$\mathrm{db}_{\mathrm{sol3}}$

Andrea Crotti (#299466), Can Liu (#286105), Sebastian Roidl (#281941)

10 Novembre 2009

Exercise 3

`DEADLINE:` *2009-11-11 Mer*

1) Recoverability

1. Prove the following

a) Every schedule belonging to avoid cascading abort (ACA) is also recoverable (RC)

Supposing that s fulfills the requirements for **ACA** and given that:

- a commit action must be always the last of the transaction
- reads x from t_j in s $\rightarrow c_j < r_i(x)$

We directly have that also $c_j < c_i$, which is the requirement for RC. So $s \in ACA \rightarrow s \in RC$

b) Each schedule that is strict is also in ACA

2. Test if in RC, ACA, ST

a) s1 = r3(y) r1(x) w1(x) r2(x) c1 w3(y) w2(x) c3 c2

- **RC** Yes, t2 reads from t1 and c1 < c2
- **ACA** No, t2 reads from t1 but c1 > r2(x). In case of abort of the transaction t1 we could still have problems here. So it's also not **ST**

b) s2 = r2(x) r3(y) w2(x) c2 r3(x) w3(x) c3 r1(x) w1(x) c1

- **RC** Yes, t3 reads from t2 and c2 $<$ c3, t1 reads from t3 and c3 $<$ c1
- **ACA** Yes, t3 reads from t2 and c2 $<$ r3(x), t1 reads from t3 and c3 $<$ r1(x)
- **ST** Yes, we can check easily that after every write of $w_i(x)$ we have a c_i, there are no writes or reads of x between last write and commit.

c) s3 = r1(x) w1(x) r3(x) w3(x) c3 r2(x) w2(x) c2 a1

- **RC** No, because t3 reads from t1 but there is not a commit c1. So as logical consequence s3 it's not also **ACA** or **ST**

2) 2PL / S2PL / Deadlock handling

1. Prove that schedules produced by 2PL are conflict serializable.

The complete proof of this is given at page 70. In other words we

- take the rules for the application of locking
- see that if two transactions are in conflict we must negate one of the constraint of 2PL
- see that the graph generated by 2PL is acyclic

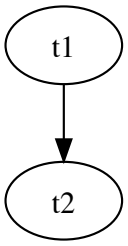
2. Given

- s1 = w1(x) r2(y) r1(x) c1 r2(x) w2(y) c2
- s2 = w1(x) r2(y) r1(x) c1 r2(x) w2(y) a2
- s3 = r1(x) r2(x) w3(x) w4(x) w1(x) c1 w2(x) c2 c3 c4

a) Compute the conflict graphs of the schedules above.

- s1

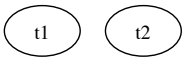
```
digraph s1 {
t1 -> t2;
}
```



Which is an acyclic graph, so $s1 \in CSR$

- s2

```
digraph s1 {
t1;
t2;
}
```

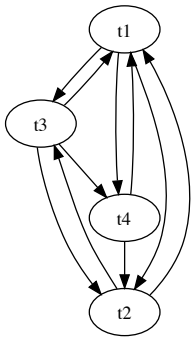


Would be the same as s1, but there's an abort action which deletes the conflict between t1 and t2, so $s2 \in CSR$

- s3

```
digraph s3 {
t1 -> t3;
t1 -> t2;
t1 -> t4;
t2 -> t3;
t2 -> t1;
t3 -> t1;
```

```
t3 -> t2;
t3 -> t4;
t4 -> t2;
t4 -> t1;
}
```



This is a cyclic graph, so $s3 \notin CSR$

b) For each input schedule write down the corresponding schedule indicating the necessary locking (rl/wl) and unlocking (ru/wu) operations.

Write down the resulting output schedules for 2PL (transactions must unlock resources as soon as possible) and S2PL. In case of a deadlock the transaction with the lowest index is aborted. Once aborted, transactions are restarted anew at the end of the original schedule (abort- restart)

- s1
  - **2PL:** wl1(x) w1(x) wu1(x) r1(x) c1 rl2(y) wl2(y) r2(y) ru2(y) r2(x) wu2(y) c2
  - **S2PL:**
    We just need to move wl1(x) w1(x) r1(x) wu1(x) c1 rl2(y) wl2(y) r2(y) r2(x) ru2(y) wu2(y) c2
- s2

  - **2PL:** wl1(x) w1(x) r1(x) wu1(x) c1
  - **S2PL:** wl1(x) w1(x) r1(x) wu1(x) c1

- s3

  - **2PL:** There are not possible scheduling, the conflict graph is not acyclic.