Practical Class 4 — Specifications[1]

This practical class uses FDR to investigate some more specifications.

Getting Started

1. Move to your CSP directory, and start an editor.

2. Type `fdr2 &`. You should see two windows: one entitled "FDR", and a file chooser window.

The Level Crossing

1. Copy the file `crossing.fdr2` to your directory, and load it into your editor and FDR. It contains the definitions for the first version of the level crossing from lectures. There are six assertions, corresponding to safety of *SYSTEM* and *SAFE_SYSTEM*, expressed in three different ways. Check all the assertions.

2. Notice that when all the events except *crash* are hidden, and *SYSTEM* fails to satisfy the specification, selecting "Debug" from the "Assert" menu does not show all of the trace; all the events except *crash* are replaced by "_tau", which is ASCII for τ. This is what happens when events are hidden; they get replaced by τ, which is a "silent" or "internal" event. Therefore there seems to be less information available about the trace which causes the specification to fail. However, by double-clicking on the process definitions which are shown on the left of the debug window, it is possible to see the actual events, generated by parts of the system, which correspond to the τ events. Try this, and see that in fact you can find out just as much about the behaviour of the process as you could when the events were not hidden.

3. The *CONTROL* process is defined in a particular way, but there are other possibilities. The definition in the file (and in the lecture notes) raises the gate immediately after a train has gone through the crossing. However, at a real level crossing, the gate stays down if another train

[1] This practical sheet is provided as course material to accompany the book 'Concurrent and Real Time Systems'.

is approaching. Change the definition of *CONTROL* so that it works in this more sophisticated way, and check the assertions again.

Peterson's Algorithm

The mutual exclusion problem is concerned with ensuring that two concurrently executing threads of control do not both reach a *critical region* of their code at the same time. This is important if a program requires exclusive control over resources such as memory in order to be sure of giving the right result.

Peterson's algorithm provides one way of achieving mutual exclusion for two concurrent processes `process 1` and `process 2`. It is described in pseudocode as follows:

```
program peterson;
(* Peterson's two-process mutual exclusion algorithm *)

var
  integer turn := 1;
  boolean flag1 := false;
  boolean flag2 := false;

process 1;
begin
  while true do
  {flag1:= true;    (* announce intent to enter *)
   turn:= 2;      (* give priority to other process *)
   while flag2 and (turn = 2) do null;
   CRITICAL SECTION 1
   flag1:= false
  }end
end;

process 2; (* similar *)
```

1. Copy the file `peterson1.fdr2` to your directory, and load it into your editor and FDR. It contains the (lengthy!) CSP definition of Peterson's algorithm for mutual exclusion. Define a process *SPEC* so that

$SPEC \sqsubseteq_t SYSTEM$ is a specification of mutual exclusion between $P1$ and $P2$, and use FDR to check that Peterson's algorithm does guarantee mutual exclusion.

2. The file `peterson2.fdr2` contains a shorter CSP definition of Peterson's algorithm. The use of channels instead of individual events makes the definitions of the processes and alphabets smaller. (Notice that $FLAG1$ and $FLAG2$ are essentially the same.) Read the definitions and see how they correspond to version 1.

Dekker's Algorithm

1. The following is pseudocode for Dekker's algorithm, the first mutual exclusion algorithm to be discovered.

```
(* Dekker's solution to the mutual exclusion problem *)

  integer turn := 1;
  boolean flag1 := false ;
  boolean flag2 := false;

process 1;
while true do
     {flag1 := true;
      while flag2 do
         {flag1 := false;
          while turn = 2 do null;
          flag1 := true
         };
      CRITICAL SECTION 1
      turn := 2;
      flag1 := false
     }

process 2; (* similar *)
```

Produce a CSP model of Dekker's Algorithm, following the example of either `peterson1.fdr2` or `peterson2.fdr2`, as you prefer. Check that mutual exclusion is guaranteed.