Solutions for Exercise Class 3

1. (a) $\Box\neg Snows(Bermuda)$

(b) $\Box(Snows(Helsinki) \Rightarrow Snows(Finland))$

(c) $\Box(Snows(Norway) \Rightarrow \Diamond Snows(Sweden))$

(d) $\Box\Diamond Snows(DK) \wedge \Box\Diamond Snows(NZ) \wedge \Box\neg(Snows(DK) \wedge Snows(NZ))$

(e) $\Box(Snows(Sahara) \Rightarrow \Box Snows(Sahara))$

(f) $\Box\exists x : Snows(x)$

2. Solution for Theory.1

Question 1.1

(a) I holds initially since $x = 0 \wedge y = 0$.

All three a-actions are potenially dangerous for I:

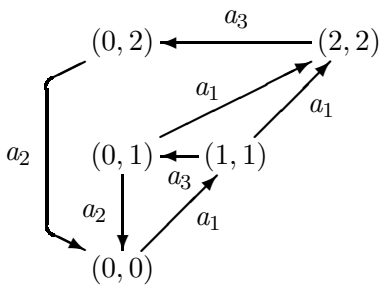
a_1: Assuming I to hold before the actions, the precondition ensures that $0 < y \leq 2$ after the actions. Further $x = y$ and therefore I holds after execution of a_1.

a_2: The precondition $x = 0$ and the effect ensures $x = 0 \wedge y = 0$ after the actions. Thus, I holds.

a_3: If I holds before the actions, we must have $0 = x \leq y \leq 2$ after the actions, thus I still holds.

Since I holds initially and is preserved by all atomic actions, I is an invariant of the program.

(b) Transition graph:



Further there are a_3 self-loops on states $(0,0)$, $(0,1)$, and $(0,2)$ plus an a_2 self-loop on state $(0,0)$. Since these are relevant only for liveness properties and only under less than weak fairness assumptions, they are not shown.

(c) From the transition graph, showing the complete reachable state space, we see by inpection that $(x, y) = (1, 2)$ is not reachable and thus $\neg(x = 1 \wedge y = 2)$ is an invariant of the program.

Question 1.2

[Assuming at least weak fairness, the self-loops on the transition graph are irrelevant.]

(a) Given a transition graph, weak fairness ensures that the execution cannot remain forever in a state which can be left by one or more actions. For the transition graph we therefore conclude that any execution must pass through $(x, y) = (1, 1)$ over and over again. Thus, $\Box\Diamond x = 1$ is satisfied.

(b) Consider the infinite execution

$$(0,0) \xrightarrow{a_1} (1,0) \xrightarrow{a_3} (0,1) \xrightarrow{a_2} (0,0) \xrightarrow{a_1} (1,0) \xrightarrow{a_3} \cdots$$

In this execution, all actions are executed infinitely often, thus strong fairness is satisfied. However, no state with $x = 2$ is met.

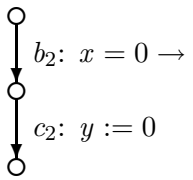
Note: The notion of fairness is related to *actions*. If a particular action is taken *in any state*, fairness is satisfied for that action.

Question 1.3

(a) If a_2 cannot be considered atomic as a whole, by the default assumption of atomic reads and writes, it will correspond to

$$b_2\colon \langle\, \textbf{await } x = 0 \,\rangle; \quad c_2\colon \langle\, y := 0 \,\rangle$$

This can be depicted by the transition diagram:



Now, the interleaving

$$(0,0) \xrightarrow{b_2} (0,0) \xrightarrow{a_1} (1,1) \xrightarrow{c_2} (1,0)$$

violates I.

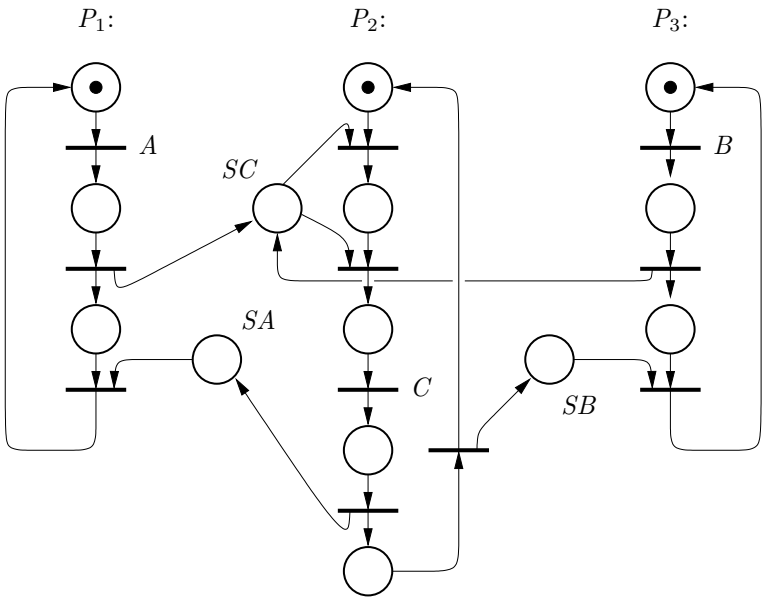
(b) H can be defined as:

$$H \;\triangleq\; I \wedge (at\ c_1 \Rightarrow y < 2) \wedge (at\ d_1 \Rightarrow x \leq t < 2)$$

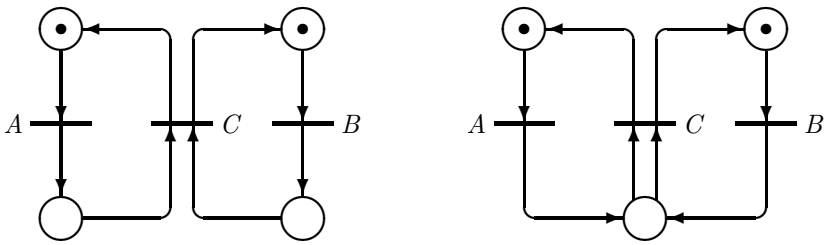
which is readily seen to hold initially and imply I. Further, H is preserved by all atomic actions. Especially, the conjunct $(at\ d_1 \Rightarrow x \leq t < 2)$ is needed to conclude I after d_1.

3. Solution for Sema.1

Direct "translation" of the program to a Petri Net:



Two Petri Nets that both directly expresses how A, B, and C are synchronized:



3. Solution for Sema.2

We here chose P_D to be the "master" that starts up P_A which in turn signals P_B or P_C:

var SA, SBC, SD : *semaphore* := 0;

process P_A;	**process** P_B;	**process** P_C;	**process** P_D;
repeat	**repeat**	**repeat**	**repeat**
P(SA);	P(SBC);	P(SBC);	V(SA);
A;	B;	C;	D;
V(SBC)	V(SD)	V(SD)	P(SD)
forever;	**forever**;	**forever**;	**forever**;

4. Solution for Sema.4

Proposal I

For each pair of processes (P_i, P_j) we introduce a semaphore S_{ij} that is used for signalling from P_i to P_j. Each process signals the two other ones and awaits a signal from each of these:

var $S_{AB}, S_{AC}, S_{BA}, S_{BC}, S_{CA}, S_{CB}$: *semaphore* := 0;

$SYNC_A$:	$SYNC_B$:	$SYNC_C$:
V(S_{AB});	V(S_{BA});	V(S_{CA});
V(S_{AC});	V(S_{BC});	V(S_{CB});
P(S_{BA});	P(S_{AB});	P(S_{AC});
P(S_{CA});	P(S_{CB});	P(S_{BC});

This solution is readily shown to be correct using the semaphore invariant.

Proposal II

Each process has a semaphore to be signalled by the other processes. Each process starts by signalling to each of the two other ones and then waits on its own semaphore for two signals (which are expected to from each of the other processes).

var S_A, S_B, S_C : *semaphore* := 0;

$SYNC_A$:	$SYNC_B$:	$SYNC_C$:
V(S_B);	V(S_A);	V(S_A);
V(S_C);	V(S_C);	V(S_B);
P(S_A);	P(S_B);	P(S_C);
P(S_A);	P(S_B);	P(S_C);

This solution is correct but works so marginally that it cannot be shown directly by use of the semaphore invariant!

NB: The "dual" solution below, where each process signals its own semaphore twice and then waits on each of the other semaphores *does not work*. The error is that a process may "use" a signal that was supposed for another process.

var S_A, S_B, S_C : *semaphore* := 0;

$SYNC_A$:	$SYNC_B$:	$SYNC_C$:
V(S_A);	V(S_B);	V(S_C);
V(S_A);	V(S_B);	V(S_C);
P(S_B);	P(S_A);	P(S_A);
P(S_C);	P(S_C);	P(S_B);

The circular solution below *does not work either*:

var S_A, S_B, S_C : *semaphore* := 0;

$SYNC_A$:	$SYNC_B$:	$SYNC_C$:
V(S_B);	V(S_C);	V(S_A);
P(S_A);	P(S_B);	P(S_C);

It has the general fault that a process does not wait for *all* the other processes. If a signalling "the other way round" is added, it will work.