at the other end. An **ALT** will block if none of its guards are ready.

We have mentioned that there are similarities between sending on an **XCHAN** / waiting for **x**-channel and sending on a channel in an **ALT**. This section will try to outline some basic differences.

The **XCHAN** will differ from an output guard (we call it **ALTOUTPUT**) in several aspects. We have not differentiated between systems that may mix input and output guards in an **ALT**, and those that in addition have *pre-conditions*[5] on guards.

1. The **XCHAN** asynchronous send will never block (but it may discard data). The **ALTOUTPUT** will block (with no data loss) if channel cannot take it and no other guard is ready.
2. The application will always be informed if an **XCHAN** would have blocked, per channel. The application will be explicitly informed about a blocked **ALTOUTPUT** only if the **ALT** contains an **else** (**SKIP**) as the only other component. If more components are in the **ALT**, then the application will only know that *all* of the components are blocked, *including* the channel in mind. When one is taken, we will not know if any other component *could have been taken*.
3. Since **XCHAN** offers this explicit a priori "overflow" information and **ALTOUTPUT** offers this non-explicitly (as part of the **ALT** set of guards), the next sending attempt of an **ALTOUTPUT** has to be part of an **ALT**, while no next *attempt* is needed *for* the **XCHAN**. The sender passively waits in an **ALT** for the **x**-channel and can service other events (e.g. the arrival of fresh data).
4. When sending on the **XCHAN** is not taken, the sender side commits to send when **x**-channel arrives, *but it does not commit to what to send*. This commitment may live through several **ALT** setups[6]. The **ALTOUTPUT** is "new" each time the **ALT** is entered.
5. This **XCHAN** commitment survives the attempted sending and is seen by the receiver when it arrives. This commitment is a state of the channel (in the same way as the presence of a process wanting to use the channel is a state). The **ALTOUTPUT** has no such "colouring" .

The above suggests that the **ALTOUTPUT** at worst offers a posteriori ("second order") indication that a particular channel is not able to communicate. The **XCHAN** offers a priori ("first order") indication.

However, this paper is not much concerned about **XCHAN** versus **ALTOUTPUT**, since **XCHAN** allows *similar* functionality to **ALTOUTPUT** *without* having to support both input and output guards. Secondly, we have not looked at the link level protocols needed to implement **XCHAN** and **ALTOUTPUT** safely and efficiently, in shared memory or distributed memory cores. Thirdly, we have not made any formal models for the two to try to see if the differences discussed above would, after all, collapse. After the reasoning we have done here, we would be surprised should this be found. Fourth, we have not discussed failure modes and recovery. However, all these factors could be interesting for further investigation.

---

[5] The *pre-condition* is a run-time evaluated expression (yielding a boolean) that may be attached to a guard in an **ALT**. If the pre-condition is false, the guard is ignored in that execution of the **ALT**. This enables run-time control over which events are considered.

[6] As mentioned in the Introduction, this is the same logic as seen with the EGGTIMER and REPTIMER primitives as described in [9]

Observe that Go allows both input and output guards in **select**, but does not have *pre-conditions*. The Appendix shows how Go may use its channels with a **select** statement mixing input and output to facilitate some of the functionality of **XCHAN**. This is also discussed there.

4. Link Level and Application Level

Application processes communicate according to protocols designed by the system designer / programmer at the *application level*.

The method that the operating or run-time system uses to transfer data between processes represents the *link level*. If only asynchronous send-and-forget is available, then synchronisation is done by adding flow control return messages. If only synchronous channels are available, then buffering is added (as we have seen) by chaining processes, or internally in a process by, for example, a ring buffer. Or the channel itself may be buffered.

"Architectural leak" from link to application level could be seen as application code that is added to compensate for missing features at link level. Chained processes and overflow buffers are needed when buffered channels are not supplied. Busy polling is needed if the link level does not deliver appropriate flow control. The *channel-ready-channel* connected to a buffered channel, described in this paper as **x**-channel, would decrease architectural leakage.

A standard occam **CHAN** does know whether the receiving end is blocked or ready. If the sender sees no receiver "first" on the channel, then the traditional solution is to deschedule (i.e., block) and itself become first. The channel contains this information, but this is not handed over to the application. **XCHAN**, in a sense, moves this information up into the application in much the same way as an output guard.

Some words need to be said about the source of the **x**-channel being from the run-time system. We have compared this to the source of a timeout channel, which often is also from the run-time system - even if may be a system-timer process proper. The system clock constitutes a single source for all the timeout messages. The **x**-channel is more complex, as each **XCHAN** would have one each. If we implemented this as a pattern, then we would have no means to "send on a channel from a channel", corresponding to sending on **x**-channel "from" **XCHAN**. Due to layering and the fact that we suggest **XCHAN** as a primitive type, the source then cannot and should not be from application level.

5. Examples

Correct use of a buffered channel with **x**-channel may be considered a programming pattern. We will call this the "**XCHAN** pattern". If the link level part of it is handled by the run-time scheduler, then the pattern would cover the application part only (perhaps verified by the compiler).

5.1 A Channel-Ready-Channel in Traditional occam

An "intelligent buffer" could be a composite process that takes input on one channel and sends out on another. It could potentially receive more than it can send, filling up an internal buffer that eventually could overflow. Since we do not have output guards, it must be prompted for output by an additional process. This is a classical occam idiom.

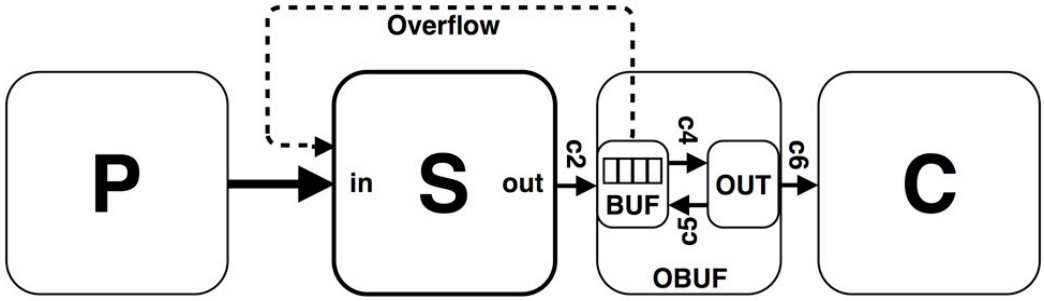

Figure 1. Example of an overflow buffer (OBUF)

The figure is rewritten from an earlier paper [15]. The composite overflow buffer OBUF consists of BUF and OUT. The channel-ready-channel between OUT and BUF is c5, and it must be **ALT**ed with c2 in BUF. This pattern has been used extensively in occam. BUF knows when sending on c4 that it will not block. What we see here is in fact a buffered channel from c2 to c6 with minimum capacity 2. A ring buffer could be added in BUF, and overflow handled by throwing away messages and inserting an overflow message in the buffer, so that it may be seen by the receiving end.

5.2 Local ChanSched ANSI C with Channel-Ready-Channel

This section shows how buffered channels with bundled channel-ready-channel could be used, provided the concept was added to the "ChanSched" system described in an earlier paper [9][7].

Autronica's ChanSched is written in ANSI C. The mentioned paper shows how rescheduling points are inserted by a tool that builds an invisible jump table. In Listing 1 there is only one synchronisation point: line 05. Lines starting with three dots are "token based" folds containing code, not shown here.

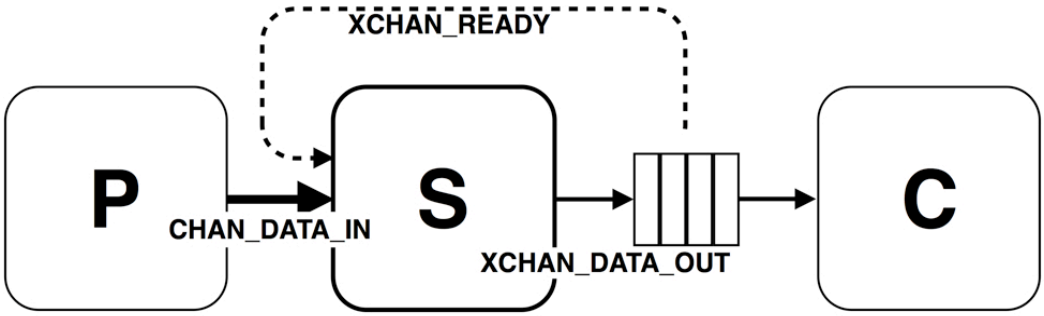

Figure 2. Buffered **XCHAN**, as shown in Listing 1 (below)

Listing 1 (below) shows the main structural part of process S in Figure 2. Listing 1 shows ANSI C code with upper case macros, where *italic* macros like ***ALT()*** are channel handling macros. There is one buffered channel, **XCHAN_DATA_OUT**, and there is one channel-ready-channel, **XCHAN_READY**. The latter is provided by the run-time scheduler, like the structures needed to support timer channels. Therefore it is drawn as coming from the buffered channel itself. This maps quite well with how these channels would be perceived.

[7] The style in this document is as if it would still exist.

Most of the time the process will hang on the **ALT** (line 05). When one of the **ALT** components has been taken, **ThisChannelId** is delivered to the 2-**case switch** statement.

The basic idea here is that this process would be used as some kind of server or driver, which may receive messages faster than it can get rid of them, but when there is an overflow situation it may still receive on its **XCHAN_DATA_IN** input, and not have to revert to busy polling to catch up when the output channel would again become ready for input.

Observe line numbers 09 and 16, where **XCHAN_OUT** is used, sending on a buffered channel. These calls can never block, meaning that the next lines in all situations are returned to. If there was no room for the message, **CP->Sent_Out** will become **FALSE**. When the process again enters the **ALT**, new messages may arrive any time and be treated in the overflow code in line 13. When the output channel is ready again, line 09 is entered, and it again sends over the buffered channel **XCHAN_DATA_OUT**.

```
01 while (TRUE) {
02   ALT();
03     ALT_SIGNAL_CHAN_IN (XCHAN_READY);  // data-less
04     ALT_CHAN_IN (CHAN_DATA_IN, Value);
05?  ALT_END(); // Delivers ThisChannelId:
06
07   switch (ThisChannelId) {
08     case XCHAN_READY: {                 // sending will succeed
09!      CP->Sent_Out = CHAN_OUT (XCHAN_DATA_OUT,Value);
10     } break;
11     case CHAN_DATA_IN: {
12       if (!CP->Sent_Out) {
13         ...  handle overflow (decide what value(s) to discard)
14       }
15       else {                            // sending may succeed:
16!        CP->Sent_Out = CHAN_OUT (XCHAN_DATA_OUT,Value);
17       }
18     } break;
19     _DEFAULT_EXIT_VAL (ThisChannelId)
20   }
21 }
```

Listing 1. Overflow handling and output to buffered channels (ANSI C and macros)

6. Zero Buffering

With no buffer in the channel, the buffer needs to be moved inside the process S. When overflow occurs, the process now is able to also manipulate the buffer completely - remove the oldest, newest or perhaps flush all in the buffer. This is of course more versatile than a FIFO-buffered channel.

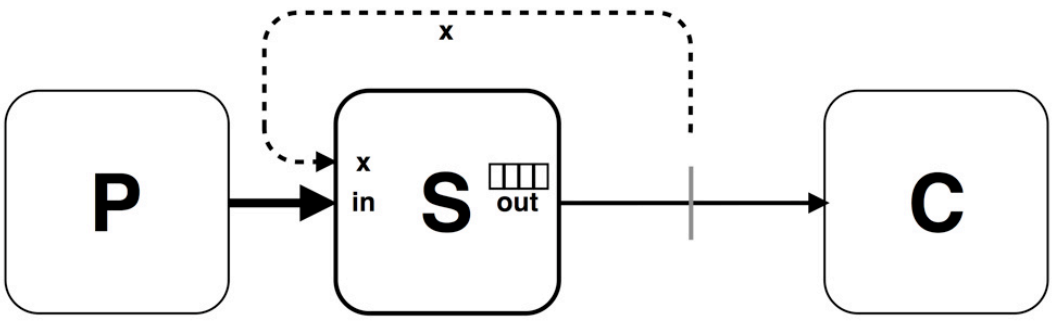

Figure 3. Zero buffered **XCHAN**

According to the **XCHAN** pattern, if S has tried to send on **out** and failed, when **x**-channel arrives, something has to be sent – either the last value or an overflow message. This is because at this time the receiver is not rescheduled before S has done this second sending. Even if it was receiving inside an **ALT**, it is blocked like it would have been sending. This property also goes for a buffered **XCHAN**.

This zero-buffered **XCHAN** would add a semantic asymmetry to sending and receiving on a synchronous channel. An asymmetry is already present for an **ALT**, since when the **ALT** is taken by one of its components all the other components have their "first"-attribute removed. At the **XCHAN** sender side a "first"-colouring is not removed since this is the commitment that we have already mentioned.

7. XCHAN breaks Deadlocking Cycle

Since an **XCHAN** does not block if the **XCHAN** pattern is used correctly, the **XCHAN** has the property that it may break a deadlocking cycle. We argue that this also is a quality enhancement feature.

As an example we describe an **XCHAN** and a **CHAN** connected in each direction between two processes. We have documented a deadlock free pattern in [15], later called "knock-come" and described and formally verified with Promela and Spin in the blog note [16].

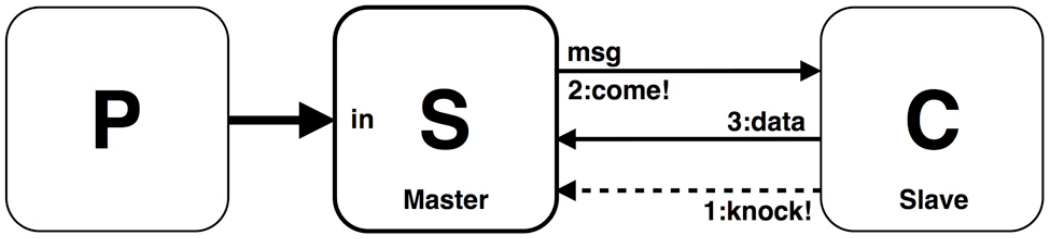

Figure 4. Traditional "knock-come" pattern

In Figure 4, we see the traditional "knock-come" pattern that we have used extensively. S and C may both send spontaneously to each other. This is a change from the other examples here, where C is not sending to S. S sends "msg" without permission since it is the Master. But the C Slave has to "knock" first with a non-blocking asynchronous signal on a signal channel. When S decides, it answers with "come" and waits atomically for "data" from C. When C receives "come", it has to immediately send "data". This pattern is deadlock free.

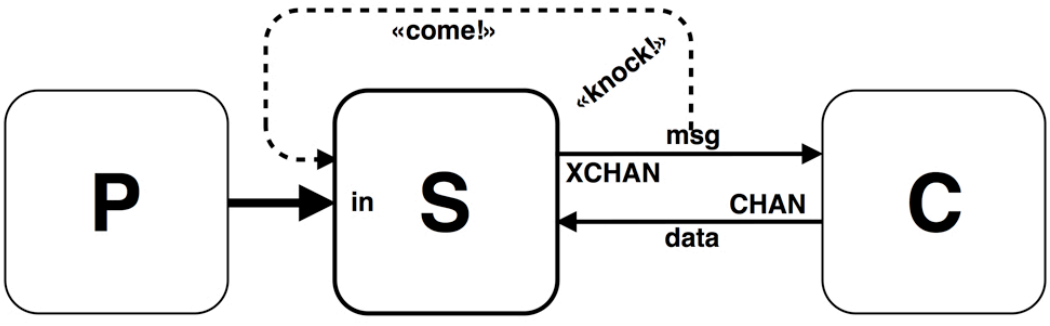

Figure 5. Same pattern with **XCHAN**

With **XCHAN**, we see that we could remove the "knock" asynchronous data-less channel and instead connect an **XCHAN** from S to C. C can send "data" any time it needs, and since S never deadlocks on **XCHAN** output, it will always be able to receive. If S should send on **XCHAN** at the same time, it will not block but go on to pick up any colliding "data". If "msg" fails, S will wait in an **ALT** for new data on in and the **x**-channel ("come"). If during this waiting a new message arrives from P, then S could decide to instead send this.

We have removed the "Master" and "Slave" role names when the **XCHAN** is used, as we have failed to find good names for these roles. The reason is that the run-time system (or the "channel itself") now has an additional third role.

8. Syntax Suggestion

Here is a rather speculative suggestion of how some languages may have the new channel primitive declared. No surprise now, we have simply prefixed '**x**' to the already existing **CHAN/chan** primitives. Promela [17] and XC [18] are also shown.

(occam: also added optional capacity)
```
XCHAN (100) OF BYTE my_xchan:
```

(XC: also added optional capacity)
```
[100] xchan my_xchan;
```

(Promela: has **select** with outputs)
```
xchan my_xchan = [100] of byte
```

(Go : has **select** with outputs)
```
my_xchan := make (xchan byte, 100)
```

Input or **ALT**-ing on the channel-ready-channel could be done by some using standard type of attribute to the channel, like "**x.my_xchan**".

A compiler would have to know the **XCHAN** software pattern and verify that the implementation is correct. Describing the usage rules for this is beyond the scope of this paper.

9. Some More XCHAN Semantics

The operations possible on both **CHAN** and **XCHAN** would be to send or read from them. However, the **XCHAN** has one additional operation: to wait for it (i.e., its **x-channel**) after failed sending.

Extending the operation set of the **XCHAN** could be of use, especially on multiprocessor systems where dynamically created channels could be closed. So, with **XCHAN** in place, we could perhaps include some more operations on the buffered channel, like close, flush, return with percentage full, signal not-full buffer when one read (space for one) or all read (empty).

However, we would be reluctant to add much more than necessary to the **XCHAN**. This would perhaps also be closer to the spirit of the original CSP design. A discussion of this would be outside the scope of this paper. We have also shown that a zero-buffered **XCHAN** would need the buffer to move inside a process for full control by the application.

As mentioned, note that there is a requirement to write to the channel if the **x**-channel is signaled[8]. This is part of the **XCHAN** pattern. It is outlined in Listing 1. If this is not done, the commitment colouring of the **XCHAN** is not ended, potentially leading to deadlock if the receiver blocks indefinitively.

Observe that it may not be possible to model a "full **XCHAN**" with standard occam code. We have seen that the buffered case may be built with a composite process, but we would need to also mimic a *fire-and-forget* channel for the **x**-channel (which is possible with yet another buffer process). However, the unbuffered **XCHAN** looks difficult for occam code, as any intermediate process tends to introduce buffering, and a standard zero-buffered occam channel introduces blocking (and an **XCHAN** needs asynchronous sends).

The **XCHAN** buffers could be statically allocated in systems where dynamic memory handling is not allowed, and the data is carried over protocols known at compile-time. This could be the case for safety-critical systems.

Observe that a classical[9] CSP channel has no associated *scheduling* queue; so reasoning about the scheduling and timing properties is easier than if the sending processes had to be queued. (Any buffer could of course be considered a *data* queue.) This timing property concern may be exemplified with the Ada Ravenscar profile for high integrity systems, which is "a subset of the Ada tasking features designed for safety-critical hard real-time computing" [19]. In the Ada rendezvous, it is not easy to analyse which processes are queued, so the Ravenscar profile prohibits use of rendezvous – since access to them is based on scheduling queues. To try to rectify, the Ada Ravenscar profile has been extended with a CSP library that provides channels the Ravenscar legal "protected objects" [20]. The paper argues that:

> *"The advantage of these Ravenscar channels is transforming the data-oriented asynchronous tasking model of Ravenscar into the cleaner message-passing synchronous model of CSP. Thus, formal proofs and techniques for model-checking CSP specifications can be applied to Ravenscar programs. In turn, this increases confidence in these programs and their reliability."* [10]

Also observe that a Go channel has a queue of senders and a queue of receivers. Further information is given in the Appendix.

10. Discussion

The asynchronous sends on **XCHAN**s break the blocking semantics of CSP. Should such a send fail, we have introduced the always present *channel-ready-channel* (**x-channel**) to compensate.

Is this **XCHAN** too easy to program at the application level, so that there is no need to bother implementing and offering in a programming interface or turn into a proper *citizen* of a language? We do not think so.

Remember, there is a history here of the asynchronous and (read *versus*) synchronous camps. The *"send-and-forget"* metaphor is transformed into a *"send-and-forget until ready if not sent, then send what you have"* metaphor. Since there is no such thing as infinite

[8] We think it is possible to implement *not* having to send after the **x**-channel signal, but this feels much more complex, as the run-time system would probably need to do agressive state cleanup and rescheduling and even require some application coding in the receiver. The latter would lead to architecture leakage.
[9] As opposed to Ada rendezvous and Go channels (see Appendix)
[10] This CSP library is rather limited, as it does not contain any **ALT** implementation.

buffering, and finding "enough buffer size" is a rather risky business [21], we have argued that having an **XCHAN** would enhance quality.

We do not see that different process priorities between the ends of an **XCHAN** should add complexity as compared to standard CSP channels.

The **XCHAN** discussed in this paper does not introduce the same *"asynchronous tasking model"* as in the Ravenscar Ada, where Ada rendezvous are gone. It merely adds **a**synchronous messaging as a first class citizen of CSP based systems.

11. Conclusion

We have tried to argue that an **XCHAN**-type CSP based channel described here may become useful in both the asynchronous and the synchronous communication camps. We have also tried to show that **XCHAN** could enhance the quality of many concurrent systems, by adding a new means to handle application overflow control as well as a new channel type that will assist in breaking deadlock cycles.

12. Acknowledgements

Thanks to Sverre Hendseth of NTNU (Norwegian University of Science and Technology, Trondheim) for reading and commenting on an early version of this paper. Also, thanks to colleagues Ommund Øgård and Per Johan Vannebo for their comments, and Rune T. Aune for valuable **select** and Go discussions (all at Autronica Fire and Security (AFS), Trondheim). Also, thanks to the helpful Go community (see Appendix). Especial thanks to the four anonymous CPA readers for their valuable feedback, some comments in form harder to digest than others. Finally, thanks to the editor Peter Welch.

References

[1] Google Go programming language by Robert Griesemer, Rob Pike, and Ken Thompson. See "The Go Programming Language Specification" at http://golang.org/ref/spec#Channel_types

[2] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall International, 1985. Read at http://www.usingcsp.com/cspbook.pdf

[3] D. May and R. Pountain. A Tutorial Introduction to occam Programming, BSP Professional Books (ISBN 0-632-01847-X); 1987.

[4] INMOS Limited. occam 2 Programming Manual, Prentice-Hall International, ISBN 0-13-629312, 1988. See http://www.wotug.org/documents.shtml

[5] Occam programming language. See http://en.wikipedia.org/wiki/Occam_programming_language

[6] P. H. Welch and F.R.M. Barnes. Communicating Mobile Processes: introducing occam-π. In '25 Years of CSP', A. Abdallah, C. Jones, and J. Sanders, Eds., Lecture Notes in Computer Science, vol. 3525. Springer Verlag, 175–210 (2005).

[7] P.H. Welch and J.B. Pedersen. Santa Claus: Formal analysis of a process-oriented solution. ACM Trans. Program. Lang. Syst. 32, 4, Article 14 (April 2010), 37 pages. DOI=10.1145/1734206.1734211 http:/doi.acm.org/10.1145/1734206.1734211

[8] S. Stepney, P.H. Welch, F. Polack, J. Timmis, F.R.M. Barnes and A. Tyrrell. CoSMoS home page. See http://www.cosmos-research.org/cosmos/. EPSRC grantsEP/E053505/1 and EP/E049419/1 (2007-12).

[9] Øyvind Teig, Per Johan Vannebo. New **ALT** for Application Timers and Synchronisation Point Scheduling. In *Communicating Process Architectures 2009* (WoTUG-32), Peter Welch, Herman W. Roebbers, Jan F. Broenink, Frederick R.M. Barnes, Carl G. Ritson, Adam T. Sampson, Gardiner S. Stiles and Brian Vinter (Eds.). IOS Press, 2009 (135-144), ISBN 978-1-60750-065-0. Read at http://www.teigfam.net/oyvind/pub/CPA2009/paper.pdf

[10] Steve Schneider, Concurrent and Real-time Systems. The CSP Approach. ISBN: 0 471 62373 3 John Wiley & Sons, Ltd, England, 2000.

[11] Linux. Pipe: http://linux.die.net/man/7/pipe, writing to a pipe: http://linux.die.net/man/2/write, and doing the **`select`**: http://linux.die.net/man/2/**`select`**

[12] Richard Beton. libcsp – a Binding Mechanism for CSP Communication and Synchronisation. In *Multithreaded C Programs. In P.H. Welch and A.W.P. Bakkers*, editors, *Communicating Process Architectures, Proceedings of WuTUG 23* volume 58 of *Concurrent Systems Engineering*, pages 239-250, Amsterdam, The Netherlands, September 2000. WoTUG , IOS Press.

[13] Gerard J. Holzmann. Design and Validation of Computer Protocols. Prentice Hall, 1991, ISBN 0-13-539834-7 paperback. Chapter 8, at chapter "Communicating Finite State Machines". Read at http://www.spinroot.com/spin/Doc/Book91_PDF/F8.pdf

[14] Øyvind Teig. High Cohesion and Low Coupling: the Office Mapping Factor. In *Communicating Process Architectures 2007*, Alistair McEwan, Steve Schneider, Wilson Ifill and Peter Welch (Eds.) IOS Press, 2007 (pages 313-322). ISBN 978-1-58603-767-3. Read at http://www.teigfam.net/oyvind/pub/CPA2007/paper.pdf

[15] Øyvind Teig. No Blocking on Yesterday's Embedded CSP Implementation. In *Communicating Process Architectures 2006*. Peter Welch, Jon Kerridge, and Fred Barnes (Eds.) IOS Press, 2006, pages 331-338 ISBN 1-58603-671-8. Read at http://www.teigfam.net/oyvind/pub/CPA2006/paper.pdf

[16] Øyvind Teig. The "knock-come" deadlock free pattern. Blog note. Read at http://oyvteig.blogspot.no/2009/03/009-knock-come-deadlock-free-pattern.html.

[17] Promela is a "process meta language", analysed with Spin, see http://en.wikipedia.org/wiki/Promela

[18] XMOS Limited, XC programming language. See http://en.wikipedia.org/wiki/XC_Programming_Language

[19] The Ada Ravenscar profile for high integrity systems. See http://en.wikipedia.org/wiki/Ravenscar_profile

[20] Diyaa-Addein Atiya and Steve King. Extending Ravenscar with CSP Channels. Department of Computer Science, University of York, UK. Read at http://www-users.cs.york.ac.uk/~king/papers/ExtendingRavenscar-AdaEur05.pdf

[21] Alex Brodsky, Jan Baekgaard Pedersen and Alan Wagner. On the Complexity of Buffer Allocation in Message Passing Systems. In *Communicating Process Architectures 2002*. James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.) IOS Press, 2002, pages 79-96. Read at http://wotug.org/paperdb/send_file.php?num=10

[22] Øyvind Teig . "Using select with input and output statements in driver between fast producer and slow consumer". A golang-nuts post. Read at https://groups.google.com/forum/?fromgroups#!topic/golang-nuts/qa2p0PRY0WE

Appendix: Mixing Input and Output in the Go `select` Statement

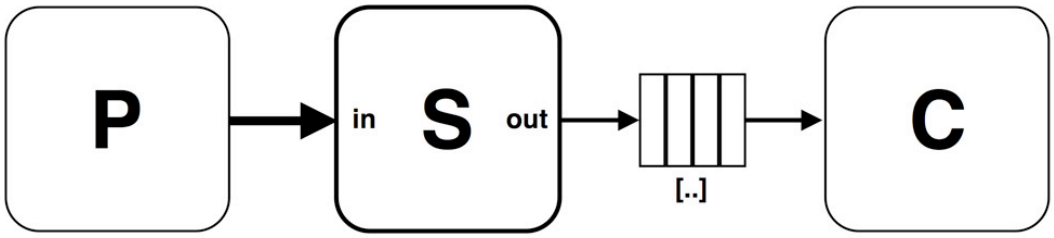

Figure 6. Go example (right channel capacity irrelevant)

This Go example uses blocking `select` (with no `default` case) with one output and one input. Go "simulates" a guard if a communication component is `nil`, so when not valid (line 08-09) only the input line 15 would ever execute. Line 12 will always listen (?) on the channel, while line 15 then may send (if not `nil`). In the receive statement of lines 12 we have dropped an optional second parameter, so we assume the channel does not become closed.

```
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 }
```

Listing 2. Managing without `xchan` in Go with goroutines

Thanks to the golang-nuts user group for this code [22], as well as for the following code.

Another way to do some of this is to take the buffering in the channel at face value:

```
01 func (ch Leaky) Send(value int) {
02     for {
03         select {
04             case ch <- value:
05                 return // sent!
06             case <- ch: // No sending above, read(!) an element..
07                 // .. and discard (no code here)
08         }
09     }
10 }
11
12 func (ch Leaky) Receive(value int) {
12    return <- ch
13 }
```

Listing 3. Managing without **xchan** in Go, implemented with a type

This code implements a leaky channel. The code will lose a value when the channel blocks. Line 06 in fact listens on the output end of the channel. This channel is now full and line 06 picks out the oldest element. This code does not listen for new input while the buffer is full, but it could be coded.

Observe that each Go channel has a queue of receivers and senders. Therefore Go in the future might need a safety critical profile subset. How this would look (except perhaps take "advantage" of the fact that there are no parallel usage rules in Go), is beyond the scope of this appendix.

Are the reasons we have given for **xchan** interesting if this particular problem may be solved with a "full" **select** with mixed input and output statements? Will an **xchan** in any way influence how Go programmers think? To try to answer this is also beyond scope for this appendix. We certainly think this is interesting.