

## ◇ Peterson's Algorithm ◇

Now that we have CSP definitions for Peterson's algorithm, we can write a specification. It's very similar to the level crossing specification: we just define

$$SPEC = p1enter \rightarrow p1critical \rightarrow p1leave \rightarrow SPEC \\ | \quad p2enter \rightarrow p2critical \rightarrow p2leave \rightarrow SPEC$$

and then hide everything except the *enter*, *critical* and *leave* events.

$$SPEC \sqsubseteq_t (SYSTEM \setminus aF1 \cup aF2 \cup aT)$$

Notice that here we've used 3 events to represent passing through the critical section, whereas in the level crossing example we just had 2. This doesn't matter, but note that there must be at least 2 events in the sequence: if we only had *p1critical* and *p2critical* in the definitions of *P1* and *P2* then we wouldn't be able to write the specification in this way.

The programs from Coursework 1, and Dekker's Algorithm, can be specified in the same way.

## ◇ Defining Hiding ◇

The transition rules defining hiding are

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \quad a \in A$$

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{a} P' \setminus A} \quad a \notin A$$

As we saw when using FDR, the hidden events are replaced by  $\tau$ , representing “silent” or “internal” events.  $\tau$  events are not normally included in traces, although as we have seen, FDR can show where in a trace the  $\tau$  events occur. When we discuss traces, we will not include  $\tau$ .

## ◇ Livelock ◇

Recall the program *attempt1* from Coursework 1, which is Peterson's algorithm without the turn variable. It is possible for both processes to loop forever, repeatedly testing the *flag* variables — this is *livelock*.

### ◇ attempt1 in CSP ◇

The definitions are similar to those for Peterson's algorithm (second version), from the practical class. In the final definition of *SYSTEM*, all events except *enter*, *critical* and *leave* are hidden.

-- Attempt1 from Coursework 1, in CSP

```
channel flag1set, flag1read, flag2set,  
        flag2read:{1..2}.{|false,true|}  
channel enter, critical, leave:{1..2}
```

```
FLAG1(v) = flag1set?x?y -> FLAG1(y)  
          [] flag1read.1.v -> FLAG1(v)  
          [] flag1read.2.v -> FLAG1(v)
```

```
FLAG2(v) = flag2set?x?y -> FLAG2(y)  
          [] flag2read.1.v -> FLAG2(v)  
          [] flag2read.2.v -> FLAG2(v)
```

```
P1 = flag1set.1.true -> P1WAIT
```

```
P1WAIT = flag2read.1.true -> P1WAIT  
        [] flag2read.1.false -> P1ENTER
```

```
P1ENTER = enter.1 -> critical.1 -> leave.1  
         -> flag1set.1.false -> P1
```

```
P2 = flag2set.2.true -> P2WAIT
```

```
P2WAIT = flag1read.2.true -> P2WAIT  
        [] flag1read.2.false -> P2ENTER
```

```
P2ENTER = enter.2 -> critical.2 -> leave.2  
         -> flag2set.2.false -> P2
```

```
aP1 = {| flag1set.1, flag1read.1,  
        flag2set.1, flag2read.1,  
        enter.1, critical.1, leave.1 |}
```

```
aP2 = {| flag1set.2, flag1read.2,  
        flag2set.2, flag2read.2,  
        enter.2, critical.2, leave.2 |}
```

```
aF1 = {| flag1set,flag1read |}
```

```
aF2 = {| flag2set,flag2read |}
```

```
PROCS = P1 [ aP1 || aP2 ] P2
```

```
FLAGS = FLAG1(false) [ aF1 || aF2 ]  
        FLAG2(false)
```

```
SYSTEM = (PROCS [ union(aP1,aP2) ||  
                  union(aF1,aF2) ] FLAGS)  
         \ union(aF1,aF2)
```

When livelock occurs,  $P1$  is in a loop, doing the event  $flag2read.1.true$ . Similarly  $P2$ . Because the *flag* events are all hidden, this infinite loop turns into an infinite loop of  $\tau$  events. An infinite loop of  $\tau$  events is what CSP calls livelock, or *divergence*.

FDR can be used to detect divergence, and indeed detects it for this example. (Select “Livelock” from the tabs below the menu bar, then select *SYSTEM* in the “Implementation” field. Clicking on “Check” does a check for divergence.)

Actually, FDR is detecting something slightly different from what we have called livelock. We have considered livelock to be the situation in which *both* processes are in infinite  $\tau$  loops. FDR detects a situation in which *any* infinite  $\tau$  loop is possible. Because the parallel operator in CSP does not make any guarantees about how often each process will be executed, it is possible for  $P1$  to go into an infinite  $\tau$  loop as soon as  $P2$  has set `flag2` to `true` — even though we know that in the Pascal FC implementation,  $P1$  would not actually be executed forever. FDR detects the infinite  $\tau$  loop involving just one process. Care is needed to be sure that the situation detected by FDR would really arise in a more practical language such as Pascal FC.