Deadlock-freedom

await statement

await B **then** S **ta**
where B is a boolean expression and S is a statement not containing a **parbegin** or another **await** statement. Let us recall the semantics of the **await** statement:

When a process attempts to execute an **await**, it is delayed until the condition B is true. Then the statement S is executed as an indivisible action. Upon termination of S, the processing continues. If two or more processes are waiting for the same condition B, any of them may be allowed to proceed when B becomes true, while the others continue waiting. The waiting processes can be scheduled by any scheduling rule.
Note that evaluation of B is part of the indivisible action of the **await** statement; after B has been evaluated but before S begins execution another process is not allowed to change variable B and to make it false.

Deadlock

Example

Consider the following parallel program:

```
a=0; b=0;
parbegin
while TRUE do
  α0: a:=a+1;
  α1: await b≠0 then
    α2: a:=a+3
    ta;
od
||
while TRUE do
  β0: a:=2*a;
  β1: await a≠1 then
    β2: b:=b+1
    ta;
od
parend
```

The following tables illustrate the behaviour of the given parallel program:

step	action	a	b
		0	0
1.	β_0: a:=2*a	0	0
2.	α_0: a:=a+1	1	0
	deadlock		

step	action	a	b
		0	0
1.	α_0: a:=a+1	1	0
2.	β_0: a:=2*a	2	0
3.	β_1: await $a \neq 1$		
4.	β_2: b:=b+1	2	1
5.	α_1: await $b \neq 0$		
6.	α_2: a:= a+3	5	1
	$\vdots$ no deadlock		

Definition of deadlock

Because of the **await** statements, a process may be delayed or blocked at an **await**, until its condition B is true.

Suppose a statement S is being executed. S is blocked if it has not terminated, but no progress in its execution is possible, because it (or all of its subprocesses that have not yet terminated) are delayed at an **await**. Blocking by itself is harmless; processes may become blocked and unblocked many times during execution. However, if the whole program becomes blocked, this is serious because it can never be unblocked and thus the program can not terminate.
Execution of a program ends in deadlock if it is blocked.

A program S with proof $\{P\}\, S\, \{Q\}$ is free from deadlock if no execution of S which begins with $P\ true$ ends in deadlock. We wish to derive sufficient conditions under which a program is free from deadlock.

The general form of the program containing parbegin statements

Note that in the literature both the two following notions are used for denoting a parralel program with statement $S_1 \ldots S_n$ to be executed in parallel:
cobegin $S_1 \| \ldots \| S_n$ **coend**
parbegin $S_1 \| \ldots \| S_n$ **parend**

The general form of the program S that contains parbegin statements looks something like this:

S :

$\vdots$
A_1 : **await** B_1 then S_1 ta;

$\vdots$

A_i : **await** B_i then S_i ta;

$\vdots$

T_1 : **parbegin** $S_1^1 \| \ldots \| S_{n_1}^1$ **parend**

$\vdots$

A_j : **await** B_j then S_j ta;

$\vdots$

T_n : **parbegin** $S_1^n \| \ldots \| S_{n_n}^n$ **parend**

$\vdots$

A_m : **await** B_m then S_m ta;

$\vdots$

We assume that the program S has m number of await statements that are not within a parbegin statement. We also assume that S has n number of parbegin statements, where the kth parbegin statement is given the following form:
T_k: **parbegin** $S_1^k \| \ldots \| S_{n_k}^k$ **parend**
We can assume that the component programs (S_1^1,...,$S_{n_1}^1$,...,S_1^n,...,$S_{n_n}^n$) have the same structure, they may contain await statements and nesting parbegin satements is also allowed.

Now, if we consider deadlock, the potential deadlock situations of the above program are the followings:

- The sequential program S is delayed at one of its await statement that is not whithin a parbegin statement. Waiting at an await statement A_j ($j \in [1..m]$) occurs if the precondition of the await statment (let us denote it by $pre(A_j)$) holds, but its guard B_j is $false$.
- One of the parallel processes of the sequential program S is blocked. Let us denote that process by T_k. Now, since T_k is blocked, then each of its processes S_i^k ($i \in [1..n_k]$) has terminated or is blocked, and moreover, at least one process of them is blocked.

Theorem

Let S be a statement with proof $\{P\}\, S\, \{Q\}$. Let the **await**s of S which do not occur within **parbegin** of S denoted by A_j.

A_j: **await** B **then** S
Let the **parbegin**s of S which do not occur whithin other **parbegin**s of S be
T_k: **parbegin** $S_1^k \| \ldots \| S_{n_k}^k$ **parend**

Let us define conditions $D(S)$ and $D1(T_k)$. Then $D(S) = FALSE$ implies that no execution of S can be blocked. Hence, program S is free from deadlock.

- $$D(S) = \left[\bigvee_{j=1}^{m} (pre(A_j) \wedge \neg B_j)\right] \vee \left[\bigvee_{k=1}^{n} D1(T_k)\right]$$

  This formula expresses that program S is delayed if it is waiting at any await statement (outside of the parbegin statements) or within any parbegin statement.

- $$D1(T_k) = \left[\bigwedge_{j=1}^{n_k} (post(S_i^k) \vee D(S_i^k))\right] \wedge \left[\bigvee_{i=1}^{n_k} D(S_i^k)\right]$$

  Program S is delayed whithin parbegin statement T_k if it has at least one delayed component S_i^k when every component is completed or delayed.

Special case of deadlock

Let us consider the following parallel program that contains two components:

parbegin $S_1 \| S_2$ **parend**

In general we may have await statements within the component programs S_1 and S_2, but in this case we have no await statements outside the parbegin statement. The formula of deadlock reduces to

$$D(S) = \bigwedge_{j=1}^{2} (post(S_i) \vee D(S_i)) \wedge \bigvee_{i=1}^{2} D(S_i) = \Big(\big(D(S_1) \vee post(S_1)\big) \wedge \big(D(S_2) \vee post(S_2)\big)\Big) \wedge \Big(D(S_1) \vee D(S_2)\Big)$$

The first part, the formula $\Big(\big(D(S_1) \vee post(S_1)\big) \wedge \big(D(S_2) \vee post(S_2)\big)\Big)$ describes four possible cases:

- $D(S_1) \wedge D(S_2)$
- $D(S_1) \wedge post(S_2)$
- $post(S_1) \wedge D(S_2)$
- $post(S_1) \wedge post(S_2)$

The last one is not a potential deadlock situation and is eliminated by $(D(S_1) \vee D(S_2))$. If $post(S_1) \wedge post(S_2)$ holds, it means that both S_1 and S_2 terminated normally and reached their postcondition. This is why we had to add in the formula above that at least one of the component programs are in deadlock.

Summarization: Deadlock may occur only in the first 3 cases. The deadlock formula can be simplified to $\big(D(S_1) \wedge D(S_2)\big) \vee \big(D(S_1) \wedge post(S_2)\big) \vee \big(D(S_2) \wedge post(S_1)\big)$
If we prove that this formula is $FALSE$, then we exclude the 3 potential deadlock cases, we can be sure that our program is free from deadlock.