Concurrency and occam-π

occam Exercises (Santa Claus)

A classic exercise in concurrency is the *"Santa Claus"* problem:

> *"Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting."*

[J.A.Trono, "A new exercise in concurrency", SIGCSE Bulletin 26(3), 1994.]

There are several papers on this in the literature. It has been modeled using most concurrency tools (starting with semaphores and going through monitors, actors, Ada tasking, Java, C#, Polyphonic C# and, most recently, concurrent Haskell with software transactional memory). It is considered a *hard* problem – maybe because of the errors in the (earlier) presented models and their not too easy to understand *(or prove)* realizations.

Your task is to produce an occam-pi model that is (a) *obviously correct*, (b) *provably so* and (c) complete with animation so we can see the story unfold. To do the latter, the Santa-Claus system must signal on external wire(s) to a *renderer* process(es). Initially, this *animation* can be simple lines of text reporting each internal state change that is signaled. No conflicts with the above informal specification should be observed – e.g. Santa should not be seen consulting with only two elves, or delivering toys with one or more reindeer still on holiday. Further, your system must never deadlock, livelock or starve any principals from service – e.g. an elf that wants to consult Santa must not continually lose out to other elves wanting to consult Santa.

The system can be viewed as a control system for some machine, where the report signals are the controls and the *"Santa Claus"* rules specify how the machine must be operated. Breaking the rules will break the machine – with bad results. All of the above, apart from the starvation issue, can be formally verified *either* directly with CSP *or* with the help of the FDR model checker. 3D graphics is left as a follow-on exercise.

A (Fairly) Classical occam Model

Obviously, we need a process for *santa*, each *reindeer* and each *elf*. To assemble the elves into groups of three, a *waiting-room* holding just that number may help. To assemble the reindeer into groups of nine, a *stable* may work; alternatively, since all nine have to assemble, a barrier sync may do the trick.

However, Santa's operation of his reindeer and elves are really just the same: when woken up by either a reindeer party or elf party, he greets them all, works with them for a while and then lets them go.

A reindeer and elf do much the same thing: they work privately for a while (on holiday or making toys), assemble to meet Santa (in a stable or waiting room), work together with Santa (delivering toys or consulting on toy R&D), and then resume their private work.

So, try and reuse code. For instance, a *waiting-room* and *stable* could be different instances of the same process parameterised differently (full with 3, full with 9, *room-type*). The same is true for *reindeer* and *elves* (where the parameters are their *names* and *agent-type*). Note: the *room-type* (just **`VAL INT`** numbers), *names* (**`VAL INT`** id numbers) and *agent-type* (**`VAL INT`** numbers) are needed only for reporting purposes; the "full with 3" and "full with 9" parameters impact the internal logic. Of course, all processes will need channel (and, possibly, barrier) parameters.

Before writing any code, **draw the picture!** The *elves* could plug into shared channels serviced by the *waiting-room* – yes, they will need two shared channels: one to say "let me in" (which will block if the room is full) and one that lets an elf know when it gets through to Santa … the *waiting-room* process being responsible for contacting Santa when 3 elves are present. The story is the same for the *reindeer, stable* and Santa – except for 9 instead of 3.

This model is mostly classical occam (unless you use barrier synchronization). The elves/reindeer send their *names* (id numbers) to the waiting-room/stable, which pass them on to Santa. The *report* channel is easiest *shared* by all these parties (the sharing is occam-pi). The reports should be brief and logical – don't try to send text here (the *look-and-feel,* including the actual text shown, is the responsibility of the *renderer* process servicing the reports). Here is a possible **`CASE`** protocol for this channel (feel free to go with your own):

```
PROTOCOL AGENT.MESSAGE           --  an agent is a reindeer or an elf
  CASE
    on.my.own; INT; INT          --* kind of agent; id of agent
    ready; INT; INT              --* kind of agent; id of agent
    waiting; INT; INT            --* kind of agent; id of agent
    work.with.santa; INT; INT    --* kind of agent; id of agent
    done.working; INT; INT       --* kind of agent; id of agent
:

PROTOCOL SANTA.MESSAGE
  CASE
    agent.ready; INT             --* kind of agent
    greet; INT; INT              --* kind of agent; id of agent
    group.engaged; INT           --* kind of agent
    group.disengaged; INT        --* kind of agent
    goodbye; INT; INT            --* kind of agent; id of agent
:

PROTOCOL MESSAGE EXTENDS AGENT.MESSAGE, SANTA.MESSAGE:
```

Note: the above uses occam-pi's *protocol inheritance* (slides 27-37 of "shared-etc"). They make things neat and safe for channels shared by a variety of process types – in this case two: *agents* and *santa*. An *agent* is either a *reindeer* or an *elf*.

Note: use *time-outs* to model actual periods of holiday/toy-delivery/toy-making/consultation. The actual time-out periods should be set randomly between **`VAL`** limits. Santa decides on toy-delivery and consultation periods; the reindeer on holiday periods; the elves on toy-making periods. The 'course' library module has a **`random`** number generating **`FUNCTION`** (documentation at `http://occam-pi.org/occamdoc/frames.html`).

Mobile Channel Version

Instead of sending their *id numbers*, each agent (reindeer and elf) sends the (server-) end of an unshared channel (constructed once as the agent initiaslises) which it uses to communicate with Santa. The *waiting-room* or *stable* gathers these into a mobile array (of size 3 or 9) and sends that whole array to Santa. Communication with Santa then becomes *explicit* (as opposed to being just implied by the *id numbers* arriving). When the work with Santa is done, he pings the channel-ends back down their respective channels to return them to the agents that own them (for use next time). The latter effect needs a *recursive channel type* declaration (slides 49-50 of "mobiles").

Mobile Process Version

This time, each agent (elf and reindeer) is an instance of a *mobile process type*. They will need *work-station* (or *holiday-station*) processes to host them whilst doing their own thing. The *waiting-room* (or *stable*) hosts them when on the move to Santa. They will need individual *santa-stations* to host them when working with Santa. The agents need an *initialization* channel (to give them their *type*, *id number*, random number seed, etc. following construction), the general *report* channel and some channel(s) to communicate directly with Santa. The agents are moved through the (fixed) network via their hosts, plugging them into channels to Santa only when they get to him.

[Note: I will be happy to help people working on this (and any other) exercise after the course finishes and I return to the UK. Just contact me at my UK email address: `phw@kent.ac.uk`*]*

Peter Welch
(18th. November, 2007)