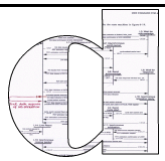
DEPENDABLE SYSTEMS AND SOFTWARE

Data Networks 2015 Dr. Arnd Hartmanns



ML2: Pipelined Protocols for Reliable Data Transfer

The soft deadline for this modelling lab is July 12. The hard deadline is July 19.

Submission: Your submission must be a `.zip` file that contains, at the top-level (i.e. not within any subfolders), the MODEST model files and text files with your solutions to the tasks of this lab, using the file names specified in the task descriptions. At the beginning of each MODEST model file, there must be a multi-line comment (`/* ... */`) containing a brief description of your model. Document any non-default settings you use for your experiments with modes (except for number of simulation runs where this is specified), and explain why you need to use them and why you can do so without affecting your results in an undesired way.

Task 1: Go-Back-N

In this task, we will use the template provided in dCMS to model and study the Go-Back-N protocol for pipelined reliable data transfer as discussed in the lecture and in the book. The `ChannelSR` and `ChannelRS` processes provided in the template model channels that can lose, but not corrupt or reorder, packets, and that include a send buffer of 16 packets. (Do not change the values of the constants defined in the template unless this is explicitly mentioned in a subtask below.)

(a) Complete the `Sender` and `Receiver` processes in the template to behave as the sender and receiver of the Go-Back-N protocol. Check that your model behaves correctly by

i) using a small value for `PKT_COUNT` and studying traces through your model generated by modes for varying small values for `N`, and

ii) verifying that the template's two error-checking properties `ProbError` and `DeliveredPackets` lead to the correct results using the original value for `PKT_COUNT` and varying small values for `N`.

Your model must be free of timed and structural nondeterminism.

(Submit your solution in a MODEST model file named `go-back-n.modest`.)

Hint: You can focus on the interesting aspects of your model in a trace by specifying a regular expression as a *variable filter*, e.g. `ChannelSR.packets|ChannelSR.timer` to only show the values of those two variables in the trace.

(b) Extend your model by adding properties and, if necessary, modifying the `Sender` and `Receiver` processes, to compute

i) the expected time t until the final packet has been delivered to the upper layers on the receiver's side, and

ii) the expected number r of retransmissions sent by the sender up to this point.

Report the results you obtain for these properties using $\texttt{N} = 4$ and 5000 simulation runs.

(Submit your solution in a text file named `go-back-n-results-b.txt`.)

(c) Determine the optimal window size `N` in terms of upper-layer throughput (which you can easily derive from t and `PKT_COUNT`). Describe your approach to finding this value (e.g. for which values of `N` do you perform how many simulation runs, and why do you do so), and briefly report on any intermediate results. Also describe how the number of retransmissions changes as you change `N`.

(Submit your solution in a text file named `go-back-n-results-c.txt`.)

(d) The value for `TIMEOUT` specified in the template is likely not optimal: It may be too low, leading to many unnecessary retransmissions, or it may be too high, affecting throughput. Find a value

for `TIMEOUT` that provides a better tradeoff between throughput and number of retransmissions, or confirm that the current value is a good one. Again, describe your approach and briefly report on any intermediate results.

(Submit your solution in a text file named `go-back-n-results-d.txt`.)

A Guide to Time and Delays in MODEST

This modelling lab requires you to build a MODEST model that includes timing aspects, but must not have timed nondeterminism. It is therefore important that you are very careful in how you model time and delays. This is a small guide to building good timed MODEST models, based on the problems that we saw in the first modelling lab.

First of all: We recommend that you do not use the `invariant` construct, since it affects all components in a parallel composition in ways that are often difficult to control. It should be easier to use only the `delay` construct, and/or clock variables in combination with `when` and `urgent`.

The `when` construct specifies a *guard*, i.e. a condition that governs whether some actions are enabled or not. An action cannot be performed if it is not enabled, even if it is available in the current location. (An action is available if it is on an edge out of the current location—this is what mosta shows.)

The `urgent` construct specifies a *deadline*, i.e. a condition that governs the passage of time: In some location, time can pass as long as no deadline of any *available* action evaluates to true. Observe that the effect of a deadline is independent of the effect of a guard.

Consider the following MODEST models, with declarations `clock c; action a;`:

1. `when(c >= 2) a`
2. `when(c >= 2) urgent(c >= 2) a`
3. `when(c >= 2) urgent(c >= 4) a`
4. `when(c >= 4) urgent(c >= 2) a`

In the first model, two time units can and must pass. After that, `a` can be performed at any later time (so there is temporal nondeterminism). In the second model, two time units can and must pass, and then `a` must be performed immediately at time 2 (so there is no temporal nondeterminism). In the third model, two time units must pass, and than `a` can be performed at any point until a further two time units have passed (so there is temporal nondeterminism); at time 4, `a` must finally be performed if that did not happen earlier. In the fourth model, two time units can pass, but then time stops: the deadline has become true, but no action is enabled.

`when urgent(e)` is a shortcut for `when(e) urgent(e)`. However, `urgent when(e)` is a shortcut for `urgent(true) when(e)` since `urgent` is a shortcut for `urgent(true)`.

In parallel composition, the guards deadlines of two synchronising actions are combined:

```
par { :: when(g1) urgent(d1) a :: when(g2) urgent(d2) a }
```

is equivalent to either `when(g1 && g2) urgent(d1 || d2) a` if `a` is declared as a normal "impatient" `action`, or to `when(g1 && g2) urgent(d1 && d2) a` if `a` is declared as a `patient action`. Using patient action thus allows one synchronisation partner to "wait" for the other. We recommend that you use patient actions whenever possible, because this makes it more difficult to create unwanted *timelocks* (cases where time stops, as in the fourth model in the list above) since conjunction is used to combine both the guards and the deadlines of synchronising actions. Note that you need to specify a deadline (which can often be just `true`) for every call of a patient action if you want to avoid temporal nondeterminism.

Finally, `delay` is nothing but a shortcut for certain variable assignments, guards and deadlines:

```
delay(exp, cond) P
```

is equivalent to `D()` where process `D` is defined as

```
process D() { clock c = 0; real x = exp; when urgent(c >= x && cond) P }.
```

(Think about why the condition `cond` is provided by `delay`—couldn't we just prefix the `delay` with appropriate `when`s and `urgent`s? Experiment with a minimal model with mosta if you are unsure.)