Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

No Blocking on Yesterday's Embedded CSP Implementation
(The Rubber Band of Getting it Right and Simple)

Øyvind TEIG
Autronica Fire and Security (A UTC Fire and Security company)
Trondheim, Norway
http://home.no.net/oyvteig

Abstract. This article is a follow-up after the paper *"From message queue to ready queue"*, presented at the ERCIM Workshop last year. A (mostly) synchronous layer had been implemented on top of an existing asynchronous run-time system. After that workshop, we discovered that the initial implementation contained two errors: both concerning malignant process rescheduling associated with timers and "reuse" of the input side of a channel. Also, the set of process/dataflow patterns was not sufficient. To keep complexity low, we have made two new patterns to reflect better the semantic needs inherent in the application. Our assumption of correctness is also, this time, based both on heuristics and "white-board reasoning". However, both the previous and this paper have been produced before any first shipment of the product, and well within full-scale testing. Our solutions and way of attacking the problems have been in an industrial tradition.

Keywords. Case study, embedded, channel, run-time system.

Introduction

The purpose of this paper is to inspire designers of embedded systems. Our target platform is a microcontroller with 64 KB volatile and 256 KB non-volatile memory, and we are programming in ANSI C. We have a specified behaviour even more complex than that handled (just) by the previous industrial architecture. We have a stated need not to design against possible overflow of message buffers. We have a need to make the process state machines handle only their own state.

So, we did … and we learned. Now, we hope that displaying and catering for the problems we encountered will increase confidence in our solution. As the final stages of the development and testing have been reached, we see that the *"CSP way of thinking"* has enabled us to implement a complex multithreaded system with easily described behaviour. System and stress testing (our heuristics) have indeed been encouraging for our scheduled release of the product, in which this (multithreaded) unit is a crucial part.

The sub-title, and real contents, of our earlier ERCIM Workshop paper[1] was: "Case study of a small, dependable synchronous blocking channels API". Here is its abstract (modulo a couple of linguistic fixes):

> *This case study shows CSP style synchronous interprocess communication on top of a run-time system supporting asynchronous messaging, in an embedded system. Unidirectional, blocking channels are supplied. Benefits are no run-time system message buffer overflow and "access control" from inside a network of client processes in need of service. A pattern to avoid deadlocks is provided with an*

added asynchronous data-free channel. Even if still present here, the message buffer is obsoleted, and only a ready queue could be asked for. An architecture built this way may be formally verified with the CSP process algebra.

A second sub-title was "Ship & forget rather than send & forget", suggesting that a system built on this technology should be easier to get right the first time.

This paper shows that for this to happen in the product, the process schedulings have to be 100% correct. The bottom layer of the channel run-time system must indeed be correct. Even if little is repeated from [1], we will sum up its basic points:

- The bottom layer consisted of a non-preemptive SDL[1] run-time system. Here, smaller messages were sent in equal N-sized chunks (N is the maximum size of each buffer position) – as many as the sender needed to send within the same scheduling. Sending was non-blocking and it could potentially cause malignant buffer overflow. This was the first objection to that architecture. The second was that, whatever its state, a receiver was not able to "protect" itself from messages. Timers were handled with a linked list of future times and any timed-out timer would enter the message queue proper. A "process" could send messages to any other process, including itself. A "driver" could only send messages to processes, and receive no messages. The Scheduler scheduled processes until the message queue was empty, and then always (for "priority") took all drivers in turn before looping around for (lower "priority") messages from processes.

- On top of this we built the (CSP/occam-like) synchronous channel layer, with channel **`OUT`**, **`IN`** and **`ALT`**, and with timeouts on the two latter. An **a**synchronous sender-side data-free channel type was also introduced, needed to start a message sequence so that the initiator would in due time be allowed to send a rich message in the same direction as the asynchronous signal. It could be part of an **`ALT`** on the receiver side. This would disallow deadlocks when two processes wanted to send messages to each other spontaneously. This way, in the light of the asynchronous signal channel, we thought we avoided overflow buffer processes for this architecture. This paper shows where we did not.

- With synchronous sending of data, the "**`CHAN_CSP`**" layer took care of the intrinsic "**`memcpy`**" when both parties were ready. *Data* messages were not any more sent in the message queue, as were *ready* or *scheduling* "messages". The queue could now not overflow. For any one process only one ready message at a time would be in the queue. Any timer event could be in the queue if it arrived before the channel. If not it should be 100% filtered to avoid a malignant scheduling. This paper is also about what happened when this filtering was not working correctly.

Our main source of inspiration was SPoC [2], where both source code and C coding style had been related to by previous usage of occam and SPoC. However, even with occam experience on the transputer, we had no knowledge of its architecture at this level. occam has lower layers hidden for us in both these cases. Because of requirement of time to target, in the industrial tradition, and since we were to build on top of an existing asynchronous system (which neither SPoC nor the transputer did), we figured that some new implementation thinking was needed. Looking back at that assumption, we now see that it would have been easier to get it right sooner using earlier known solutions.

[1] An *Autronica Fire and Security* developed "SDL type" run-time system, written in C. SDL stands for *Specification and Description Language* and is ITU-T Recommendation Z.100.

1. Run-time System Malignant Schedulings Removed

1.1 Enough is Zero Extra Timeouts

In our system we have two types of timers:

- The high-level timers are function calls that a process makes in order to initialise, increment and check for timeouts, or to stop – all based on a low-level single 32 bit 1 ms system tick. Detecting that half an hour has passed is then up to the process by exercising the function calls directly. All functions on these timers are "atomic" since they operate on a read-only copy of the global clock taken immediately after rescheduling in each process.
- Additionally, time outs on channels (always as part of an **`ALT`**) or when no channel is involved (delay) are implemented. This is what we thought worked, but actually it did not.

We had been aware of the underlying problem for a long time: that in the asynchronous run-time system on which we built the CSP layer, it was not 100% assured that a timer, once subscribed to, would not arrive too late – even if it had been explicitly (or implicitly) cancelled. When the timer event had come as far as into the message queue (and was not just a node in the sorted list of timeouts), it was too late. The solution in that system was to "colour" timers in a certain way to detect later that that timer event could be discarded. This was all done at application level by the process. Then, the process in a way becomes its own scheduler.

However, a CSP process is not allowed to be shaken by unanticipated scheduling. We could have made an exception and done the same colouring as mentioned. After all, it was not a channel that would have caused the scheduling, so no other process would have been involved. Still, we went for the clean solution and thought this would be easier in the end.

In order to do this, we did an invisible (to the process) colouring of the timer and invisible discard of it. But this was the final result. Let us go back some steps.

Prior to this (when [1] was written), we had implemented the filtering of unwanted timer events by **`#ifdef`**'ing some code in the run-time system, based on the state of the process' **`ALT`**. (With all C **`#ifdef`**s, the original run-time system is unaltered, albeit not uncluttered – just like all other operating systems we have studied.)

The **`ALT`** state takes one of the three values:

```
typedef enum {
    CHAN_ALT_OFF_A              = 0,
    CHAN_ALT_ENABLED_ON_A       = 1,
    CHAN_ALT_ENABLED_INACTIVE_A = 2
} StateALT_a;
```

If the **`ALT`** state was "inactive" and there was no reason to take action, we threw the timer event away by not letting it into the message queue. So far so good. This worked in a product where *all* channel input was part of an **`ALT`**. But with new users, *individual* inputs and delays were used and then, after some time, we saw processes crash from unwanted timer events. The problem was that the **`ALT`** state naturally follows the life of a process, the "inactive" state may be a later phase's "inactive"; similarly for the "not inactive" state. *So, we threw away too few.* In order to fix this, we had inserted a filter in the **`Scheduler`** itself, based on a boolean in the process' context. Thinking it over now, this filter had absolutely no effect. But it had seemed that all unwanted timer events were gone. We would have to do it 100% safe next round.

We saw that for a timeout event to be "taken" by a process, one precondition was that the process should not ever have been scheduled in the meantime. So, we introduced a mandatory second value in the process context (after the mandatory **`State`**, both reachable from **`Scheduler`**). We called this **`ProcSchedCnt`** and let the scheduler increment it every rescheduling. Now, whenever a timer has timed out and is first in the timer list, we check that the process **`ProcSchedCnt`** and the **`ProcSchedCnt`** present in the timer (copied when the timeout was asked for) are equal. Now we do this only at the place where a timer would enter the ready queue, not when actual scheduling takes place. The mentioned "absolutely no effect" filter was then also removed. We now test on **`ProcSchedCnt`** and **`StateALT`** and know that no other scheduling event would be present in the ready queue if **`ProcSchedCnt`** and **`StateALT`** were used to filter. So, a timer scheduling event that reaches the **`Scheduler`**, is now indeed a true timeout.

A boolean instead of **`ProcSchedCnt`** cannot be used here. The process could go on and ask for a new timer, and do this again and again. A boolean state cannot distinguish these states from the state in which the process was, when it asked for the initial timer. A "linear" tagging must be done.

The only problem we see with this is the word width of **`ProcSchedCnt`**. We have chosen 16 bits. It is very unlikely that a process should have been scheduled some 65 thousand times before the timer event fires. An 8 bit value probably would work fine, too. An effect that helps here is a built-in feature of the underlying SDL system to cancel any earlier timers once we order a new. Even if it were not able to, as mentioned, cancel it once it had entered the message queue.

1.2 A Channel must be Empty from Both Sides Before a Refilling is Attempted

This problem involved two processes. To analyze the error scenario was the hardest. Once we understood the problem, the solution worked for all cases – but with a small run-time penalty.

The error had not been seen in an earlier use of the **`CHAN_CSP`** layer. This time, new users had reused an input channel differently than in the previous product.

A process may be "first" on a channel either when it is blocking for input on it (alone or as part of an **`ALT`**) or blocking for an output. The output blocking is final; the process would only ever be scheduled again when a true output has been done. We have no timeout on output. However, the input "first" process on a channel may be cancelled if it is in the set of an **`ALT`** that was not taken. Almost worse, it may appear again if a new **`ALT`** or input is done after this.

The error appeared only during certain scheduling sequences. Sarcastically, we would say it behaved like an asynchronous non-blocking system: the sequence of schedulings will happen a year after shipment, and then the service telephone would ring more and more often. Of course, the designers of such a system would have "seen" this event in their state diagrams, and so it would have been taken account of, and after the year all would still be fine. No service trip. After all, asynchronous designs will also work, if designed and implemented correctly.

Luckily for us, the misbehaving sequence(s) appeared during tests.

Initially, when a channel had been taken by the second contender, and the process was scheduled, the channel's "first" info was cleared to "none". After this any next "first" would not cause any extra scheduling because it would truly be first. This seemed until the tests, to be working. But then we saw that some "first" information was not cleared, causing pathological reschedulings. The new "first" was seen as "last" since there seemed to already be a "first" on the channel. When we started looking for this error, we saw that this

was fine for the channel in mind, but not for the other channels being part of an **ALT**. The people at Inmos must have seen this when they implemented the **ALT** on transputers in the 1980s.

We had known that the usual **ALT** implementation started with a set-up; and when the **ALT** had been taken, a tear-down. To our surprise we imagined that we did not need this. At least, not before we were forced[2]. Much code had been written, and we did not want to modify too much in user code. And we did not really want to shake the CSP layer by including set-up and tear-down functions. We were too far into the project.

Our solution requires a minimum of user code changes.

The **StateALT** variable is one per process, not one per channel. We decided to include a list of pointers in the **ALT** state structure: pointers to the maximum set of channels ever to participate in the **ALT** – pointers to their "first" variables. When a channel was taken on an **ALT**, we were now able to "null" not only the participating channel, but also the others. But we had to make sure that removing the first process on the channel could only be done if the first was in fact "this" process. If not, we would effectively stop other future communications. The change in user code was in the single module that initialises all the channels (they are global C objects, so this is possible). We also had to initialise the new pointers to the channel's "first" values.

The price we had to pay for this is the added run-time overhead of the looping to test and clear the "first" values. But this is acceptable in our application.

We think the CSP layer now has matured to a level where we feel quite at rest. The SDL layer had reached this level before we built CSP on top of it.

2. Architectural Patterns Extended

2.1 Two Times Zero Solved a Case better than One Times Zero

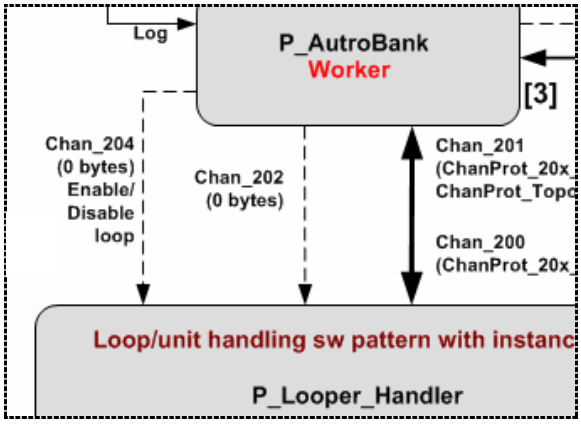

Figure 1. Pattern, here with two asynchronous channels (cut-out from our main diagram)

Now over to the application. The diagram in Figure 1 shows two processes communicating over data-free asynchronous channels (dotted lines with arrows) and one synchronous channel in each direction containing data (solid line with two arrows). The contract between the processes is deadlock free. The lower process may always send on the up **Chan_201** (and get or not get a reply immediately on the down **Chan_200**). If the upper process wishes to initiate a sending down it starts with a non-blocking signal on **Chan_202**

[2] With hindsight we should have also reused that part of SPoC [2]. In another world (where C/C++ would not rule), we could have used occam & SPoC plus C for the project.

or (see next paragraph) **Chan_204**. It then, sooner or later, receives a response on **Chan_201** saying "ok, come on" after which it immediately sends down on **Chan_200** what it had. While the upper is waiting for the reply, it is perfectly well able to serve an event or a directive/response pair. The diagram of this is in [1]. Further semantics are that the upper will not try to send down again before it has got rid of the present message. This has the added benefit that the addition of the asynchronous channel, like the synchronous scheme, will never cause message queue overflow.

Note that we now have *two* asynchronous channels down. One is for high priority and the other for low.

During long states, like taking up a fire detection loop, which may take minutes, the upper process may want to send data down. It does not know (and should not know) the state of the lower. So it gets a reply from down saying: "busy, try me again later". The lower process had its single (at the time) asynchronous channel switched off in the **ALT** guard in this state. But then we wanted to send a high priority escape signal saying to the lower: "stop what you are doing even if it is important". It was this need that had us introduce the "busy" reply. Now, the high priority asynchronous channel never has the lower process send a "busy" reply up, it will always pull down the escape type command we had in mind.

This could have been solved with an extended communication sequence between the two processes, even with a single asynchronous channel: the lower could have asked for the boolean information explicitly. But inserting the extra asynchronous channel was fine – and it gives less overhead than any extra communications would have done.

2.2 The Deadlock-Free Pattern that was Too Simple for a Certain Complexity

Much design is done in collaboration. The group members (some 5 of us) have known the application domain for years. Good and bad solutions, failures and successes were, so to say, implicitly on the table during discussions. We knew what to make, not least because we had been supplied with a specification. We also knew what not to repeat. And since we had decided not to use the UML based modeling and code generation tool that the host processor team used, we had decided for a process/dataflow model of the architecture. Textual descriptions and message diagrams were our main tools, in addition to informal discussions and white boards.

During the architectural discussions, we saw that data would flow as spontaneous and directive / response protocols – in both directions. Even in the complete system data would spontaneously flow both from the lowest level processes to the top, and the other way down. Process roles became important. We therefore needed the above described deadlock-free pattern between processes that needed both types of protocols.

Great was our surprise when we discovered that there was higher-level behaviour implicit in the system that rendered our deadlock free pattern so complex to use that we needed to devise something different.

In the final design we removed the asynchronous up **Chan_303** (Figure 2) with the synchronous up **Chan_301** (Figure 3) and introduced an overflow buffer (composite) process. The contract now is that the upper process never sends down more packets than the downward overflow buffer **P_OBuff** may hold – i.e. that the overflow buffer will never overflow. Therefore the upper process is always able to receive from down. Down will never block on a sending up, just like with the asynchronous channel case. So, the design is still deadlock-free.

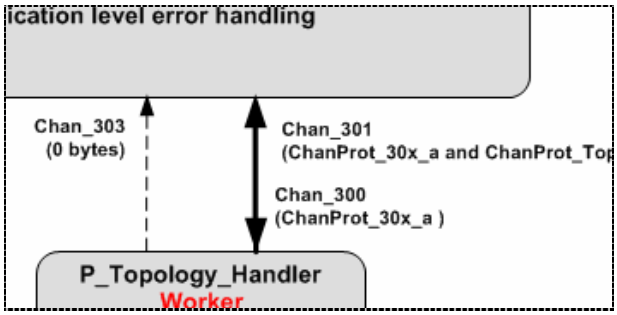

Figure 2. Standard pattern with one asynchronous channel (cut-out from our main diagram)

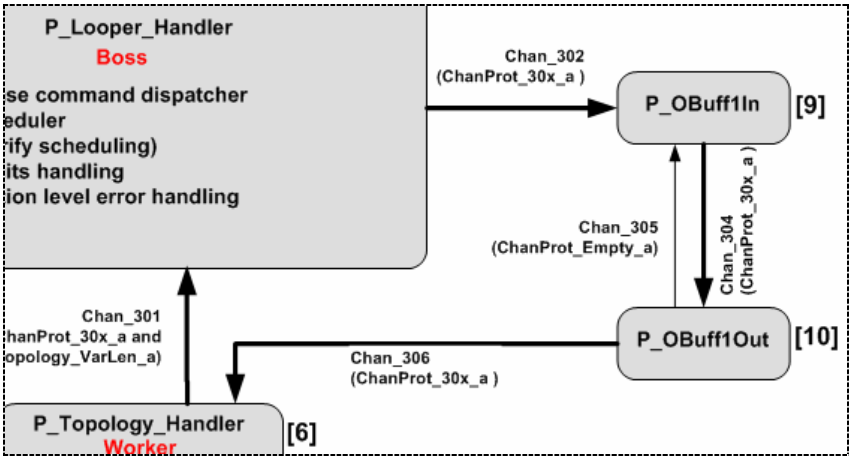

Figure 3. Overflow buffer pattern and no asynchronous channel (from our main diagram)

The implicit requirement is that the lower process handles both upcoming spontaneous (alarm type) events and sends directives down (to a process which handles the detector loop); and that it may wait for responses from detectors on the loop (going up). But the alarm type events are so important that they are queued in the lower process and need to be sent up instead of a response from a detector loop unit on an earlier directive. We try to avoid implementations where a process tends to be a scheduler of itself. When "up" gets a message from "down" that it has alarms, they are pulled up by a "give me more" message until the last message comes from "down" saying that this was the last. At the same time a directive down or a reply to "up" may be pending. The main problem was that the up-going event sequence might cross with a down-going event, and that we would need a complex solution to buffer down-going events in the lower process. Or, to end up with the simple solution that we now have: one single down-going event is stored away in the lower process, to be picked up when the up-going event queue becomes empty.

We essentially replaced the zero-information initial asynchronous signal to "up" with a data rich initial message from "down" to "up". We then avoided an extra sequence to establish the higher-level states of the two … and the need for the lower process to become a complex buffer handler and scheduler of itself … and the added complexity of the protocol.

The essential point here is that the initial architecture was traded for a seemingly more complex one to make the inner semantics and the protocols and message diagram simpler. Much anguish was experienced by the white board before we saw the picture. We felt a rising familiarity with a new set of tools: processes; synchronous vs. asynchronous channels; deadlock avoidance; internal vs. external complexity; seeing when the behaviour of

one process "leaks" into another; understanding the importance of, in some states, not listening to a synchronous (or sender-side asynchronous and data-free) channel – and last but not least: how to get rid of complexity by using these tools and combining from the best of our knowledge.

3. Conclusion

The CSP concept used in a small embedded controller certainly has helped us in making a complex design implementable. We have some 15 processes and drivers and, since roles are so clearly defined, the behaviour is explainable both at the process and the system level. (A team member burst out with this statement: "this is tenfold simpler than what I used to work with".) The main objective was that there is now no need to handle just any message in any internal state. Once the channel functionality is in place, it always works.

However, we have also seen how sensitive the methodology is to having a correct run-time layer. Assuring that the implementation is in fact correct is not trivial. Our strategy to help with this is always to "crash" on a rescheduling that is not anticipated, and be open about it and rectify immediately. Our heuristics, then, is the absence of any such behaviour. This is of course in addition to white-board studies of the solutions.

Also, having a critical view on communication patterns versus internal complexity, or cross-process complexity, has helped us. Rethinking one may help in simplifying the other.

Acknowledgements

The project leader Roar Bolme Johansen and fellow channel communicator Bjørn Tore Taraldsen for enduring non-blocking behaviour. My wife Mari, for just being.

References

[1] Øyvind Teig, "From message queue to ready queue (Case study of a small, dependable synchronous blocking channels API – Ship & forget rather than send & forget)". In ERCIM Workshop on Dependable Software Intensive Embedded Systems, in cooperation with 31st. EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Porto, Portugal, August/September 2005. Proceedings: ISBN 2-912335-15-9, IEEE Computer Press. [Read at **http://home.no.net/oyvteig/pub/pub_details.html#Ercim05**]

[2] M. Debbage, M. Hill, S. Wykes, D. Nicole, "Southampton's Portable occam Compiler (SPOC)", In: R. Miles, A. Chalmers (eds.), 'Progress in Transputer and occam Research', WoTUG 17 proceedings, pp. 40-55. IOS Press, Amsterdam, 1994.

Øyvind Teig is Senior Development Engineer at *Autronica Fire and Security, a UTC Fire and Security company*. He has worked with embedded systems for some 30 years, and is especially interested in real-time language issues. See **http://home.no.net/oyvteig/** for publications.