GRACEFUL TERMINATION — GRACEFUL RESETTING

P.H.Welch
Computing Laboratory, The University, Canterbury, Kent – CT2 7NF, ENGLAND

ABSTRACT

Correct — let alone graceful — termination of parallel systems is sometimes thought to be a difficult problem. This is particularly imagined to be so under the pure message-passing MIMD discipline of *occam* and *transputer* networks, where global operations (like setting a shared flag or abortions) are not allowed and where time-outs cannot be set for every communication. This paper describes some common, but erroneous, *occam* approaches to this problem and contrasts them with what can be done in *Ada* [0, 1, 2]. These methods are all rejected on the grounds of insecurity and performance overheads. A simple, legal, secure and efficient *occam* method is then presented. This method also solves a much more important problem — the general (or partial) resetting of a parallel system (or sub-system). The resetting mechanism is quite independent of the parallel application algorithm, which can therefore be developed without worrying about such matters. This separation of concerns is good software engineering and is fully supported by the *occam* philosophy. Finally, an application of this resetting mechanism is described that permits the dynamic reconstruction of *occam* network topologies.

The Problem

Given a (sub-)network of processes with arbitrary topology, message protocol and synchronisation regime, arrange for it to terminate. The initiative to kill the system may come from *one or more* of the processes themselves and/or from *one or more* points outside (if the network is not a closed system).

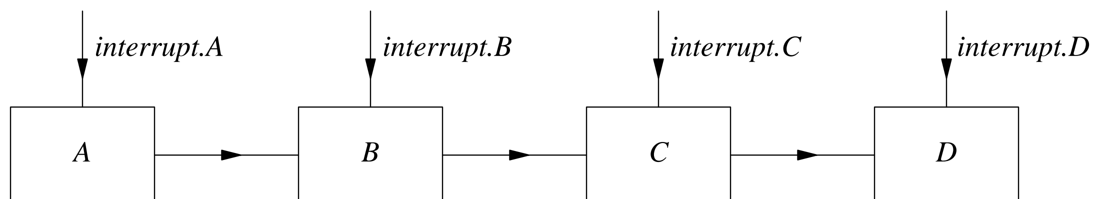
The pit-fall we have to avoid is committing a process to communicate with a terminated neighbour. If this were to happen, that communication would never terminate and, therefore, the network would never terminate.

How Not To Do It — I

Equip every process with an extra *interrupt* channel. If a process is implemented as a parallel network of sub-processes, ‘‘fan-out’’ this *interrupt* channel down to each of them. If a process has a sequential implementation, modify its algorithm so as to terminate if ever an *interrupt* signal arrives.

The trouble with this scheme is that its success is sensitive to the order in which the processes are closed down. Specifically, the processes must be killed off in a ‘‘topological’’ ordering with respect to the network data-flow.

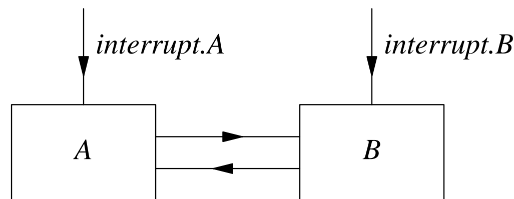
For instance, if the network were a pipeline of four processes :–



we must arrange for the *interrupts* to arrive in the sequence *A*, *B*, *C*, *D*.

Suppose we did not. Suppose that *interrupt.C* fired before *interrupt.B*. Then, there is a good chance that process *C* will terminate before process *B* notices its *interrupt.B*. In that case, process *B* may make a fatal attempt to output a message and get suspended for ever. The *interrupt.B* signal never gets acknowledged. This might cause further damage by blocking the *interrupt* generating process — thus leaving many other parts of the network still active.

Unfortunately, if the network has feed-back, there is *no* secure ordering possible for firing these *interrupts* — e.g. :–



If *A* terminates before *B*, *B* may get stuck trying to output — and vice versa!

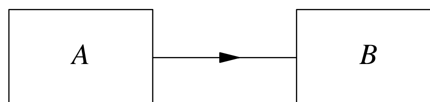
In [1], *A* is given the responsibility for pulling the *interrupt.B* (after it has received an *interrupt.A*). *B* is assumed to check its *interrupt.B* line between every output to *A*. Then :–

- If *B* were committed to output to *A* when *A* tries to *interrupt.B*, deadlock is avoided by having *A* always *interrupt.B* in parallel with inputting from *B*. *A* outputs this *interrupt.B* at high priority so that it will be pending at *B* before *B* completes its normal output to *A*. That way, *B* will detect it next time and not attempt to communicate again with *A*.
- Unfortunately, if the above pre-condition were not true, *B* will simply detect its *interrupt.B* and terminate without ever sending anything back to *A* — leaving *A* stranded. Of course, *A* might try timing-out on this final communication from *B* — but see the next section.

How Not To Do It — II

The process that decides to kill the system off simply terminates. Impose some ‘‘time-out’’ mechanism on all the lowest level (i.e. sequential) network processes so that they simply give up and die if they are blocked long enough awaiting input.

This scheme is very sensitive to setting the time-out values correctly. For instance, consider :–



Suppose *A* decides to terminate. *B* must know the minimum regularity with which it is supposed to receive messages in order to time-out correctly. If the communications are irregular (or if *A* were modified at some stage so that its regularity changed), *B* may time-out incorrectly. If *A* then tried to output, the system gets deadlocked as before.

On the other hand, if it was *B* that decided to terminate, *A* has no inputs on which to time-out and so detect that the system is closing down.

Neither of these problems, of course, can be overcome by setting time-outs on outputs. Even if such things were legal in *occam*, they still would not work — which brings us to the next section.

What Ada Might Do (or How Not To Do It — III)

Ada does have the capability of setting a time-out on an ‘‘output’’ (although the *Ada* equivalent notion is a ‘‘rendezvous call’’, over which the actual data-flow may be in either — or both — directions). However, as we are well aware from all the discussions about output guards in *occam*, the run-time overheads for this are horrendous. Even if we were able to accept this cost, this method is still insecure. If any time-outs are set wrong, there is an ever-present danger of the whole system closing down accidentally. I would not want to fly in an aeroplane controlled by such logic.

What Else Ada Might Do (or How Not To Do It — IV)

Ada might try setting a global flag that each component polls to see if it's time to close down. This could be mimicked in a non-shared-memory discipline (like *occam*) by having a special ''flag process'', running in parallel with the main network, that everyone polls. The problem with this is that the order of shut-down of the components is again very difficult to control, results in deadlock if done wrongly and, unfortunately, cannot be done correctly if the network has feedback (see **How Not to Do It — I**).

What Ada Often Does (or How Not To Do It — V)

Ada gives any process the ability to ''abort'' any other process whose name it knows. This is not a secure feature! I would not want any process of mine to be at the mercy of any (perhaps quite unconnected) process who happens to know its name. If a process needs to be closed down, it must be given the opportunity to perform whatever termination activities are essential to maintain a clean system environment. Good *Ada* design guidelines usually go to a lot of trouble (and it is a lot!!) to make processes anonymous — simply to prevent such abortions. *Occam*, of course, has no such features and none, happily, could be added — all processes in *occam* are anonymous.

What Ada Usually Does (or How Not To Do It — VI)

Ada has her ''terminate option''. This seems to be an elegant and secure feature designed precisely to solve this problem. *Occam* has no equivalent! If *occam* did have this feature, it might take the form of another kind of *ALT* construct — e.g. :–

```
TERMINATE ALT
  in.0 ? message
    ... process this message
  in.1 ? message
    ... process that message
```

A process that becomes blocked on an *TERMINATE ALT* construct would be marked as being ''terminatable''.

A deadlocked (sub-)network of terminatable processes terminates, courtesy of some magic in the run-time system. [A deadlocked (sub-)network of terminatable processes is one in which every process is waiting on a *TERMINATE ALT* for input either from other processes in this group or from terminated processes.]

Therefore, the solution would be to have a *TERMINATE ALT* guarding every input — even when there is only one channel to monitor. To kill the system, do whatever needs to be done to cause the right kind of deadlock — for instance, terminate all the sources of the data-flow. Unfortunately, this can again prove difficult for networks with feedback — if a process ends up blocked on output, it would not be terminatable!

There are three other problems :–

- this feature provides an improper ''way out'' from a loop that invalidates simple formal (or informal) rules for reasoning about loop properties. [In this, it is as bad as loop *exit* statements, subroutine *return* statements, *exception* raising, the *goto* and other nasty discontinuities that *occam* does not possess.]
- the run-time overheads for managing such options are alarming;
- the *transputer* (micro-coded) process scheduler does not maintain the information needed to implement it.

One of the reasons why *transputer* process scheduling is fast is that only local information need be examined. A process is associated with channels — not other processes. A process does not maintain pointers to whatever processes may be at the other ends of those channels — it does not know (and does not need to know) whether its neighbours are alive, dead, terminatable, siblings (i.e. created by the same *PAR* construct as itself) or even if they have ever existed. [Things are different for *Ada*!]

Therefore, the information is not directly available to find out if the conditions for triggering the *TERMINATE* option have been met. Such information is global information and is very expensive to maintain — especially in a distributed implementation! We are talking about a mass suicide pact which proceeds only if all the participants are convinced that all of them are actually going to do it — and where all the participants cannot be gathered together in the same place!!

What To Do If Really Desperate (or How Not To Do It — VII)

Drop into *assembler* or *C*. Ignore all *occam*'s safety rules. Have a go at the process scheduling data-structures! Good luck, but don't bet anything valuable (like someone's life!!) on the result.

How To Do It (If You're Sure You Really Want To)

Spread some ''poison'' throughout the circuit using the existing application channels.

For any message *PROTOCOL* on any channel, identify some value that can represent a special ''poisonous'' message that is distinct from ''normal'' messages. If no such value can be identified, extend the protocol with tags. For instance, if the protocol were just an *INT* and all *INT*s were valid messages, define :–

```
PROTOCOL TAGGED.INT
  CASE
    normal; INT
    poison
:
```

With such a *PROTOCOL*, a higher-level ''protocol'' must be observed : if we send *poison* down a channel, it is the last message we send down that channel.

Consider all the lowest level sequential components in the network. For example :–

```
PROC thing (CHAN OF TAGGED.INT in.0, in.1, in.2,
            CHAN OF TAGGED.INT out.0, out.1)
  ... local declarations
  SEQ
    ... initialise
    ... main.cycle (infinite)
:
```

Change the *main.cycle* of the component so that it :–

- terminates if it ever receives *poison* on any input;
- sets a flag (*FALSE*) to indicate each input on which the *poison* arrived (*poison* may arrive on more than one channel if there were a *PAR* input).

Then, add an extra *finalise* phase to the component, giving us :–

```
PROC terminatable.thing (CHAN OF TAGGED.INT in.0, in.1, in.2,
                         CHAN OF TAGGED.INT out.0, out.1)
  ... local declarations
  SEQ
    ... initialise
    ... main.cycle
    ... finalise
:
```

In this last phase, the component in parallel :–

- outputs *poison* on all output channels;
- ''black holes'' all input channels until *poison* arrives.

For example :–

```
{{{ finalise
PAR
  out.0 ! poison
  out.1 ! poison
  black.hole (flag.0, in.0)
  black.hole (flag.1, in.1)
  black.hole (flag.2, in.2)
}}}
```

where :–

```
PROC black.hole (BOOL flag,  CHAN OF TAGGED.INT in)
  WHILE flag
    in? CASE
      INT any:
      normal; any
        SKIP
      poison
        flag := FALSE
:
```

To kill the network from outside, simply inject in *poison*. If one of the internal components decides spontaneously to zap the system, all it has to do is terminate its *main.cycle* and enter its *finalise* phase. The *poison* will eventually reach everything ‘‘down-stream’’ from where it originated. Make sure that all components are down-stream from all potential starting points — this is a modest design constraint. This method will cope with multiple kill signals arriving from outside and/or being generated internally.

The network will gracefully terminate in some (unpredictable) order. When you try to input from a ‘‘poisoned’’ neighbour, you get ‘‘poisoned’’ yourself. Active parts of the network will not get blocked trying to output to ‘‘poisoned’’ neighbours. ‘‘Poisoned’’ neighbours leave behind a cluster of *black.hole* processes that stay alive, swallowing and acknowledging output so long as those active parts remain active.

There are, of course, some run-time penalties to pay. Every message communicated requires an extra check to detect *poison*. If variant protocols are necessary to represent *poison* values, each *normal* message is extended by the tag. Compared to the other approaches, these overheads are very light.

Unlike the other approaches, the method is secure (and may be formally proved so to be).

Are You Sure You Really Want To Do It?

Most of the sequential components I write emulate a single, rather small, object with a natural life expectation of infinity. They generally consist of a single *WHILE TRUE* loop.

Parallel processes do not need to terminate to produce useful results — unlike procedures or (first order) functions. Results are ‘‘communicated’’ on-the-fly, not ‘‘returned’’ upon exit.

One of the joys of designing like this is that we only need worry about loop-invariants and can forget about loop-termination. Much agonising has been reported from some quarters about the ‘‘serious difficulties’’ of reasoning about parallel processing that are caused by the fact that parallel loops might not terminate ‘‘together’’. Relax — with infinite loops, those problems never show up!

Another reason for not wanting to do this is the source text overhead for small processes. It can double the length of some of them! For designing the algorithms and understanding the semantics of the network, it just gets in the way. With the method outlined above, termination can be automatically ‘‘retro-fitted’’ should the need arise.

The best reason for not doing this is that it’s not really termination that is ever required — it’s resetting. If you really want to terminate a system, turn the power off!

The Real Problem

Given a (sub-)network of processes with arbitrary topology, message protocol and synchronisation regime, reset it to a known state.

The initiative that causes the reset may come from *one or more* of the processes and/or from *one or more* of the external inputs. ''Simultaneous'' resets must be catered for without falling over!

How To Solve The Real Problem

Spread the ''reset'' signal throughout the circuit using the existing application channels.

Proceed as for the termination problem. Identify a *reset* value for each *PROTOCOL* in the system (use tags if necessary).

Modify the lowest-level sequential processes as in the following example :–

```
PROC resettable.thing (CHAN OF TAGGED.INT in.0, in.1, in.2,
                       CHAN OF TAGGED.INT out.0, out.1)
  WHILE TRUE
    ... local declarations
    SEQ
      ... initialise to reset state
      ... main.cycle
      ... finalise
:
```

As before, the *main cycle* terminates if it receives a *reset* signal, flagging the channel (or channels) on which it was received.

The *finalise* phase is as before, with *reset* used instead of *poison*.

From wherever *resets* originate, a ''wave'' of signals ripples across the whole system. Colliding waves propagate no further. At any particular moment some of the network processes will still be in their *old* states, some will have just been *reset* and some will be into their *post-reset* activity.

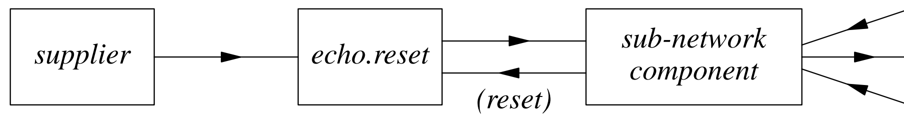
This loose coupling is perfectly secure. After any process (say *P*) has been *reset*, the first communication it makes on any of its channels will be the first communication that the process on the other end of the channel (say *Q*) makes on that channel after *Q* has been *reset*. Semantically, it is equivalent to stopping the whole system, waiting for every component to be *reset* and, only then, letting them all start again.

This mechanism for resetting or terminating networks is an example of how global changes of state can be consistently and safely managed using purely local (and therefore simple and fast) interactions. It reflects how *Nature* builds the ''real-world'' — complex *macro*-phenomema are merely the result of massively parallel interactions between simple *micro*-phenomema.

Resetting Sub-Networks

If we want to be able to reset just a sub-network, components that directly communicate with that sub-network must be made aware of this possibility. Their *main.cycle* needs be modified so that if a *reset* is received from the sub-network, they respond with a *finalise* (restricted to deal only with those channels that connect them to the sub-network). Their *main.cycle* would then resume without being broken.

Of course, this does mean that neighbouring components ought to take some input from the sub-network. We are using the sub-network — whatever it may be — as a medium for distributing the information about its impending reset. A resettable sub-network, therefore, needs protecting from a blind data-supplier. This can simply be achieved by offering the supplier an output channel from the sub-network (that is only ever used to carry a reset). On the other hand, if we do not mind introducing an extra buffer, we may even avoid modifying the data-supplier by introducing an extra process :–




```
PROC echo.reset (CHAN OF TAGGED.INT from.supplier, from.net, to.net)
  WHILE TRUE
    PRI ALT
      from.net ? CASE reset
        to.net ! reset                          -- "finalise"
      from.supplier ? CASE
        INT n:
        normal; n
          to.net ! normal; n
        reset
          PAR                                   -- "finalise"
            to.net ! reset
            from.net ? CASE reset
:
```

Such a process may be inserted on *every* input channel to the sub-network. The surrounding compnonents would then be insulated from any need to look out for and participate in any reset!

Similar things need to be done to terminate a sub-network successfully — with *poison* instead of *reset*. However, sub-network termination is somewhat pointless unless we want to re-allocate its resources in some way — e.g. to install a different sub-network. Otherwise, we may as well just deadlock the sub-network — hardly any code is ever required for that!

Dynamic Network Topology

Consider the *main.cycle* of *resettable.thing*. Suppose it is itself a sub-network of parallel processes, all of which terminate upon generation or consumption of a "*reset*" signal. Suppose further that the particular network that *main.cycle* installs depends upon run-time data computed during the *initialise* phase (where we could, of course, receive instructions from the outside world). We now have the ability to alter the network topology of an application dynamically. We can close down a sub-part and bring it up again in a different shape without stopping the rest of the network.

Given electronic switching of *transputer* links (or dynamic routing silicon on future *transputers*), there is no reason why much more of *occam* should not be used as the configuration language to accommodate these operations at the hardware level. This allows dynamic re-allocation (and re-connection) of physical *transputers* to different tasks in response to changing run-time conditions — all within the existing *occam* model of parallel processing. Perhaps, *occam* is not such a static language after all!

Final Comments

Resetting a system to a known state is a real requirement for most systems that have to operate in the "real world". Typical applications are in (real-time) control systems and (layered) communication network protocols.

The *occam* method of parallel system design directly reflects an "object-oriented" system analysis, permitting the rapid development of a range of secure and high-performance implementations with the basic functionality. "Special effects" — like this arbitrary resetting capability — can be grafted on afterwards in a quite mechanical way. Does anybody want to write a few tools?

References

[0] INMOS Ltd.: ‘‘*Occam 2 Reference Manual*’’; Prentice-Hall (ISBN 0-13-629312-3); 1987.

[1] A.Chalmers: ‘‘*Useful Titbits*’’; Occam User Group Newslettter No. 9 (pp. 15-19); September, 1988.

[2] Ada Joint Program Office: ‘‘*Ada Language Reference Manual*’’; 1983.