On Specifying Timeouts[*]

Rob van Glabbeek

National ICT Australia
and School of Computer Science and Engineering
The University of New South Wales
rvg@cs.stanford.edu

Abstract: This paper raises the question on how to specify timeouts in process algebra, and finds that the basic formalisms fall short in this task.

Consider the following protocol for a mail server:

> Set a timer for an unspecified but finite amount of time, and try to send a message again and again until it either succeeds or the timer goes off. In the latter case return an error message. Optionally, someone may deactivate the timer before it goes off, in which case the system may run forever.

My question is how to model this simple protocol by means of process algebra. Even though languages like CCS, CSP and ACP and their many variants have been around for twenty five years, it is still particularly tricky to do so. As this problem didn't specify any real-time constraints it appears less natural to use a real-time process algebra. The specification should keep it completely open how long each activity lasts. In particular, there is no upper bound on the number of trials that are made before the timer goes off. Still we know that within a finite amount of time either the message is send successfully, or an error message is returned, unless the timer is deactivated.

When abstracting, in part, from the timer, the process can be specified as

$$\texttt{set} \cdot \mu X.(\texttt{fail} \cdot X + \texttt{succeed} + \texttt{timeout} \cdot \texttt{error})$$

and a specification of the entire protocol (freely mixing ACCSP) could be

$$\texttt{set} \cdot \Big(\mu X.(\texttt{fail} \cdot X + \texttt{succeed} + \texttt{timeout} \cdot \texttt{error}) \Big\|_{\texttt{timeout}} \tau \cdot \texttt{timeout} + \texttt{deactivate}\Big).$$

However, this specification leaves open the option that the process keeps failing forever: the standard operational semantics of ACCSP generates a transition system that features a path with infinitely many `fail`-actions and no `deactivate` (see Figure 1).

One solution is to invoke Koomen's Fair Abstraction Rule (KFAR) [1] to prove, by abstraction from `fail`, that either `succeed` or `timeout` will happen eventually. However,

[*] Partly based on joint work with Frits Vaandrager.

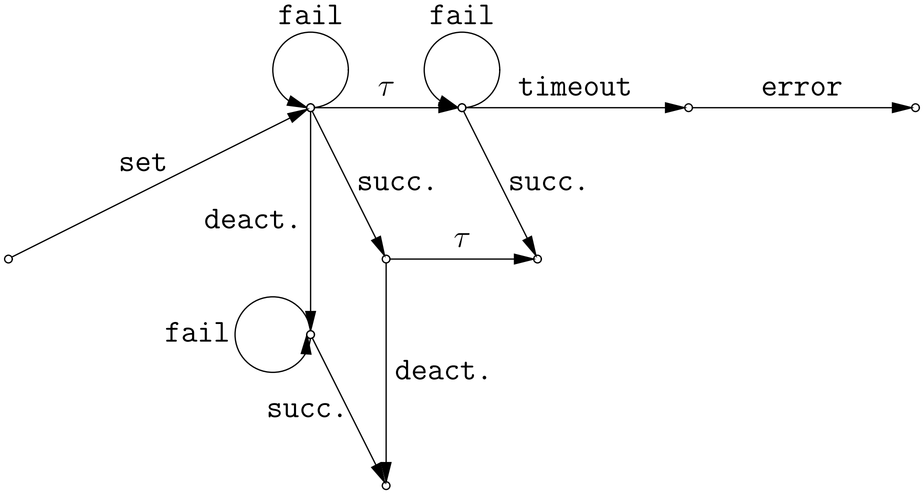

Figure 1: Process graph of the ACCSP specification

this hinges on a global fairness assumption that is not warranted in this example. Even if the action `deactivate` occurs, KFAR allows us to derive, contrary to the specification, that `succeed` will happen eventually.

What appears to be needed here is some kind of priority mechanism, saying that when `timeout` can occur, it takes precedence over the alternative actions `fail` and `succeed`. When performing abstraction by renaming into silent steps, priority mechanisms are cumbersome in process algebra, because in weak and branching bisimulation semantics the processes $a(b+c)$ and $a.(\tau.(b+c)+b)$ are equivalent, but if c has priority over b one would expect that only the latter can do a b-step.

Even when the priority mechanism is in place, the process of Figure 1 still allows an infinite sequence of `fail`-actions without `deactivate`, due to the interleaving of the components in the parallel composition. Maybe an elegant solution can be found

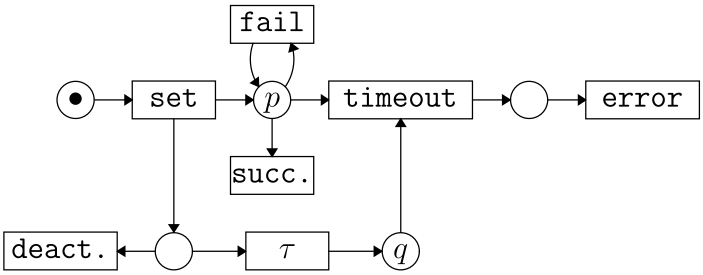

Figure 2: Petri net of the ACCSP specification

by modelling the specification as Petri net—see Figure 2. Under a normal progress assumption, provided that no `deactivate` occurs, sooner or later there will be a token in place q. Now `timeout` should have priority over the actions `fail` and `succeed` that compete for the token in p, but this priority takes effect only when there is a token in q. How to best formalise such reasoning is suggested as topic for future research.

[1] J.C.M. Baeten, J.A. Bergstra & J.W. Klop (1987): *On the consistency of Koomen's fair abstraction rule. Theoretical Computer Science* 51(1/2), pp. 129–176.