An Introduction to Proofs about Concurrent Programs

K. V. S. Prasad
(for the course TDA384/DIT391)
Department of Computer Science
Chalmers University

Lecture 8, Monday 18 Sep 2017

These are only a rough sketch of notes, released now since it will be too late for this course if we wait till the notes are polished.

1 Examples from Chapter 3 of Ben-Ari's book

This chapter roughly parallels Chap 3 of Ben-Ari, which analyses a series of programs using state diagrams. Here, we study the same programs by analysing the text.

- **State diagram based proofs** - these are easy to understand in principle, though it is also easy to see that the diagrams quickly become too big for manual analysis. Large state diagrams are analysed mechanically by tools called *model checkers* such as SPIN. (Neither SPIN, nor its modelling language, PROMELA, are examinable material in our course).

  So one goal for us is to learn, if only in principle, how to use a model checker. Typically, we say what properties we think a program has, and the model checker hunts for counter-examples. Assertions are one way to state program properties. These are often enough for safety properties, but will not do for liveness in general. For that, we need (linear) temporal logic (LTL), covered in later notes.

- **Syntactic proofs** (i.e., arguing from the program text). We do these here in parallel with state diagram proofs, but 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).

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 "Hardware processes"

We have so far worked with an abstract world implemented by run-time support (RTS). This world consists of entities that we can call "software processes", events, messages, atomic actions, and so on. Here, a process can be marked *blocked* while waiting for some event, and be unblocked and marked *ready* by the RTS when the event occurs. So the command `await B` can be interpreted as `block until B`. Only ready processes are *scheduled* (given CPU time); blocked processes are not, since they cannot run.

For these software processes, we also introduced semaphores and other abstract synchronisation and communication structures. How these structures and software processes are implemented by the RTS is not a concern for us here[1]. We only need to know what has been implemented. One striking feature of this abstract world is that even for a ready process, we do not know if it is actually running. A related matter is that we know nothing about the speed at which any process runs.

By contrast, the world in this chapter is simpler. The processes here can be called "hardware processes". Once *spawned*, these simply run until they terminate: they do not block. We assume that each process runs on a separate dedicated CPU. We interpret `await B` to mean

```
loop
  skip
until B
```

; that is, keep doing a skip (do nothing) until B becomes true. This re-formulation is called *busy-waiting*.

1.3 Definitions: fairness, deadlock, livelock, starvation

Because only one CPU at a time can access a shared variable, we still face issues of scheduling—not a process onto a CPU, but a CPU to a shared variable by a bus arbitrator or similar. We assume *weak fairness*: a scenario is weakly fair if a continually enabled command will be executed at some point.

Since there is no blocked state, and no blocking command, 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

[1] Some detail can be found in Ben-Ari's book. For more, see books on Operating Systems (OS), such as "Operating Systems: Three Easy Pieces", by Remzi and Andrea Arpaci-Dusseau, 2015.

non-competitive starvation, or NC-starvation, where p starves if q loops in its NCS.

A working equivalence is that in deadlock and livelock, processes mutually starve each other. 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, i.e., the only starvation possible is mutual.

In the proofs that follow, a basic idea we explore is that of INVARIANTS.

1.4 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: $\mathrm{T}_1{=}t1 \vee t2$ and $\mathrm{T}_2{=}\neg(t1 \wedge t2)$. 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, $q2 \rightarrow t2$.

1.4.1 Mutex

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

1.4.2 Livelock

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

1.4.3 Starvation

NC-starvation is possible. If $q1$ loops in its NCS (before it executes the `await`, which it may, according to the conditions of the CS problem), the scenario $p1, p2, q1$ starves p. In this scenario, $t2$ holds forever, so p will get stuck in $p1$.

1.5 Second attempt, Alg. 3.7, p. 56

The program:

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

We write wp for `wantp` and wq for `wantq`.

Note that only p sets wp and only q sets wq. Let $\mathrm{T}_1 = (p1 \vee p2) \leftrightarrow \neg wp$, and $\mathrm{T}_3 = p3 \leftrightarrow wp$. Then T_1 and T_2 are invariant. Similar invariants hold for q.

Note that we cannot claim $p2 \to \neg wq$ 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 $q2$.

1.5.1 Mutex

This would require that $(p2 \vee p3) \to \neg(q2 \vee q3)$. But to ensure anything about where q is, we have to ensure somthing about wq. For example, $wq \to \neg q2$. 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: $p1, q1$.

1.5.2 Livelock

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

1.5.3 Starvation

As in the first attempt above, both the NCS and the pre-protocol are notated $q1$ in the abbreviated program. Let $\mathrm{S} = p1 \wedge wq \wedge q1$. If q is looping in its NCS, $q1$ will always be true. Can then S be always true? If so, it will show NC-starvation of p. But $q1 \to \neg wq$, so S is self-contradictory. That is, $\neg$S is invariant, and p cannot starve this way.

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

1.6 Third attempt, Alg. 3.8, p. 57

The program:

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

We write wp for `wantp` and wq for `wantq`. Again, only p sets wp and only q sets wq.

Let T_1=$(p1 \vee p2) \leftrightarrow \neg wp$, and T_2=$(p3 \vee p4 \vee p5) \leftrightarrow wp$. Then T_1 and T_2 are invariant. Similar invariants hold for q.

Note that we cannot claim $p4 \rightarrow \neg wq$ even though $\neg wq$ is needed for p to get past $p3$, since we do not know where q is. It may just have executed $q2$.

1.6.1 Mutex

We have to show that M=$\neg(p4 \wedge q4)$ 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 $p4$, and q is not already at $q4$. To get to $q4$, we need $\neg wp$ so that q can get past $q3$. But this contradicts T_2. So M is invariant: mutex is assured.

1.6.2 Livelock

Let L= $p3 \wedge wq \wedge q3 \wedge wp$; then L defines livelock. But L can be true; nothing in the invariants contradicts it, so livelock can happen. A scenario for this is: $p1, q1, p2, q2, p3, q3$.

1.6.3 Starvation

Let S=$p3 \wedge wq \wedge q1$. If S can be true, p can be NC-starved. But T_1 says $q1 \rightarrow \neg wq$, which contradicts S. So $\neg$S is invariant; NC-starvation cannot occur.

But can $p3 \wedge wq$ forever, thus starving p, in some other scenario? Since $wq \leftrightarrow (q3 \vee q4 \vee q5)$ is invariant, this means $(q3 \vee q4 \vee q5)$. If either $q4$ or $q5$ can hold forever, individual starvation can result. But q has to pass $q4, q5$ in finite time. So there is no individual starvation, but mutual starvation is possible (livelock, with the case $q3$).

1.7 Fourth attempt, Alg. 3.9, p. 59

The program:

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

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

We write wp for `wantp` and wq for `wantq`. Again, only p sets wp and only q sets wq.

Let T_1=$(p1 \vee p2 \vee p5) \leftrightarrow \neg wp$, and T_2=$(p3 \vee p4 \vee p6 \vee p7) \leftrightarrow wp$. Then T_1 and T_2 are invariant. Similar invariants hold for q.

Note that we cannot claim $p4 \rightarrow \neg wq$ even though $\neg wq$ is needed for p to get past $p3$, since we do not know where q is. It may just have executed $q2$ or $q5$.

1.7.1 Mutex

We have to show that M=$\neg(p6 \wedge q6)$ 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 $p6$, and q is not already at $q6$. To get to $q6$, we need $\neg wp$ so that q can get past $q3$. But this contradicts T_2, which says $p6 \rightarrow wp$. So M is invariant: mutex is assured.

1.7.2 Livelock

Let L= $p3 \wedge wq \wedge q3 \wedge wp$; 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: $p1, q1, p2, q2, p3, q3$, followed by the execution of the pre-protocol loops $p3, p4, p5$ and $q3, q4, q5$ in parallel.

1.7.3 Starvation

Let S=$p3 \wedge wq \wedge q1$. If S can be true, p can be NC-starved. But T_1 says $q1 \rightarrow \neg wq$, which contradicts S. So $\neg$S is invariant; NC-starvation cannot occur.

But can $p3 \wedge wq$ forever, thus starving p, in some other scenario? Since $wq \leftrightarrow (q3 \vee q4 \vee q6 \vee q7)$ is invariant, this means $(q3 \vee q4 \vee q6 \vee q7)$. 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, wq is false in $q1$, so p is stuck forever only if the scheduler never lets $p3$ execute when $q1$. Fair?