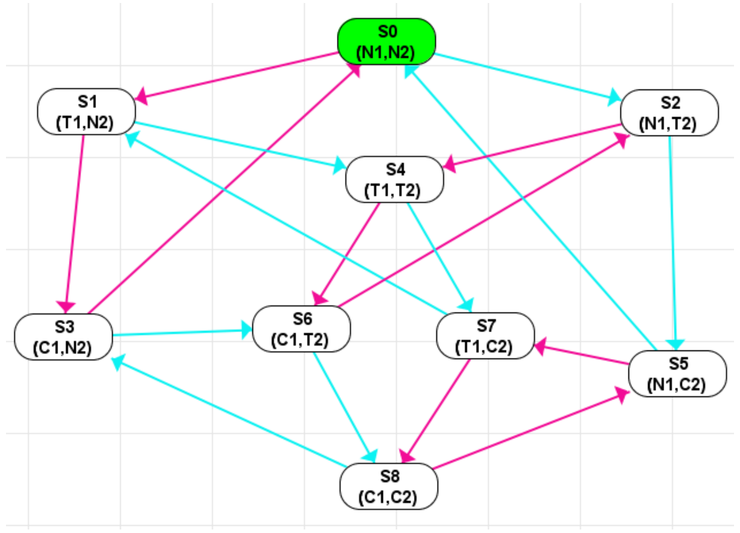
Ex 1: Fix Mutex to satisfy liveness

Definitions:

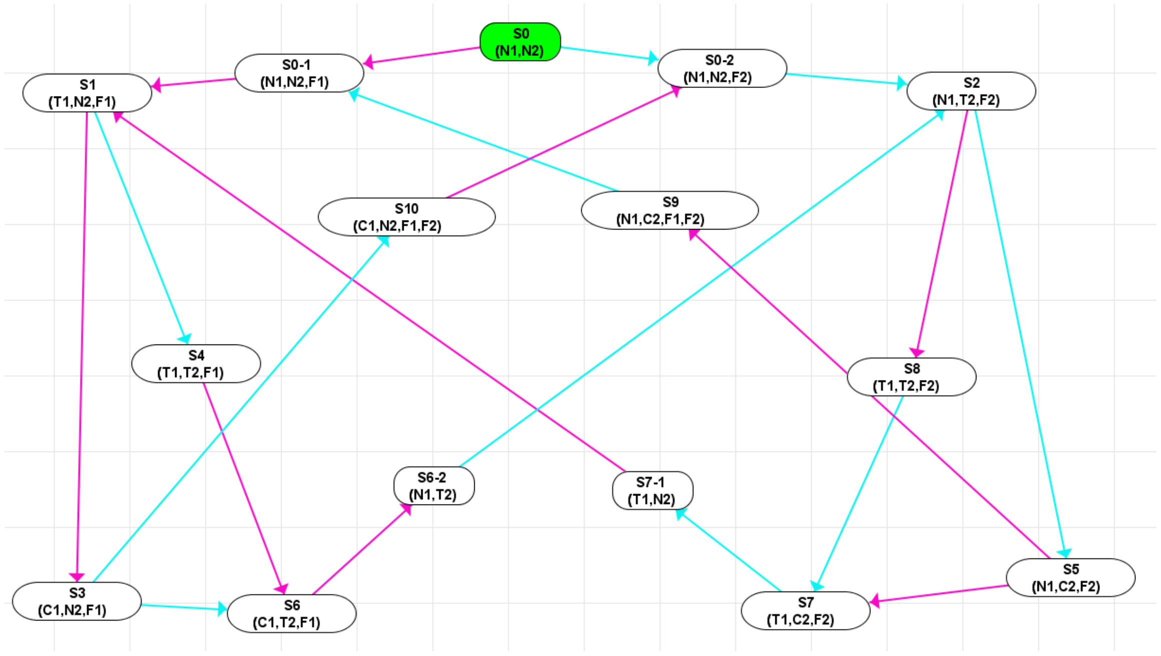
- Ni = Process i is neutral
- Ti = Process i is trying
- Ci = Process i is in the critical section
- Fi = Process i has priority to enter critical section
- i = { 1, 2}

Construction:

1. We started by using “Mutex2” model that came with Eshmun.



The end result was the following:



2. S8 was removed since we have a precondition NOT(C1 AND C2)
   Remark: we later introduce another state with the same naming, but different purposes.

3. S4 was split into 2 states S4(T1,T2,F1) and S8(T1,T2,F2) in order to have fairness to decide who enters the critical section. F1 and F2 were used in several states for this purpose.

4. In order to achieve liveness we make sure the following scenarios are true.

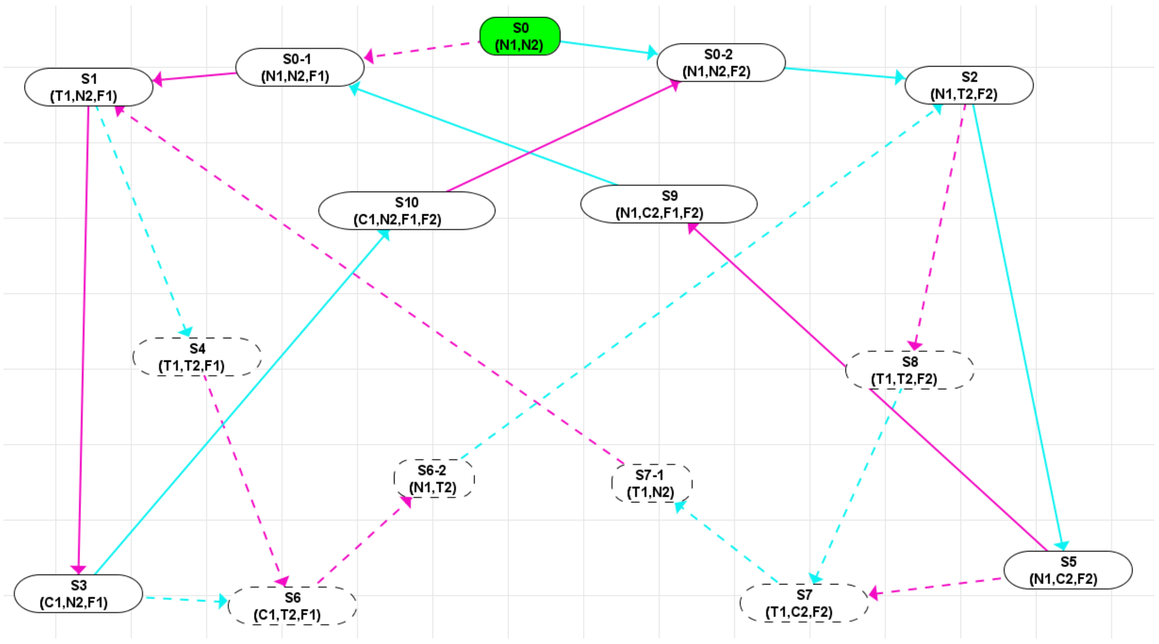
   a. If P1 asks to be in the critical section and P2 doesn't. Then P1 will enter the critical section and the next process to enter the critical section will be P2 when it requests to do so. (same if P2 asks and P1 doesn't)
      Example: S0 -> S0-1 -> S1 -> S3 - > S10 -> S0-2

   b. If both P1 and P2 ask to be in the critical section, then the one who asked first will enter, then the other.
      Example: S0 -> S0-1 -> S1 -> S4 -> S6 -> S6-2 -> S2 -> S5 -> S9 -> S0-1

5. S0-1 and S0-2 serve respectively as the states were P1 and P2 have priority to enter next into critical section upon a request.

6. S10, S9, S7-1, S6-2 serve as middle men in order to get to the next state, since we can't go directly because we don't have shared memory.

After the model has been repaired with the liveness property: (AG(T1 => (AF(C1)))) & (AG(T2 => (AF(C2))))

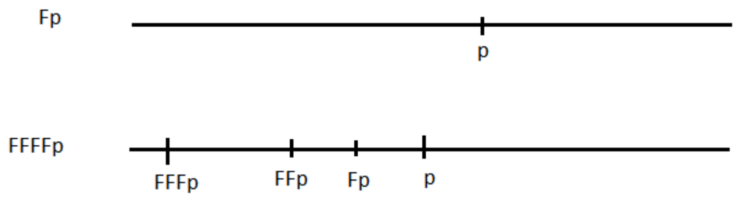


Ex 2: Simplify f = O1O2O3……….. On p where each Oi is either F or G

We will prove that f can be simplified into the following set S = { Fp, Gp, FGp, GFp }. First we will introduce some proofs that we need, then we will give the procedure.

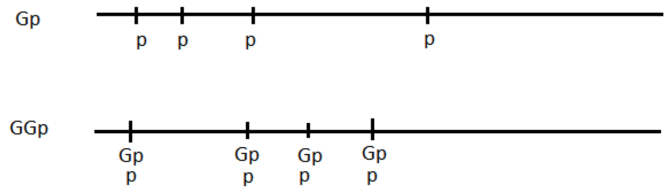
Prerequisite proofs:

1. FFF…..Fp is equivalent to Fp



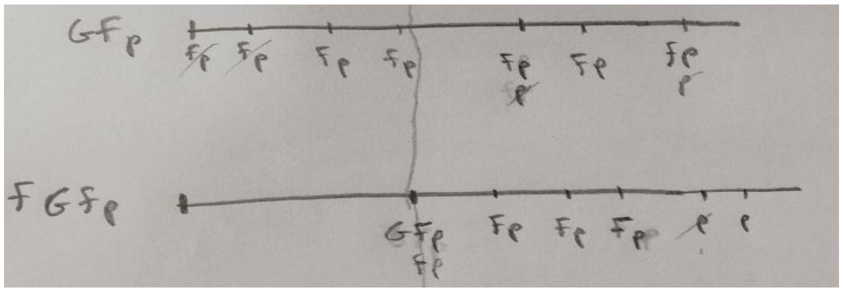
If we have FF….n times…Fp, this means that we finally will see FF….(n-1)times..Fp and so on until we finally see Fp which eventually gives p.
If we have p, then going backwards it's possible to have Fp and before that FFp and so on until we have FF….n times…Fp

2. GGG….Gp is equivalent to Gp



GGp means that we have on every state GP, but this means that on each state we will have p. In fact the very first Gp is enough to get this.
Similarly if have GGGp then we will have GGp on each state which gives us Gp on each state which gives us p on each state.
Thus by induction we can prove that GG….Gp is equivalent to Gp.

3. FGFp is equivalent to GFp



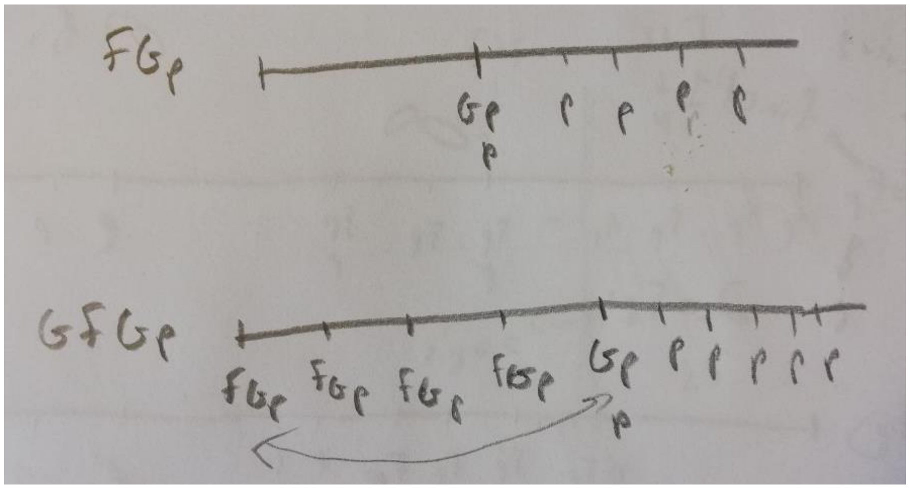
GFp: we should get infinitely many p's to satisfy the Fp's
FGFp: at a certain point we should we should get infinitely many p's to satisfy the Fp

I assume that each Fp will require a p. For example for each request T of a process it should get a C.

GFp => FGFp since we have infinitely man Fp's, the few "extra" Fp's that GFp produces are just a constant in comparison. That means GFp won't produce more p's eventually than FGFp

FGFp => GFp since we have FGFp has infinitely many p's eventually, so we can cross them with the "extra" Fps from GFp and we will have still have infinitely man p's to cross the infinitely many Fp's.

4. GFGp is equivalent to FGp



FGp: at a certain points it's we're always going to have continuously infinitely many p's.
GFGp: eventually each FGp is going to get a Gp

FGp => GFGp since at a certain points it's we're always going to have continuously infinitely many p's. So the property GFGp is going to be satisfied since if at a certain point we're going to have Gp, then FGp can be satisfied no matter where you put it, even if on every state i.e. GFGp.

GFGp => FGp since if at every state we have FGp, then at a certain point they are going to have to be satisfied. Actually, once the first FGp is satisfied as shown in the image above, then we get GFGp => FGp. Since the rest of the FGp's are just going to be 'true'

Procedure:

I. Using proof 1, reduce all consecutive F's to just one F
II. Using proof 2, reduce all consecutive G's to just one G
III. Now we should have 4 scenarios :
    a. Fp (The path was just an Fp or a sequence of Fs followed by p) then accept and stop procedure.

b. Gp (The path was just a Gp or a sequence of Gs followed by p) then accept and stop procedure
c. (FG)+ (F)? p i.e. a sequence of FGs and possibly an F at the end followed by p
d. (GF)+ (G)? p i.e. a sequence of GFs and possibly a G at the end followed by p

So far S = {Fp,Gp}

In case of scenario c, we have FGFG….FG(F)?p Using proof 3, the first 3 characters can be reduced to GFG…FG(F)?p Using proof 4, the new first 3 characters can be reduced again. And we keep doing this until we finally either reach FG(F)?p which can give us either FGp or FGFp and the latter can be reduced to GFp.

So far S = {Fp, Gp, FGp, GFp}

In case of scenario d, similarly we will either reach GFp or GFGp and the latter is equivalent to FGp.

S remains equal to {Fp, Gp, FGp, GFp}