

# An Introduction to Proofs about Concurrent Programs

K. V. S. Prasad  
(for the course TDA383/DIT390)  
Department of Computer Science  
Chalmers University

September 26, 2016

Rough sketch of notes released since it will be too late for this course if we wait till the notes are polished.

The plan is roughly this:

- **State diagram based proofs** - these are easy to understand in principle, and are what SPIN does. The text-book does these in enough detail, including in Chap. 3.

The remaining issue is how to tell SPIN what we think is true, so it can hunt for counter-examples. We have already seen one way, assertions. These are often enough for safety properties, but will not do for liveness in general. For that, we need LTL, covered later.

- **Syntactic proofs** (i.e., arguing from the program text). We do these in parallel with state diagrams, but do them in stages. The first stage uses informal but hopefully rigorous arguments, with a little propositional calculus notation for compactness. Simple theorems of propositional calculus are assumed. Temporal aspects (arguing about coming or previous states) are first treated entirely informally. Later sections use LTL notation. It is possible to formalise the programming language semantics, but that is not included here. So our arguments will continue to be held together by informal steps.

The goal is first to get you to follow the informal reasoning. Can you make your own arguments? (By the way, no one does formal reasoning before doing the informal thing first). Finally, and only in principle, the goal is to get you to see how to use a SPIN like model checker.

A basic idea we explore first is INVARIANTS.

## 1 Chapter 3

For the examples here, Chap. 3 of the textbook gives arguments based on the state diagram. Here we give text-based arguments.

## 1.1 Notation

Let the boolean  $p2$  mean that process  $p$  is at label  $p2$ , etc. Abusing notation, we sometimes also write  $p2$  to mean the label  $p2$  itself.

Logical symbols: We use  $\vee$  for *inclusive or*,  $\wedge$  for *and*,  $\neg$  for *not*,  $\rightarrow$  for *implies*, and  $\leftrightarrow$  for *implies and is implied by*.

## 1.2 Definitions: deadlock, livelock, starvation

We interpret `await B` to mean `loop skip until B`. The latter re-formulation can be called *busy-waiting*. To keep matters simple, assume also that  $p$  and  $q$  run on separate dedicated CPUs. Because of shared variables, this does not postpone issues of scheduling and *fairness*. Assume *weak fairness*: a scenario is weakly fair if an always enabled command will be executed at some point.

Since there is no blocking command in the mini-language we are using to program, these processes do not have a *blocked* state. So processes are either running or terminated. This means in this set-up we cannot have *deadlock*, which we define as “everyone blocked”. We can have *livelock*, which we define as “everyone busy-waiting”. Note that these definitions differ from those of the textbook (I find those definitions confusing).

We agree with the textbook’s definition of (individual) *starvation*: a process can get stuck forever (busy)-waiting to enter its critical section. A special case is that of *non-competitive starvation*, or NC-starvation, where  $p$  starves if  $q$  loops in its NCS.

A working equivalence is that in *deadlock* and *livelock*, all processes starve together. In individual starvation, a scenario exists where one particular process starves. The third attempt below shows a program that can livelock even though no process NC-starves.

## 1.3 First attempt, Alg. 3.5, p. 53

The program:

| integer turn := 1 |                  |
|-------------------|------------------|
| p                 | q                |
| loop forever      | loop forever     |
| p1: await turn=1  | q1: await turn=2 |
| p2: turn:=2       | q2: turn:=1      |

We write  $t$  for the variable `turn`, and let  $t1$  mean  $t = 1$  and  $t2$  mean  $t = 2$ .

Then we have invariants:  $T_1 = t1 \vee t2$  and  $T_2 = \neg(t1 \wedge t2)$ . The first is established by noting what values are assigned to  $t$ , and the second follows from the nature of variables—they cannot hold two values simultaneously.

Then it follows that  $p2 \rightarrow t1$  because  $p$  has just got past  $p1$ , and any interference from  $q$  can only result in (re)-setting  $t$  to 1. Similarly,  $q2 \rightarrow t2$ .

### 1.3.1 Mutex

We have to show that  $M = \neg(p2 \wedge q2)$  is invariant. We have  $p2 \rightarrow t1 \rightarrow \neg t2 \rightarrow \neg q2$ , and similarly  $q2 \rightarrow \neg p2$ , so  $M$  holds.

### 1.3.2 Livelock

Let  $L = p1 \wedge \neg t1 \wedge q1 \wedge \neg t2$ . Then  $L$  contradicts  $T_1$ . Thus  $\neg L$  is an invariant, and since  $L$  defines livelock, we have shown that livelock cannot happen.

### 1.3.3 Starvation

NC-starvation is possible. The scenario  $p1, p2, q1$  achieves this if  $q1$  loops, which it may, according to the conditions of the CS problem.

## 1.4 Second attempt, Alg. 3.7, p. 56

| boolean wantp := false, wantq := false |                         |
|--|-------------------------|
| p                                      | q                       |
| loop forever                           | loop forever            |
| p1: await wantq = false                | q1: await wantp = false |
| p2: wantp := true                      | q2: wantq := true       |
| p3: wantp := false                     | q3: wantq := false      |

The program:

We write  $wp$  for  $\text{wantp}$  and  $wq$  for  $\text{wantq}$ .

Note that only  $p$  sets  $wp$  and only  $q$  sets  $wq$ . Let  $T_1 = (p1 \vee p2) \leftrightarrow \neg wp$ , and  $T_3 = p3 \leftrightarrow wp$ . Then  $T_1$  and  $T_2$  are invariant. Similar invariants hold for  $q$ .

Note that we cannot claim  $p2 \rightarrow \neg wq$  even though  $\neg wq$  is needed for  $p$  to get past  $p2$ , since we do not know where  $q$  is. It may just have executed  $q2$ .

### 1.4.1 Mutex

This would require that  $(p2 \vee p3) \rightarrow \neg(q2 \vee q3)$ . But to ensure anything about where  $q$  is, we have to ensure something about  $wq$ . For example,  $wq \rightarrow \neg q2$ . The premise for the mutex statement tells us nothing about  $wq$ . So we cannot prove mutex, and indeed it is easy to write a scenario where it is broken:  $p1, q1$ .

### 1.4.2 Livelock

Let  $L = p1 \wedge wq \wedge q1 \wedge wp$ . Then  $L$  defines livelock, and contradicts  $T_1$ , so  $\neg L$  is invariant. That is, livelock cannot happen.

### 1.4.3 Starvation

Let  $S = p1 \wedge wq \wedge q1$ . Then  $S$  defines NC-starvation of  $p$ , where  $q$  loops in its NCS. But  $q1 \rightarrow \neg wq$ , so  $S$  is self-contradictory. That is,  $\neg S$  is invariant, and  $p$  cannot starve this way. Note that here  $q$  is looping in its NCS, not in its pre-protocol. Both are notated  $q1$  in the abbreviated program.

But  $p$  can starve if it is only scheduled to look at  $wq$  after  $q2$ . Is tis weakly fair?

## 1.5 Third attempt, Alg. 3.8, p. 57

| boolean wantp := false, wantq := false |                          |
|--|--------------------------|
| p                                      | q                        |
| loop forever                           | loop forever             |
| p1: non-critical section               | q1: non-critical section |
| p2: wantp := true                      | q2: wantq := true        |
| p3: await wantq = false                | q3: await wantp = false  |
| p4: critical section                   | q4: critical section     |
| p5: wantp := false                     | q5: wantq := false       |

The program:

We write  $w_p$  for **wantp** and  $w_q$  for **wantq**. Again, only  $p$  sets  $w_p$  and only  $q$  sets  $w_q$ .

Let  $T_1 = (p_1 \vee p_2) \leftrightarrow \neg w_p$ , and  $T_2 = (p_3 \vee p_4 \vee p_5) \leftrightarrow w_p$ . Then  $T_1$  and  $T_2$  are invariant. Similar invariants hold for  $q$ .

Note that we cannot claim  $p_4 \rightarrow \neg w_q$  even though  $\neg w_q$  is needed for  $p$  to get past  $p_3$ , since we do not know where  $q$  is. It may just have executed  $q_2$ .

### 1.5.1 Mutex

We have to show that  $M = \neg(p_4 \wedge q_4)$  is invariant.  $M$  holds at the start. Can we go from a state where  $M$  holds to one where it doesn't? Suppose  $p$  is at  $p_4$ , and  $q$  is not already at  $q_4$ . To get to  $q_4$ , we need  $\neg w_p$  so that  $q$  can get past  $q_3$ . But this contradicts  $T_2$ . So  $M$  is invariant: mutex is assured.

### 1.5.2 Livelock

Let  $L = p_3 \wedge w_q \wedge q_3 \wedge w_p$ ; then  $L$  defines livelock. But  $L$  can be true; nothing in the invariants contradicts it, so livelock can happen. A scenario for this is:  $p_1, q_1, p_2, q_2, p_3, q_3$ .

### 1.5.3 Starvation

Let  $S = p_3 \wedge w_q \wedge q_1$ . If  $S$  can be true,  $p$  can be NC-starved. But  $T_1$  says  $q_1 \rightarrow \neg w_q$ , which contradicts  $S$ . So  $\neg S$  is invariant; NC-starvation cannot occur.

But can  $p_3 \wedge w_q$  forever, thus starving  $p$ , in some other scenario? Since  $w_q \leftrightarrow (q_3 \vee q_4 \vee q_5)$  is invariant, this means  $(q_3 \vee q_4 \vee q_5)$ . The case  $q_3$  is livelock; and  $q$  has to pass  $q_4, q_5$  in finite time. So there is no individual starvation.

## 1.6 Fourth attempt, Alg. 3.9, p. 59

| boolean wantp := false, wantq := false |                          |
|--|--------------------------|
| p                                      | q                        |
| loop forever                           | loop forever             |
| p1: non-critical section               | q1: non-critical section |
| p2: wantp := true                      | q2: wantq := true        |
| p3: while wantq                        | q3: while wantp          |
| p4: wantp := false                     | q4: wantq := false       |
| p5: wantp := true                      | q5: wantq := true        |
| p6: critical section                   | q6: critical section     |
| p7: wantp := false                     | q7: wantq := false       |

The program:

Note that this program has dispensed with the `await` statement, writing out the *busy-waits* explicitly.

We write  $w_p$  for `wantp` and  $w_q$  for `wantq`. Again, only  $p$  sets  $w_p$  and only  $q$  sets  $w_q$ .

Let  $T_1 = (p_1 \vee p_2 \vee p_5) \leftrightarrow \neg w_p$ , and  $T_2 = (p_3 \vee p_4 \vee p_6 \vee p_7) \leftrightarrow w_p$ . Then  $T_1$  and  $T_2$  are invariant. Similar invariants hold for  $q$ .

Note that we cannot claim  $p_4 \rightarrow \neg w_q$  even though  $\neg w_q$  is needed for  $p$  to get past  $p_3$ , since we do not know where  $q$  is. It may just have executed  $q_2$  or  $q_5$ .

### 1.6.1 Mutex

We have to show that  $M = \neg(p_6 \wedge q_6)$  is invariant.  $M$  holds at the start. Can we go from a state where  $M$  holds to one where it doesn't? Suppose  $p$  is at  $p_6$ , and  $q$  is not already at  $q_6$ . To get to  $q_6$ , we need  $\neg w_p$  so that  $q$  can get past  $q_3$ . But this contradicts  $T_2$ , which says  $p_6 \rightarrow w_p$ . So  $M$  is invariant: mutex is assured.

### 1.6.2 Livelock

Let  $L = p_3 \wedge w_q \wedge q_3 \wedge w_p$ ; then a path where states repeatedly satisfy  $L$  defines *extended livelock*. But  $L$  can be true; nothing in the invariants contradicts it, so livelock can happen. A scenario for this is:  $p_1, q_1, p_2, q_2, p_3, q_3$ , followed by the execution of the pre-protocol loops  $p_3, p_4, p_5$  and  $q_3, q_4, q_5$  in parallel.

### 1.6.3 Starvation

Let  $S = p_3 \wedge w_q \wedge q_1$ . If  $S$  can be true,  $p$  can be NC-starved. But  $T_1$  says  $q_1 \rightarrow \neg w_q$ , which contradicts  $S$ . So  $\neg S$  is invariant; NC-starvation cannot occur.

But can  $p_3 \wedge w_q$  forever, thus starving  $p$ , in some other scenario? Since  $w_q \leftrightarrow (q_3 \vee q_4 \vee q_6 \vee q_7)$  is invariant, this means  $(q_3 \vee q_4 \vee q_6 \vee q_7)$ . Suppose  $p$  is in its pre-protocol loop. Either  $q$  is also stuck in its pre-protocol loop, or it escapes. In the latter case,  $w_q$  is false in  $q_1$ , so  $p$  is stuck forever only if the scheduler never lets  $p_3$  execute when  $q_1$ . Fair?