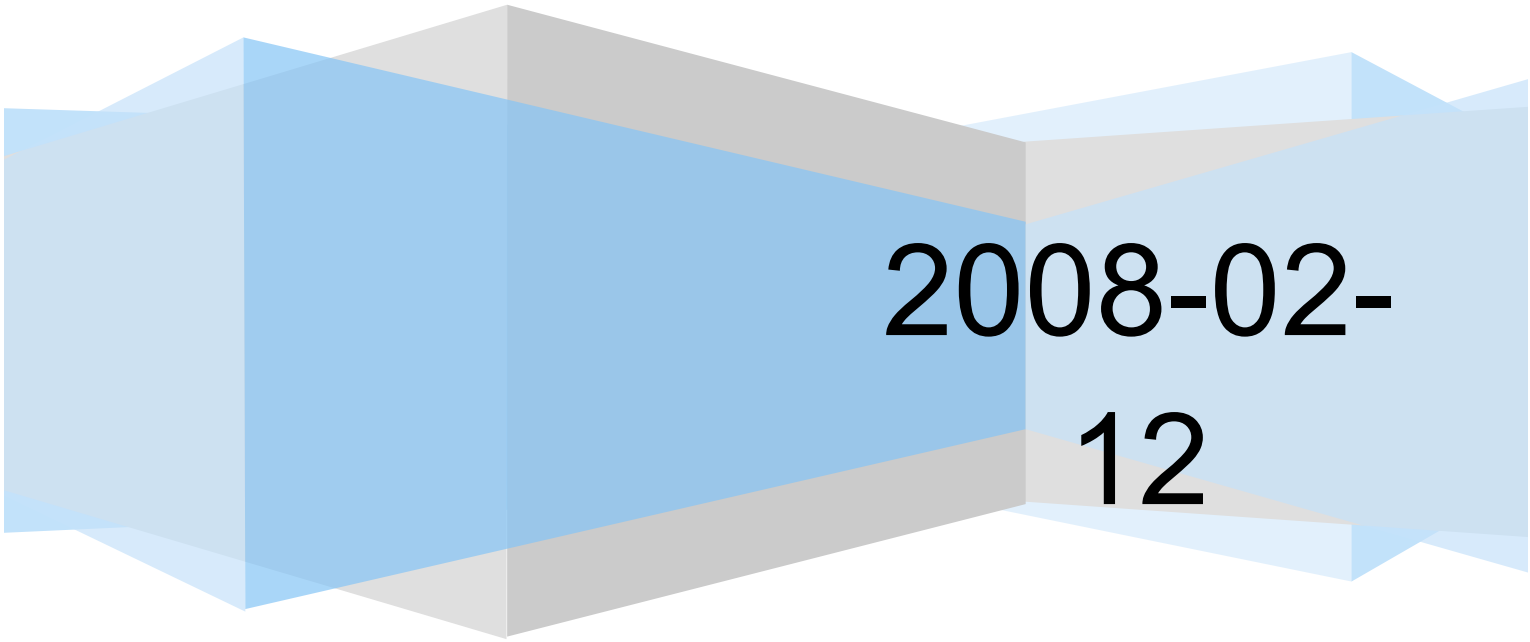
Università “La Sapienza” – Laurea Specialistica in Ingegneria Informatica

Gestione della concorrenza

Modellazione e verifica formale dei criteri di correttezza di protocolli per la gestione della concorrenza

Francesco Minicucci



2008-02-12

Sommario

Modellazione e verifica formale dei criteri di correttezza di algoritmi per la gestione delle transazioni in un dbms.

Introduzione

La problematica relativa alla gestione della concorrenza in un sistema transazionale ha rivestito un ruolo sempre più importante negli ultimi anni. Il controllo di concorrenza è una delle funzioni principali di un dbms. La parola transazione può avere diverse accezioni, nel nostro caso la intendiamo come una sequenza di operazioni che modellano l'esecuzione di una procedura software. Le operazioni principali contenute in una transazione sono: begin, read(a), write(a), end, commit e rollback. I "comandi" begin e end indicano l'inizio e la fine di una transazione, read e write leggono e scrivono da/su una risorsa condivisa, infine commit e rollback rendono definitivo oppure annullano le operazioni effettuate sulla base di dati. Il dbms deve garantire che l'esecuzione concorrente di transazioni avvenga senza interferenze in caso di accessi agli stessi dati. In generale, un insieme di proprietà fondamentali che si desidera il dbms garantisse in caso di accesso concorrente e/o in caso di guasto è il seguente:

- Atomicità: le transazioni vanno eseguite per intero oppure per niente.
- Consistenza: il significato più indicato per questa proprietà tradotta letteralmente dall'ingleseconsistency è forse coerenza. In tutti i

casi, le esecuzioni delle transazioni devono preservare lo stato della base di dati.

- Isolamento: Ogni transazione si deve “isolare” dalle altre transazioni durante la sua esecuzione.
- Durabilità: l’effetto di ogni transazione deve durare nel tempo.

Queste proprietà vengono chiamate ACID. In questo contesto un concetto essenziale è dato dal significato di schedule. Lo schedule è una sequenza di operazioni sulla base di dati appartenenti a varie transazioni. Lo scheduler del dbms prende in input questo schedule e deve garantire le proprietà suddette. Per finire la trattazione di questo argomento, diamo i concetti di schedule seriali e serializzabili. Gli schedule seriali sono sequenze ordinate di transazioni. Mi spiego meglio: date due transazioniT1 = {begin, read(A), write(A), commit } e T2= {begin , read(A), write(A), commit }, uno schedule seriale è uno schedule che prevede prima tutte le operazioni della transazione T1 e poi tutte quelle di T2 oppure il viceversa. Uno schedule serializzabile invece è uno schedule il cui effetto è lo stesso di un qualche altro schedule seriale sulla stessa istanza di basi di dati.

A questo punto, dato questo insieme di definizioni,possiamo entrare a parlareoperativamente di gestione della concorrenza.

Prima di fare questo volevo chiarire gli obiettivi della mia tesina. L’obiettivo principale sarà quello di modellare alcuni protocolli che definirò nel seguito e verificare importanti proprietà tra cui quelle citate, ACIDe poi di studiare i casi di stallo.

Controllo di concorrenza mediante lock

Introduciamo altre due operazioni che le transazioni possono eseguire sulle istanze di basi di dati: lock ed unlock. Quando una transazione esegue un lock sull'elemento A sta dichiarando di dover utilizzare l'elemento e se ottiene il lock questo non può essere utilizzato da nessun'altra transazione. L'unlock è ovviamente il contrario, la transazione dichiara di non dover più utilizzare l'elemento e lo mette nuovamente a disposizione. Quindi in sostanza una transazione per dover operare su di un elemento deve effettuare un'operazione di lock , quando avrà finito lo rilascerà effettuando un unlock. Inoltre i lock possono essere condivisi oppure esclusivi, per il momento consideriamo i secondi e nel seguito allargheremo questa ipotesi. Le transazioni che seguono il protocollo dei lock devono essere ben formate quindi, come sopra, ogni operazione su un elemento deve essere compresa nella coppia (lock, unlock) e gli schedule devono essere legali, cioè nessuna transazione esegue un lock su un elemento già "lockato" da un'altra transazione. Queste due assunzioni le possiamo rilassare essendo compito dello scheduler inserire in maniera opportuna le operazioni di lock ed unlock. Infine dobbiamo facciamo un'importante assunzione: ogni transazione legge e scrive al massimo una volta un dato. Se non così non fosse si potrebbe spezzare la transazione in due transazioni equivalenti.

Detto questo possiamo definire il protocollo Two Phase Locking (2PL).

Uno schedule segue il protocollo 2PL se "in ogni transazione Ti presente nello schedule tutte le operazioni di lock di Ti precedono tutte le operazione di unlock di Ti", (definizione presa dalle dispense del prof.

Lenzerini sulla gestione della concorrenza). A questo punto possiamo valutare come vengono valutati gli schedule in entrata allo scheduler, quindi come questo si comporti. Ricordiamo che stiamo operando solo su lock esclusivi.

Dato in input uno schedule S e n transazioni Ti per i={1..n}:

1. Per ogni operazione com in S:
    a. If (com = lock su A di Ti)
        i. If (A non è stato lockato da nessun altra Tj & com non è preceduto da nessun unlock di Ti)
            1. Esegui il lock su A di Ti;or
        ii. Else if (A è stato già lockato da Tj & com non è preceduto da nessun unlock di Ti)
            1. Sospendi la transazione Ti;
        iii. Else
            1. Lo schedule non segue il protocollo 2PL;
    b. If (com = read(A) di Ti | com = write(A) di Ti)
        i. Esegui com; *Essendo compito dello scheduler inserire i comandi di lock e unlock non può presentarsi il caso di read o write non lockate precedentemente
    c. If (com = unlock su A di Ti)
        i. If (esistono Tj sospese per lock su A)
            1. Esegui unlock;
            2. Riattiva Tj;
        ii. Else
            1. Esegui Unlock;
    d. If (com = commit di Ti)

i. Termina Ti;

Una spiegazione è d'obbligo.

1) Ho usato la parola lockato per indicare un'operazione di lock in precedenza per comodità pur non essendo grammaticalmente corretta.
2) Per ogni operazione contenuta in S l'algoritmo esegue un controllo e blocca le transazioni che stanno richiedendo un lock su elementi su cui è già stato effettuato il lock. Ricordiamo che non può capitare che una transazione esegua due volte il lock sullo stesso elemento
3) Osserviamo che alcune condizioni si possono rilassare. Si può programmare lo scheduler in modo tale da inserire opportunamente i comandi lock e unlock. Quindi lo scheduler oltre a garantire che ogni transazione sia ben formata e che lo schedule sia legale, può garantire che segua il protocollo 2PL. Questo significa che alcune condizioni le possiamo evitare come 1.a.III.
4) Possiamo scrivere una nuova versione dell'algoritmo considerando solo i comandi di lock unlock e commit.

Dato in input uno schedule S e n transazioni Ti:

1. Per ogni operazione com su S:
    a. If (com = lock su A di Ti)
        i. If (A è già stato lockato da Tj)
            1. Blocca Ti;
        ii. Else

1. Effettua com;

b. If (com = unlock su A di Ti)
   i. If (Tj stava in attesa di unlock su A)
      1. Esegui com:
      2. Sblocca Tj;
   ii. Else
      1. Esegui com;
c. If (com = commit di Ti)
   i. Termina Ti;

Questa versione decisamente più leggibile della precedente sarà presa in considerazione per la verifica di alcune proprietà che diremo nel seguito come la presenza di situazioni pericolose che possono portare allo stallo.

Diagramma UML degli stati e delle transizioni

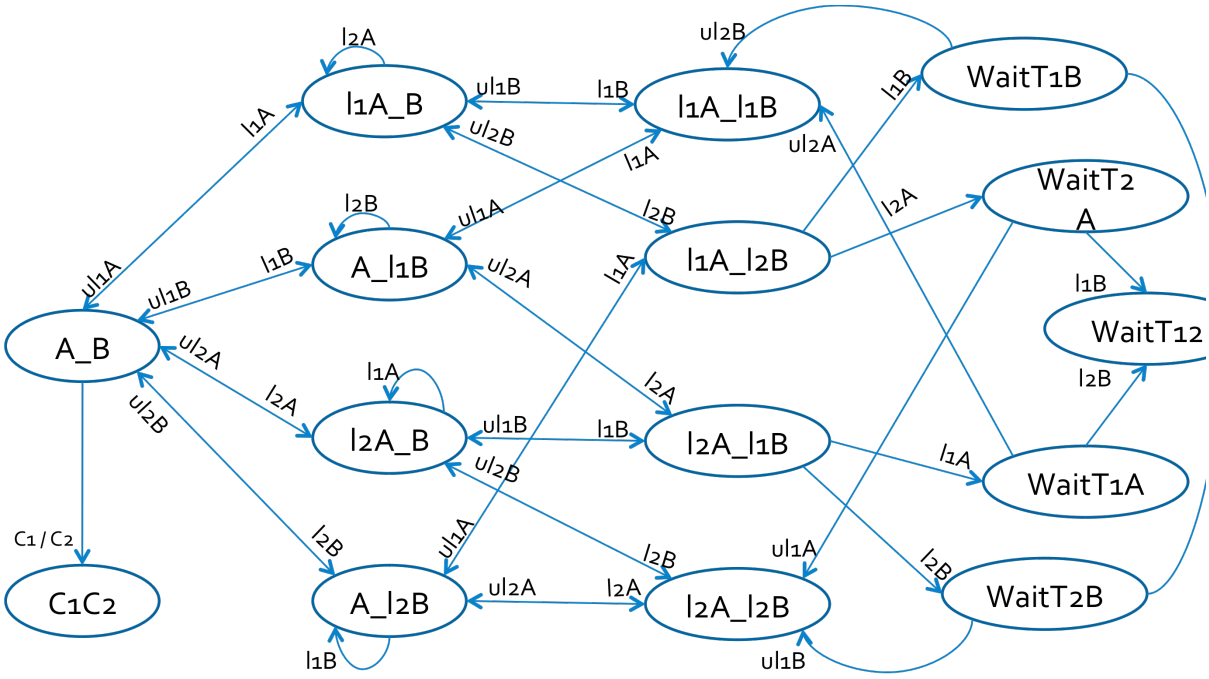
La difficoltà maggiore incontrata nel costruire tale diagramma è stata quella dell'esplosione degli stati con l'incrementare delle transizioni e degli elementi considerati. Nel modello riportato di seguito ci si è limitati a porre l'attenzione a due transazioni concorrenti che possono accedere a due elementi. Tale assunzione è stata fatta per motivi di limitata difficoltà nella rappresentazione e perché tutte le proprietà e le situazioni "pericolose" che osserveremo nel seguito valgono e si possono facilmente estendere per il caso a n transazioni e m elementi.
Gli elementi che consideriamo saranno denominati A e B, e le transazioni T1 e T2. Gli stati sono in totale 9. L'insieme I degli stati è = {

A_B , l1A_B, A_l2B, l2A_B, A_l2B, l1A_l1B, l1A_l2B, l2A_l1B, l2A_l2B, C1C2 }. Facciamo chiarezza sul significato delle sigle. Il carattere underscore "_" sta a separare i due elementi, inoltre sui due elementi può essere stato fatto un lock oppure no. Nel primo caso mettiamo davanti l'elemento la lettera "l" e il numero identificativo della transazione (1 o 2) e, nel secondo caso, non si mette nulla. Quindi, per esempio, lo stato l1A_B sta a significare che la transazione T1 ha effettuato il lock su A mentre non è, al momento, stato effettuato il lock su B (potrebbe essere stato fatto seguito da un unlock).L'insieme E degli eventi è dato da: { l1A, l1B, l2A, l2B, c1c2 }. I primi quattro hanno il significato ovvio mentre per l'ultimo c1c2, senza perdita di generalità, abbiamo considerato che le due transazioni effettuassero il commit alla fine dello schedule ed abbiamo assunto per comodità che questo fosse l'evento c1c2. Ci sono molte cose da considerare per ben interpretare il diagramma UML. Prima di tutto ci sono una serie di requisiti e di condizioni che dobbiamo discutere e cercare di imporre per dare un significato corretto al modello finale. I requisiti in italiano sono i seguenti:

- Ogni transazione una volta effettuato il lock su una variabile deve effettuare in un tempo futuro un unlock su tale variabile. Quindi ad esempio se l'evento è l1A, a questo seguirà prima o poi l'evento ul1A.Questo per garantire che le transazioni sia ben formate.
- Ogni transazione effettua al massimo una volta l'operazione di lock su una variabile. Questo in base all'assunzione fatta in precedenza che per ogni transazione abbiamo al massimo un'operazione di read e di write su un elemento.

- Ogni transazione una volta effettuata un'operazione di unlock non può più effettuare un'operazione di lock su qualsiasi elemento. Questo per garantire il protocollo 2PL.
- Se ho effettuato un'operazione di lock su un elemento ed un'altra transazione chiede di fare il lock su tale elemento ciclo sullo stesso stato ed aspetto che l'elemento venga rilasciato. Questo per come è fatto il protocollo.
- Se sto in uno stato con un elemento non lockato e l'evento è la richiesta di lock da parte di una transazione il prossimo stato prevede il lock su tale elemento della stessa transazione.
- Viceversa, in uno stato con un elemento su cui è stato effettuato un lock e l'evento è l'unlock sull'elemento da parte della transazione che aveva ottenuto il lock, il prossimo stato prevede l'elemento "libero", senza lock. Chiaramente, per le assunzione fatte sopra, la stessa transazione non più chiedere il lock su nessun elemento.

Nella pagina seguente è riportato il diagramma UML degli stati e delle transazioni.



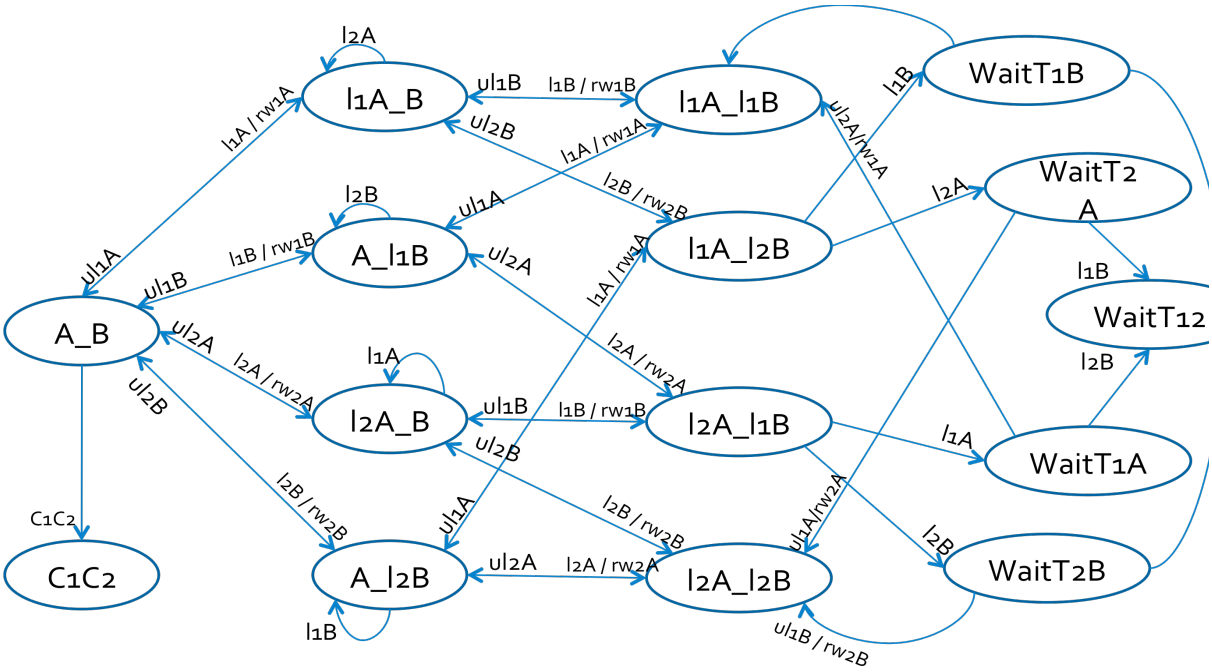
A questo punto ci si potrebbe chiedere dove sono finite le operazioni di read e write. Per quanto detto ogni operazione di read e write deve essere contenuta tra una coppia di operazioni (lock ….. unlock). Quindi possiamo immaginare che negli stati in cui la transazione Ti ha chiesto ed ottenuto il lock su un elemento A, si effettuino le operazioni di read e write relative ad A da parte di Ti. Quindi ai nostri fini mi sembra corretto concentrarmi sulle operazioni di lock, unlock e commit considerando poi nel seguito le operazioni read/write come azioni conseguenti.

È interessante notare e studiare possibili schedule che potrebbero portare a situazioni di stallo. In particolare ci si potrebbe chiedere se partendo dallo stato A_B con evento diverso da null e diverso da c1c2 si può sempre arrivare nello stato C1C2.

Come si vede dalla figura ho dovuto aggiungere degli stati di attesa. Si noti bene come questi stati sono successivi agli stati in cui ho i due elementi su cui è stato effettuato un lock. Mi spiego meglio, ho una

situazione di stallo quando i due elementi A e B sono stati lockati da T1 e T2 (o viceversa) e ho due richieste di lock di T2 e T1 (e rispettivamente viceversa) su A e B. Ho ritenuto necessario aggiungere degli stati di attesa. Inoltre quando ho una richiesta di lock da parte di Ti su un elemento su cui è stato già effettuato un lock, mettendo in attesa Ti è come se si stessero spostando tutte le operazione di Ti in avanti. Ad esempio,se sto nello stato l1A_l2B e l'evento è l1B passo allo stato di attesa di T1: WaitT1B e quando T2 effettuerà ul2B andrò nello stato l1A_l1B. Questo presuppone che avendo in sospeso l1B se si effettua ul2B, l'elemento B viene automaticamente lockato da T1. Un'ultima considerazione sugli stati di stallo. Il sistema entra in stallo quando le due transazioni detengono il lock su una risorsa e chiedono il lock dell'altra risorsa. Più precisamente se T1 e T2 hanno effettuato il lock su A e su B si entra in stallo quando T1 e T2 chiederanno il lock su B e su A rispettivamente senza effettuare l'unlock precedentemente. Notiamo quindi la differenza tra stati di attesa e di stallo, uno stato d'attesa non è per forza uno stato di stallo mentre uno stato di stallo è anche uno stato d'attesa.

Nell'implementazione in Nusmv ho tenuto conto anche delle operazioni di lettura e scrittura su un elemento. Per quanto detto in precedenza posso considerare queste operazioni come azioni. Ad esempio se sto nello stato A_B e la transazione T1 richiede il lock su A passaallo stato l1A_B. In quest'ultimo stato presumibilmente ci sarà un'operazione di lettura e/o scrittura su A. Le azioni in Nusmv le ho chiamate rw1A, rw2A, rw1B, rw2B con ovvi significati. Riporto qui di seguito il diagramma UML degli stati e delle transizioni ottenuto considerando le azioni di read/write:



Implementazione in Nusmv

Nell'implementazione di Nusmv ho usato come base del sistema il diagramma degli stati e della transazioni. Ho utilizzato come variabili di base quelle spiegate in precedenza, ossia:

stato: {A_B, C1C2, l1A_B, A_l1B, l2A_B, A_l2B, l1A_l1B, l1A_l2B, l2A_l1B, l2A_l2B, WT1B, WT2A, WT1A, WT2B, WT1T2};

evento: {l1A, l1B, l2A, l2B, ul1A, ul1B, ul2A, ul2B, c1c2, null};

azione: {rw1A, rw1B, rw2A, rw2B, null};

Quindi sono andato a definire il modulo responsabile del funzionamento del protocollo 2PL e il modulo main. Nel seguito ho utilizzato una serie di condizioni che volevo far rispettare al diagramma UML, quindi una serie di vincoli che devo imporre. Li descrivo di seguito:

- Ogni transazione una volta effettuato il lock su una variabile deve effettuare in un tempo futuro un unlock su tale variabile

(G(p.evento = l1A -> F p.evento = ul1A) &
G(p.evento = l1B -> F p.evento = ul1B) &
G(p.evento = l2A -> F p.evento = ul2A) &
G(p.evento = l2B -> F p.evento = ul2B) &

- Ogni transazione effettua al massimo una volta l'operazione di lettura scrittura su una variabile

(G(p.azione = rw1A -> X G p.azione != rw1A) &
G(p.azione = rw1B -> X G p.azione != rw1B) &
G(p.azione = rw2A -> X G p.azione != rw2A) &
G(p.azione = rw2B -> X G p.azione != rw2B) &

- Evitiamo gli stalli dovuti a richieste continue di lock:

G(p.stato = l1A_B & p.evento = l2A -> X G (p.stato = l1A_B -> (p.evento != l2A & p.evento != l2B))) &
G(p.stato = A_l1B & p.evento = l2B -> X G (p.stato = A_l1B -> (p.evento != l2B & p.evento != l2A))) &
G(p.stato = l2A_B & p.evento = l1A -> X G (p.stato = l2A_B -> (p.evento != l2B & p.evento != l2A))) &
G(p.stato = A_l2B & p.evento = l1B -> X G (p.stato = A_l2B -> (p.evento != l2B & p.evento != l2A))) &
G(p.stato = l1A_l1B & (p.evento = l2A | p.evento = l2B) -> X G (p.stato = l1A_l1B -> (p.evento != l2A & p.evento != l2B))) &
G(p.stato = l2A_l2B & (p.evento = l1A | p.evento = l1B) -> X G (p.stato = l2A_l2B -> (p.evento != l1A & p.evento != l1B))) &

- Ogni transazione una volta effettuata un'operazione di unlock non può più effettuare un'operazione di lock su qualsiasi elemento:

G(p.evento = ul1A | p.evento = ul1B -> X G (p.evento != l1A & p.evento != l1B))
&
G(p.evento = ul2A | p.evento = ul2B -> X G (p.evento != l2A & p.evento != l2B)
))

Verifica dello stallo

Nell'implementazione ottenuta non possiamo provare lo stallo perché per come ho progettato la sezione LTLSPEC dobbiamo trovare un altro metodo. Questo perché per far rispettare il diagramma UML delle condizioni ho usato il metodo PRE -> ASS, dove PRE è l'insieme di formule che voglio garantire mentre ASS è l'asserzione che voglio inferire. La situazione di stallo mi porta un'attesa indefinita da parte delle transazioni, ma se queste attendono l'unlock degli elementi significa che queste non effettueranno mai unlock! Quindi le PRE vanno a cadere e la formula è sempre vera. Devo escogitare un altro metodo che porta ad una situazione di stallo. A pensarci è normale perché abbiamo imposto delle precondizioni troppo forti (PRE) che di per se stesse evitano ogni situazione di stallo. Passiamo ora a PRE_ .Sono riuscito a provare lo stallo rilassando una condizione, quella che una volta che una transazione richiede il lock prima o poi effettuerà un unlock sulla stessa variabile. Questo perché se entro in uno stallo significa che le due transazioni restano in attesa e che entrambi hanno già effettuato il lock su una variabile, quindi non ci sarà mai un unlock da parte loro proprio per la definizione di stallo. Se avessi imposto anche questa condizione il sistema Nusmv non sarebbe mai potuto entrare in stallo o meglio se

entrava in stallo non rispettava le condizioni PRE_ e quindi l'implicazione è sempre vera. Comunque rilassando solo questa condizione, abbiamo ottenuto che non è vero che il sistema partendo sempre da A_B (A e B quindi non lockati) arriva sempre ad uno stato di commit C1C2. Fornisco l'output che ho ottenuto:

-- specification (((((((((((G (p.azione = rw1A -> X (G p.azione != rw1A)) &G (p.azione = rw1B -> X (G p.azione != rw1B))) & G (p.azione = rw2A -> X (G p.azione != rw2A))) & G (p.azione = rw2B -> X (G p.azione != rw2B))) & G ((p.stato = l1A_B & p.evento = l2A) -> X (G (p.stato = l1A_B -> (p.evento != l2A & p.evento != l2B))))) & G ((p.stato = A_l1B & p.evento = l2B) -> X (G (p.stato = A_l1B -> (p.evento != l2B & p.evento != l2A))))) & G ((p.stato = l2A_B & p.evento = l1A) -> X (G (p.stato = l2A_B -> (p.evento != l2B & p.evento != l2A))))) & G ((p.stato = A_l2B & p.evento = l1B) -> X (G (p.stato = A_l2B -> (p.evento != l2B & p.evento != l2A))))) & G ((p.stato = l1A_l1B & (p.evento = l2A | p.evento = l2B)) -> X (G (p.stato = l1A_l1B -> (p.evento != l2A & p.evento!= l2B))))) & G ((p.stato = l2A_l2B & (p.evento = l1A | p.evento = l1B)) -> X(G (p.stato = l2A_l2B -> (p.evento != l1A & p.evento != l1B))))) & G ((p.evento = ul1A | p.evento = ul1B) -> X (G (p.evento != l1A & p.evento != l1B)))) &G ((p.evento = ul2A | p.evento = ul2B) -> X (G (p.evento != l2A & p.evento !=l2B)))) -> F (G p.stato = C1C2)) is false

-- as demonstrated by the following execution sequence

Trace Description: LTL Counterexample

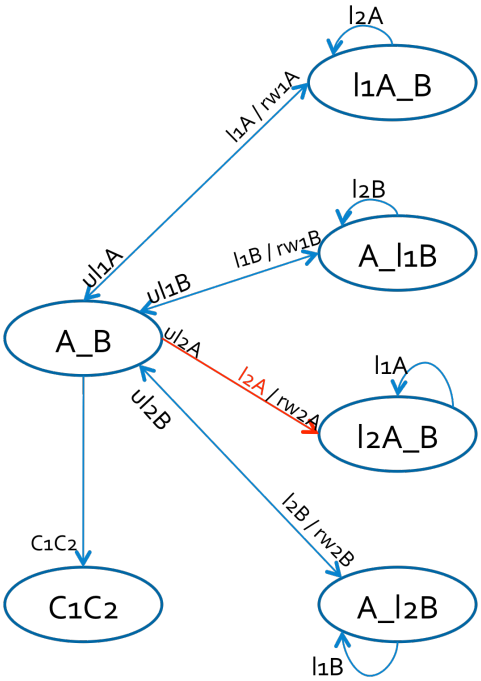
Trace Type: Counterexample

-> State: 1.1<-

p.stato = A_B

p.evento = l2A

p.azione = null

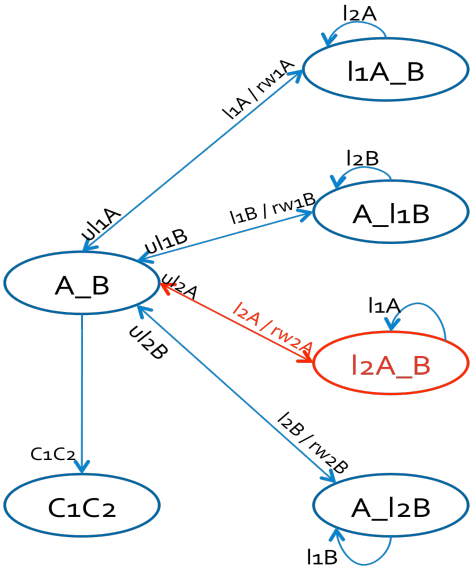


-> Input: 1.2 <-

-> State: 1.2 <-

p.stato = l2A_B

p.azione = rw2A

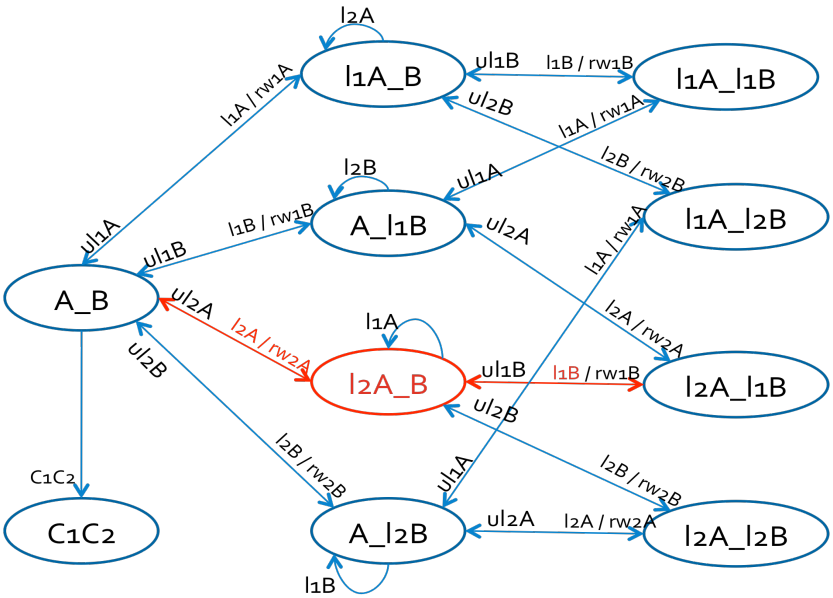


-> Input: 1.3 <-

-> State: 1.3 <-

p.evento = l1B

p.azione = null



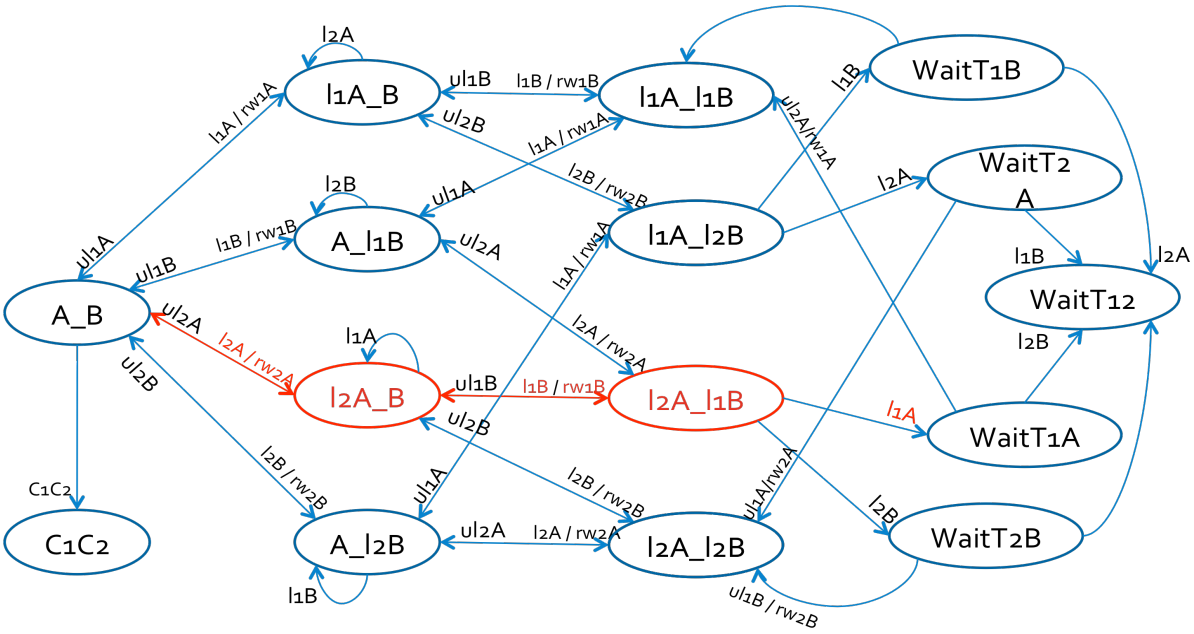
-> Input: 1.4 <-

-> State: 1.4 <-

p.stato = l2A_l1B

p.evento = l1A

p.azione = rw1B

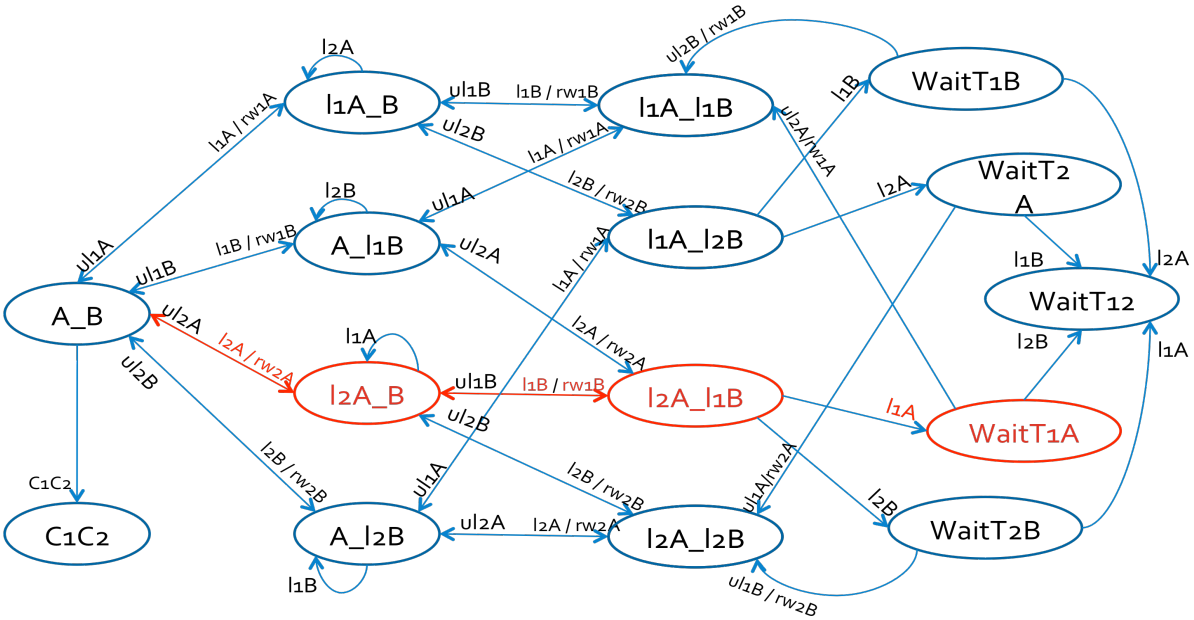


-> Input: 1.5 <-

-> State: 1.5 <-

p.stato = WT1A

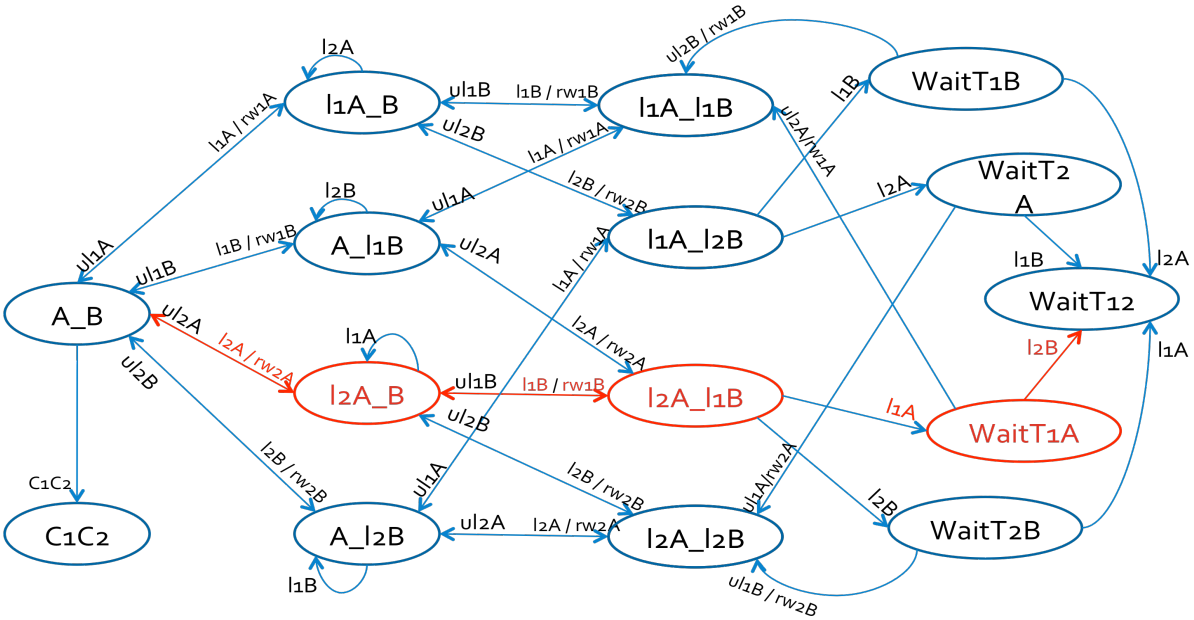
p.azione = null



-> Input: 1.6 <-

-> State: 1.6 <-

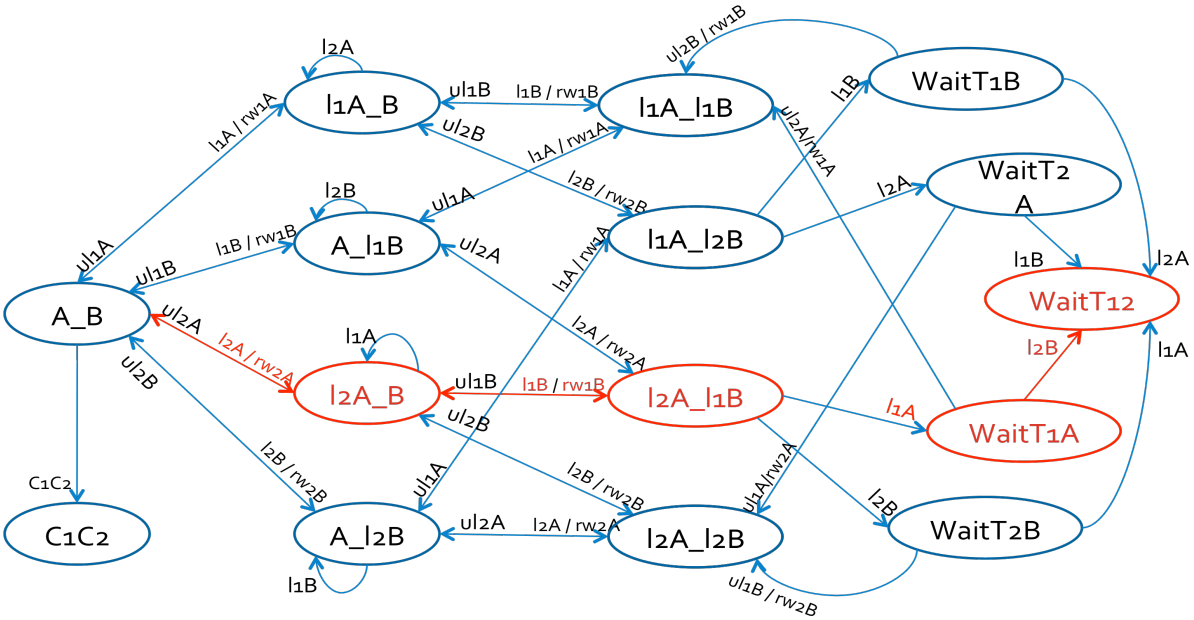
p.evento = l2B



-> Input: 1.7 <-

-> State: 1.7 <-

p.stato = WT1T2



-> Input: 1.8 <-

-- **Loop starts here**

-> State: 1.8 <-

p.evento = null

-> Input: 1.9 <-

-> State: 1.9 <-

Per la verifica dello stallo dobbiamo tenere conto che esistono modi anche efficienti per la gestione di tali situazioni. Questi modi possono prevedere la ‘prevenzione dello stallo’ e questo comporta un utilizzo di un protocollo ancora più stringente, oppure si utilizza la cosiddetta tecnica ‘Avoidance Deadlock’ ove l’assegnazione delle risorse alle transazione avviene secondo schemi tali che il deadlock non possa verficarsi, infine ‘detection/recovery’, quindi non si cerca di prevenire il problema bensì di individuarlo ed adottare delle tecniche per la sua

soluzione. Le tecniche possono prevedere una terminazione parziale o totale delle transazione con ovvie conseguenze sulle proprietà ACID garantite dal dbms oppure il ripristino di una situazione precedente correttamente funzionante. Anche in quest'ultimo caso bisogna fare attenzione al dbms perché si potrebbero verificare situazioni di inconsistenza.

VERIFICA PROPRIETA' ACID

Durabilità

Ora cerchiamo di provare le proprietà ACID. Possiamo già dire che la proprietà di **durabilità** è una caratteristica fisica del DBMS e quindi non è sotto il nostro controllo, non si può provare.

Atomicità

Passiamo ora alla proprietà di **atomicità**. Ricordiamo che l'atomicità impone che le transazioni devono essere eseguite per intero o per niente. Quindi possiamo immaginare intuitivamente che quando una transazione vuole effettuare un'operazione su un elemento chiederà il lock di tale variabile e se l'ottiene prima o poi ci sarà l'unlock altrimenti quella transazione non deve cominciare. Quindi ancora più semplicemente o viene eseguita per intero o per niente. Possiamo verificare questa proprietà supponendo se tutte le caratteristiche del 2PL vengono rispettate allora o le transazioni non cominciano (quindi rimando in A_B) oppure faranno il commit e quindi entreranno nello

stato C1C2. Pertanto verificheremo: PRE_→ ASS_ATOM, dove con PRE intendiamo le caratteristiche che ho già citato mentre con ASS_ATOM s'intende la condizione di atomicità:

G (F (stato = A_B | stato= C1C2)).

Questo è l'output che ottengo:

-- specification (((((((((((G (p.azione = rw1A -> X (G p.azione != rw1A)) & G (p.azione = rw1B -> X (G p.azione != rw1B))) & G (p.azione = rw2A -> X (G p.azione != rw2A))) & G (p.azione = rw2B -> X (G p.azione != rw2B))) & G ((p.stato = l1A_B & p.evento = l2A) -> X (G (p.stato = l1A_B -> (p.evento != l2A & p.evento != l2B))))) & G ((p.stato = A_l1B & p.evento = l2B) -> X (G (p.stato = A_l1B -> (p.evento != l2B & p.evento != l2A))))) & G ((p.stato = l2A_B & p.evento = l1A) -> X (G (p.stato = l2A_B -> (p.evento != l2B & p.evento != l2A))))) & G ((p.stato = A_l2B & p.evento = l1B) -> X (G (p.stato = A_l2B -> (p.evento != l2B & p.evento != l2A))))) & G ((p.stato = l1A_l1B & (p.evento = l2A | p.evento = l2B)) -> X (G (p.stato = l1A_l1B -> (p.evento != l2A & p.evento!= l2B))))) & G ((p.stato = l2A_l2B & (p.evento = l1A | p.evento = l1B)) -> X(G (p.stato = l2A_l2B -> (p.evento != l1A & p.evento != l1B))))) & G ((p.evento = ul1A | p.evento = ul1B) -> X (G (p.evento != l1A & p.evento != l1B)))) &G ((p.evento = ul2A | p.evento = ul2B) -> X (G (p.evento != l2A & p.evento !=l2B)))) -> G (F (p.stato = C1C2 | p.stato = A_B))) is **false**

-- as demonstrated by the following execution sequence

Trace Description: LTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

p.stato = A_B

p.evento = l2A

p.azione = null

-> Input: 1.2 <-

-> State: 1.2 <-

p.stato = l2A_B

p.azione = rw2A

-> Input: 1.3 <-
-> State: 1.3 <-
p.evento = I1B
p.azione = null
-> Input: 1.4 <-
-> State: 1.4 <-
p.stato = I2A_I1B
p.evento = I1A
p.azione = rw1B
-> Input: 1.5 <-
-> State: 1.5 <-
p.stato = WT1A
p.azione = null
-> Input: 1.6 <-
-> State: 1.6 <-
p.evento = I2B
-> Input: 1.7 <-
-> State: 1.7 <-
p.stato = WT1T2
-> Input: 1.8 <-
-- Loop starts here
-> State: 1.8 <-
p.evento = null
-> Input: 1.9 <-
-> State: 1.9 <-

Era quello che mi aspettavo. In poche parole otteniamo lo stesso risultato della sezione precedente, ossia la verifica dello stallo. Questo perché una volta che il sistema entra nello stallo non verifica più la proprietà di atomicità. Ora proviamo una cosa interessante dal punto di vista teorico anche se difficilmente realizzabile in pratica. Prendiamo non

più PRE_ bensì PRE comprendente di tutte le condizioni e proviamo che se noi non entrassimo mai in stallo allora il sistema garantirebbe anche la proprietà di atomicità. E' questo l'output che otteniamo:

-- specification (((((((((((((((G (p.evento = l1A -> F p.evento = ul1A) & G(p.evento = l1B -> F p.evento = ul1B)) & G (p.evento = l2A -> F p.evento = ul2A)) & G (p.evento = l2B -> F p.evento = ul2B)) & G (p.azione = rw1A -> X (G p.azione != rw1A))) & G (p.azione = rw1B -> X (G p.azione != rw1B))) & G (p.azione = rw2A -> X (G p.azione != rw2A))) & G (p.azione = rw2B -> X (G p.azione != rw2B))) & G ((p.stato = l1A_B & p.evento = l2A) -> X (G (p.stato =l1A_B -> (p.evento != l2A & p.evento != l2B))))) & G ((p.stato = A_l1B & p.evento = l2B) -> X (G (p.stato = A_l1B -> (p.evento != l2B & p.evento != l2A))))& G ((p.stato = l2A_B & p.evento = l1A) -> X (G (p.stato = l2A_B -> (p.evento != l2B & p.evento != l2A))))) & G ((p.stato = A_l2B & p.evento = l1B) -> X (G (p.stato = A_l2B -> (p.evento != l2B & p.evento != l2A))))) & G ((p.stato = l1A_l1B & (p.evento = l2A | p.evento = l2B)) -> X (G (p.stato = l1A_l1B -> (p.evento != l2A & p.evento != l2B))))) & G ((p.stato = l2A_l2B & (p.evento = l1A | p.evento = l1B)) -> X (G (p.stato = l2A_l2B -> (p.evento != l1A & p.evento != l1B))))) & G ((p.evento = ul1A | p.evento = ul1B) -> X (G (p.evento != l1A & p.evento != l1B)))) & G ((p.evento = ul2A | p.evento = ul2B) -> X (G (p.evento != l2A & p.evento != l2B)))) -> G (F (p.stato = C1C2 | p.stato = A_B))) is**true**

Anche qui ottengo quello che mi aspettavo ma è chiaro se mi metto nelle condizioni di non entrare in stallo, la proprietà di atomicità è rispettata.

Isolamento

Passiamo ora alla proprietà di **Isolamento**. Questa proprietà impone che ogni transazione si deve isolare dalle altre. Quindi le operazioni che esegue una transazione non devono essere disturbate o fatte in contemporanea con altre operazioni di altre transazioni. Quindi

dobbiamo astrarre le transazioni e pensarle come se fossero sole. Quando andiamo a scrivere su un elemento questa scrittura non deve essere disturbata. Posso procedere come sopra verificando appunto che PRE→ ASS_ISOL, dove PRE ha il significato ovvio mentre ASS_ISOL esprime la condizione di isolamento: G (evento = l1A & G evento != ul1A → G azione != rw2A) & G (evento = l1B & G evento != ul1B → G azione != rw2B) e simile anche per la transazione T2. Quindi praticamente vado a verificare che le due transazioni non si disturbano e che quindi l'effetto sulla risorse è lo stesso che si otterrebbe eseguendole isolatamente. Ottengo l'output mostrato di seguito:

-- specification (((((((((G (p.evento = l1A -> F p.evento = ul1A) & G (p.evento = l1B -> F p.evento = ul1B)) & G (p.evento = l2A -> F p.evento = ul2A)) & G (p.evento = l2B -> F p.evento = ul2B)) & G (p.evento = l1A -> X (G p.evento != l1A))) & G (p.evento = l1B -> X (G p.evento != l1B))) & G (p.evento = l2A -> X (G p.evento != l2A))) & G (p.evento = l2B -> X (G p.evento != l2B))) & G ((p.evento = ul1A | p.evento = ul1B) -> X (G (p.evento != l1A & p.evento != l1B)))) & G ((p.evento = ul2A | p.evento = ul2B) -> X (G (p.evento !=l2A & p.evento != l2B)))) -> (((G ((p.evento = l1A & G p.evento != ul1A) -> G p.azione != rw2A) & G ((p.evento = l1B & G p.evento != ul1B) -> G p.azione != rw2B)) & G ((p.evento = l2A & G p.evento != ul2A) -> G p.azione != rw1A)) & G ((p.evento = l2B & G p.evento != ul2B) -> G p.azione != rw1B))) is **true**

Anche qui otteniamo che la proprietà di Isolamento è garantita dal protocollo Two Phase Locking.

Consistency

Infine analizziamo la proprietà di **Consistency**, ossia che lo schedule deve lasciare in uno stato coerente la base di dati. Quindi dobbiamo andare a vedere che non ci siano situazioni pericolose che possano portare ad anomalie. Pensiamo, ad esempio, al caso in cui uno schedule preveda S: … rw1A ... rw2A … rw1A. Questo schedule può portare ad anomalie pericolose perché la transazione T1 può leggere valori incoerenti oppure ci può essere perdita di aggiornamento oppure, in alternativa, un aggiornamento fantasma. Procedo come sopra e vado a verificare: PRE→ ASS_CONS, dove ASS_CONS è la seguente:

G ((azione = rw1A & F azione = rw2A) → G X azione != rw1A), chiaramente questa condizione è messa in and con l'altra condizione m analoga su B. Si noti che questa condizione non significa solamente che il secondo comando rw1A avviene in un istante precedente a rw2A, ma che se in questo istante (t0) l'azione è rw1A ed esiste un istante futuro t' in cui azione = rw2A allora per tutti gli istanti temporali t'' > t0 azione != rw1A. t' e t'' sono non confrontabili quindi possono essere maggiori o minori l'uno dell'altro ma se questa condizione è soddisfatta allora è vera anche per t'' maggiore di t'.

Questo è l'output che si ottene:

-- specification (((((((((((G (p.evento = l1A -> F p.evento = ul1A) & G (p.evento = l1B -> F p.evento = ul1B)) & G (p.evento = l2A -> F p.evento = ul2A)) & G (p.evento = l2B -> F p.evento = ul2B)) & G (p.evento = l1A -> X (G p.evento != l1A))) & G (p.evento = l1B -> X (G p.evento != l1B))) & G (p.evento = l2A -> X (G p.evento != l2A))) & G (p.evento = l2B -> X (G p.evento != l2B))) & G ((p.evento = ul1A | p.evento = ul1B) -> X (G (p.evento != l1A & p.evento != l1B)))) & G ((p.evento = ul2A | p.evento = ul2B) -> X (G (p.evento !=l2A & p.evento != l2B)))) -> (((G ((p.evento = l1A & G p.evento != ul1A) -> p.azione != rw2A) & G ((p.evento = l1B &

G p.evento != ul1B) -> p.azione != rw2B)) & G ((p.evento = l1A & G p.evento != ul1A) -> p.azione != rw2A)) & G ((p.evento = l1B & G p.evento != ul1B) -> p.azione != rw2B))) is **true**

Ottengo anche qui che il protocollo Two Phase Locking rispetta la proprietà di Consistency.

Protocollo 2PL con lock condivisi (shared locks)

Fino ad ora ho considerato lock esclusivi, cioè lock che escludono il possesso di un elemento ad ogni altra transazione. Introduco un nuovo tipo di lock, il lock condiviso sl, chiamato anche lock in lettura. Questa estensione non introduce novità essenziali da un punto di vista semantico ma aumenta l'efficienza perché effettivamente permette di poter accedere concorrente in lettura su elementi condivisi da parte di varie transazioni. Si possono quindi facilmente estendere i concetti di transazione ben formata e schedule legale.

Il protocollo del Two Phase Locking con lock condivisi si esprime nel seguente modo: "..uno schedule segue il protocollo del two-phase docking se in ogni transazione Ti presente nello schedule tutte le operazioni di lock di Ti precedono tutte le operazioni di unlock di Ti" [definizione presa dalle dispense del prof. M. Lenzerini].

Diagramma UML degli stati e delle transizioni

Passo ora a dare una versione più operativa del protocollo riportandolo ai miei obiettivi. Appena si prova a modellare il diagramma UML degli stati e delle transazioni oppure a implementare il protocollo in Nusmv ci

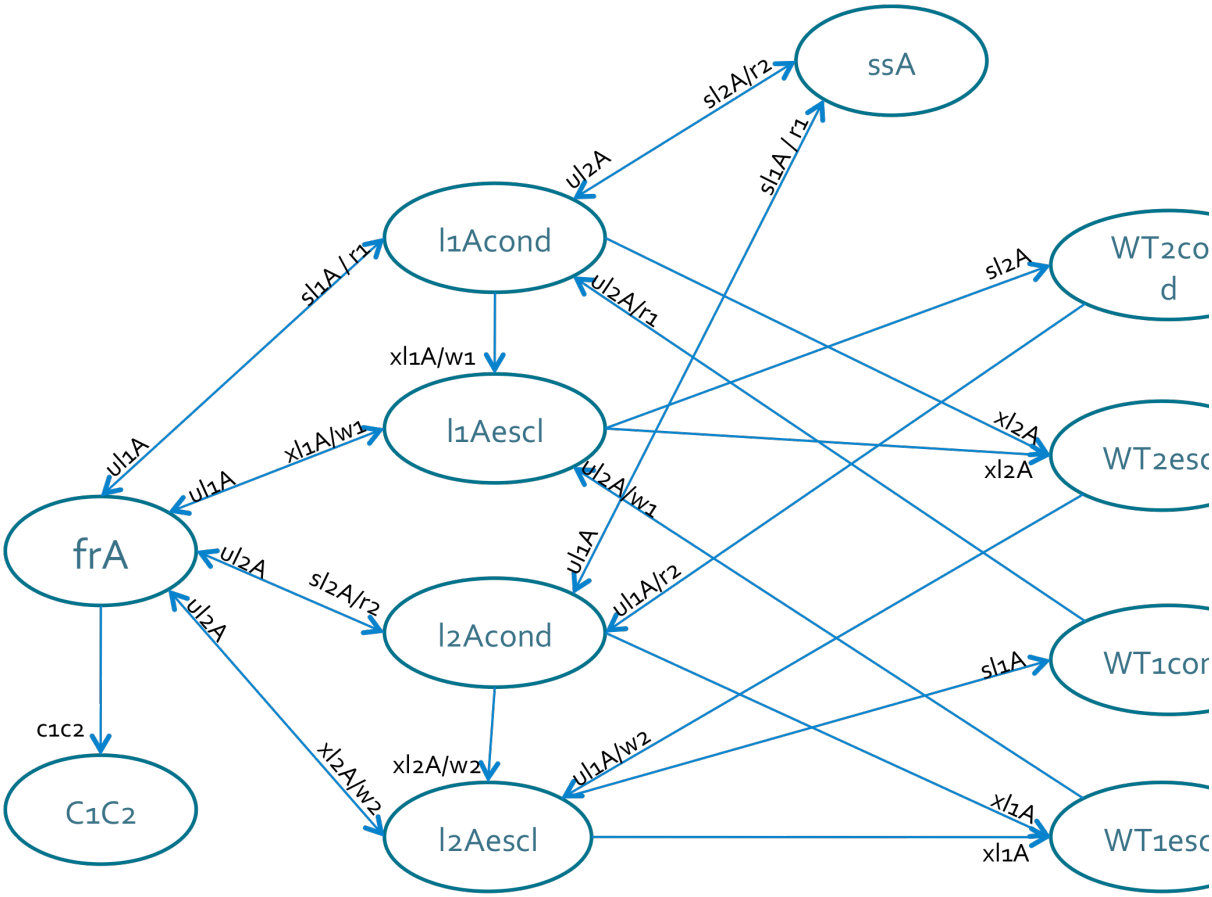
si rende subito conto dell'aumento notevole del numero degli stati. La difficoltà non è nell'implementazione quanto nella rappresentazione del diagramma e nelle conseguenti numerose linee di codice. A questo punto ho dovuto fare una scelta tenendo ben presente l'obiettivo della tesina e le novità introdotte da questo protocollo. Ho preferito dare una versione ridotta della dimensione di questo problema approfondendo in seguito il protocollo strict 2PL che mi permette di dare importanti novità a livello semantico e di rilassare l'assunzione piuttosto restrittiva che tutte le transazioni effettuino il commit. Ho voluto introdurre e trattare comunque questo protocollo perché ho ritenuto importante spiegare questo tipo di lock, il lock condiviso, che permette di migliorare le performance del dbms in maniera notevole.

In maniera intuitiva,anche se è possibile approfondire il discorso in maniera formale, si possono estendere tutte le considerazioni fatte nel caso di lock esclusivi e del protocollo Two Phase Locking. L'introduzione di lock condivisi ci permette di eliminare talune situazioni pericolose in cui si verificano uno stallo per una lettura doppia. Mi spiego meglio, riusciamo ad evitare delle situazioni in cui delle transazioni entravano in stallo poiché dovevano effettuare delle operazioni di lettura e nel contempo avevano ottenuto il lock su risorse che volevano solo leggere. Paradossalmente una situazione di stallo poteva essere data dal seguente schedule: r1A, r2B, r1B, r2A. Come si intuisce, in questa situazione le risorse A e B potevano essere benissimo condivise, quindi con il nuovo protocollo potevamo richiedere solo lock condivisi e permettere questo tipo di schedule.

Nell'implementazione in Nusmv per quanto detto prima ho considerato un caso limitato a 2 transazioni che accedono ad una risorsa. Il caso è

restrittivo ma mettendo insieme quanto verificato nella prima parte con quando detto in questo paragrafo è ovvio che questi due argomenti sono complementari ed integrabili per quanto l'implementazioni risulti in un'esplosione di stati.

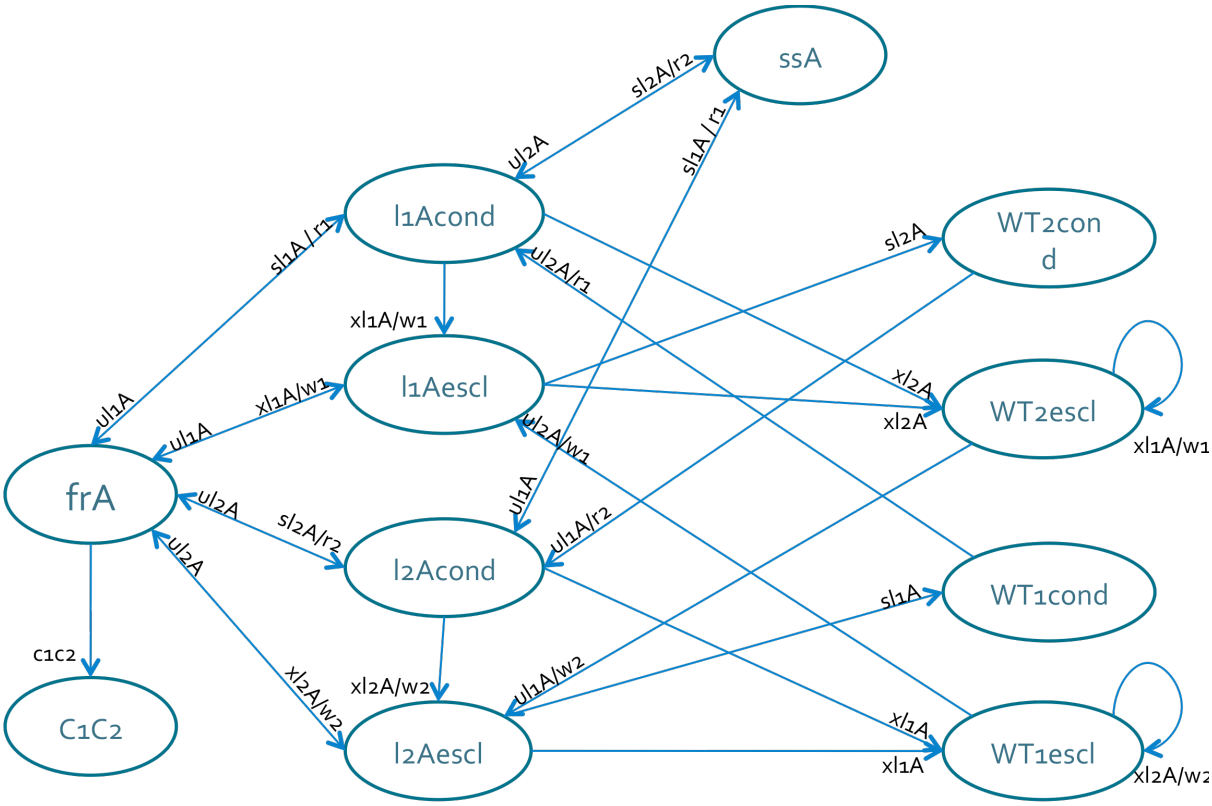
Siano T1 e T2 le due transazioni e A l'elemento condiviso. Di seguito riposto il diagramma UML degli stati e delle transazioni.



Dal grafico si può fare subito un'osservazione importante. Ho dato la possibilità ad una transazione di ottenere il lock esclusivo avendo già un lock condiviso sulla stessa risorsa. Il meccanismo consentito è noto come lock upgrade.

Ho dovuto considerare 4 situazioni di attesa, questo perché quando entro in attesa significa che sto aspettando un unlock di un elemento e devo mantenere un'informazione sul comando che ha provocato la situazione d'attesa. Le etichette degli stati, degli eventi e delle azioni hanno il significato ovvio già discusso nel precedente paragrafo.

Durante l'implementazione e una visita approfondita del diagramma UML ho potuto verificare che questo diagramma soffre di un'anomalia. In particolare esaminando i due WT1escl e WT2escl ed il percorso con cui si è arrivati in tali stati ci rendiamo conto che non è possibile, così come è fatto il modello, fare il lock upgrade. Mi spiego meglio con un esempio. Mettiamo di stare nello stato sl1A e che l'evento sia xl2A, chiaramente non potendo esaudire tale richiesta di lock lo scheduler mette in attesa la transazione T2 passando allo stato WT2escl aspettando l'unlock di T1. Nel frattempo può però darsi che la transazione T1 voglia eseguire un lock upgrade, quindi che l'evento sia xl1A, per come è modellato il sistema ora ciò non è possibile. Provo allora a porre una piccola modifica al diagramma UML precedente permettendo tale operazione:



Ho aggiunto due archi agli stati WT2escl e WT1escl. Così facendo permetto il lock upgrade pur mantenendo in attesa la transazione T1 o T2. Ci si potrebbe domandare se tale aggiunta sia corretta o meno perché in questi due stati ci si arriva con due percorsi diversi sia per l'uno che per l'altro. Infatti a WT2escl ci si arriva da l1Acond e l1Aescl. In questo secondo caso potremmo pensare che sia scorretto lasciare la possibilità di eseguire un'altra volta un lock esclusivo perché ciò va contro all'assunzione fatta in precedenza che le transazioni eseguano una sola volta l'operazione di lock (in questo caso esclusivo). In realtà, stando al protocollo ed ai requisiti descritti nel caso del 2PL con lock esclusivi ma che valgono anche adesso, io non percorrerò mai l'arco aggiunto se in passato ho già effettuato e ottenuto il lock esclusivo su A. In Nusmv implementerò la possibilità di effettuare solo una volta

l'operazione di lock esclusivo e condiviso. Si noti bene però che è ammessa la possibilità di passare dal lock condiviso al lock esclusivo.

IMPLEMENTAZIONE IN NUSMV

Nell'implementazione in Nusmv ho verificato solamente le situazioni di stallo che nel caso considerato chiaramente non ci sono.

Verifica dello stallo

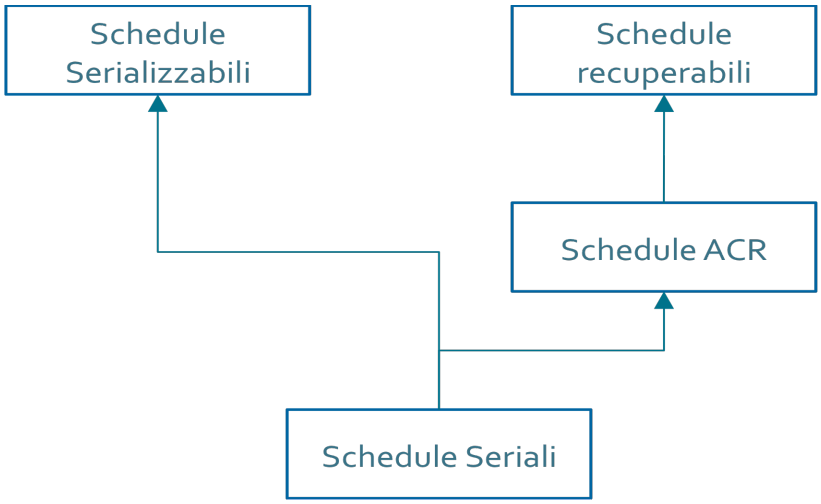
Le assunzioni e le considerazioni sono in larga quelle effettuate nella sezione precedente. In LTLSPEC ho utilizzato un'espressione del tipo: PRE -> POST, mettendo in PRE tutti i requisiti e le condizioni del protocollo e in POST l'asserzione della quale volevo verificare la validità. Ho verificato che partendo dallo stato iniziale frA valesse sempre la formula per la quale esiste sempre in istante nel futuro in cui è sempre vero che siamo in C1C2, ovvero che le due transazioni effettuino il commit. Ecco l'output restituito:

-- specification ((((((((G ((p.evento = sl1A | p.evento = xl1A) -> F p.evento= ul1A) & G ((p.evento = sl2A | p.evento = xl2A) -> F p.evento = ul2A)) & G (p.evento = sl1A -> X (G p.evento != sl1A))) & G (p.evento = xl1A -> X (G p.evento != xl1A))) & G (p.evento = sl2A -> X (G p.evento != sl2A))) & G (p.evento = xl2A -> X (G p.evento != xl2A))) & G (p.evento = ul2A -> X (G (p.evento != sl2A & p.evento != xl2A)))) & G (p.evento = ul1A -> X (G (p.evento !=sl1A & p.evento != xl1A)))) -> F (G p.stato = C1C2)) is **true**

Protocollo strict 2PL

Fino ad ora ho introdotto dei protocolli molto interessanti ma con una importante assunzione, tutti le transazioni finiscono ed eseguono il commit. Sebbene i casi fin qui visti sono di elevata importanza a livello teorico, nella pratica purtroppo questa assunzione è molto limitativa. Useremo tutto quello visto fino a questo punto ma rilasseremo l'assunzione di commit delle transazioni e introduciamo delle nozioni e dei concetti che ci porteranno a definire il protocollo Strict Two Phase Locking, che chiameremo per brevità **strict-2PL**. All'inizio di questo documento ho definito un'operazione di rollback di una transazione. Questa disfa il lavoro svolto fin al momento della sua invocazione sulla transazione con conseguente molto delicate e da studiare sul dbms. Quest'operazione in qualche modo sembra stravolgere tutte le nostre convinzioni acquisite sino ad ora, in realtà invece vedremo nel seguito che questa sensazione è in buona parte sbagliata. Vedremo un'estensione del protocollo Two Phase Locking e quindi continueranno a valere i concetti precedenti ma dobbiamo fare delle ulteriori considerazioni. Procediamo con calma studiando prima gli effetti del rilassamento dell'assunzione di cui sopra. Ammettendo che una transazione Ti effettui il rollback che indicherò con r_i, garantire che uno schedule rispetti il protocollo 2PL non basta più. Sia uno schedule dato da 2 transazioni: l1(A), w1(A), ul1(A), l2(A), r2(A), ul2(A), r1, c2. Questo schedule è 2PL ma non è ammissibile perché la transazione T2 legge il valore di A modificato dalla transazione T1 che poi effettuerà il rollback, quindi T2 legge un valore "sporco". Questa anomalia è chiamata 'lettura sporca' (cfr. dispense di M.Lenzerini). Questa prima conseguenza può

essere devastante per applicazioni bancarie, e-commerce, ecc. Ma vediamo un altro effetto ancora più negativo. Riprendiamo lo schedule di prima, cosa bisogna fare con T2 che legge un valore sporco? Possiamo gestire la situazione non permettendo a T2 di leggere valori scritti da transazioni che non siamo ancora terminate oppure effettuare l'abort di T2. Ora considerando l'ultimo caso ed estendendo lo schedule a più transazioni l'abort di una di queste può provocare l'abort di molte altre transazioni. Questo fenomeno è chiamato Cascading Rollback (anche qui, cfr. dispense di M.Lenzerini). Questo può essere gestito da un modulo del dbms, il Recovery Manager, oppure tramite protocolli più stringenti. Detto questo, introduciamo allora un importante classe di schedule, schedule recuperabili. '.. uno schedule è recuperabile se nessuna transazione in S esegue il commit prima che tutte le altre transazioni dalle quali essa ha letto abbiano eseguito il commit .. '. Dopo gli schedule serializzabili allora ho introdotto questa particolare classe, il cui concetto è ortogonale da quello di serializzabilità poiché esistono schedule recuperabili non serializzabili e viceversa. E' da chiarire che tali schedule recuperabili ammettono ancora il fenomeno del Cascading Rollback e quindi dobbiamo introdurre un'ulteriore classe che evita tale fenomeno chiamata ACR. Questi schedule sono ACR se ' .. ogni transazione legge solo valori scritti da altre transazioni che hanno già eseguito il commit .. '. Il discorso precedente è chiarito dalla figura seguente:



Le frecce rappresentano collegamenti ISA. Ora dobbiamo capire però come utilizzare l'esperienza fin qui accumulata per creare un protocollo che garantisce insieme serializzabilità e recuperabilità. Procedendo gradualmente definiamo una sottoclasse degli schedule ACR, gli schedule stretti. Diciamo che uno schedule è stretto se ogni transazione legge e scrive solo su valori scritti da transazioni che hanno già effettuato il commit. Diamo finalmente la definizione di un particolare schedule stretto, ossia 2PL stretto. Un protocollo segue lo strict 2PL se '.. segue il protocollo 2PL ed, inoltre, i lock di ogni transazione Ti vengono mantenuti fino al commit o l'abort di Ti.. '.

Si può facilmente capire come questo protocollo sia di grande importanza poiché è una sottoclasse sia di schedule serializzabili e che ACR (e quindi recuperabili). E' sicuramente quello più forte e stringente che abbiamo incontrato ma ci permette di rilassare qualsiasi tipo di assunzione fatta in precedenza. Per fare un certo ordine mentale è utile immaginare graficamente le varie relazione tra tutti gli schedule che abbiamo incontrato e vedere le relazioni. In riferimento a quando detto

prima circa l'ortogonalità degli schedule recuperabili e serializzabili, dal grafico seguente capiamo la grande importanza del protocollo strict 2PL. Ogni schedule strict 2PL è sia uno schedule serializzabile che recuperabile.

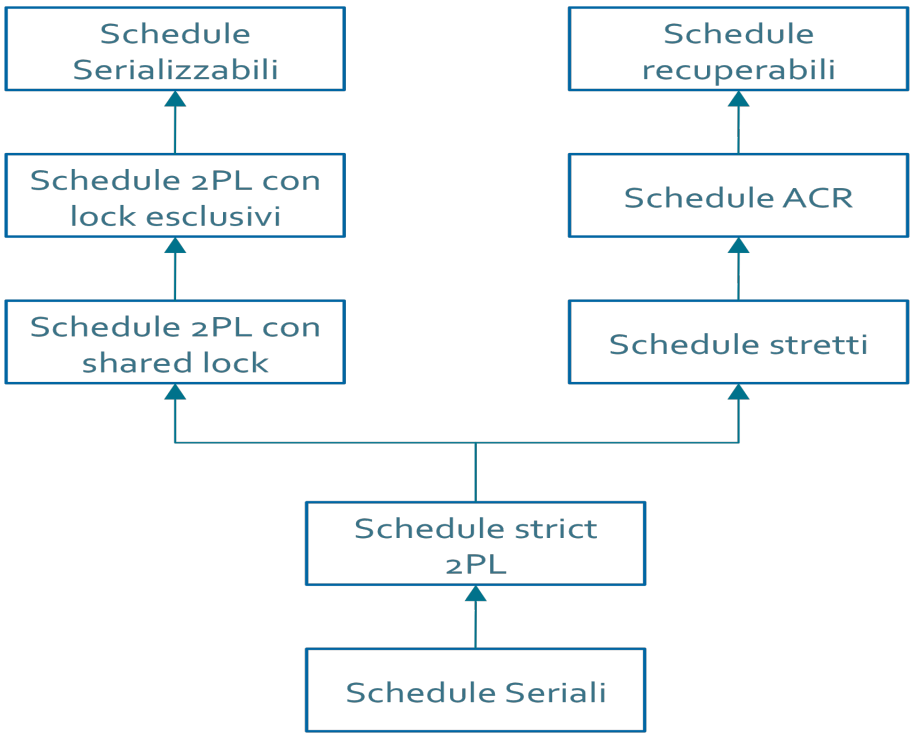


Diagramma UML degli stati e delle transizioni

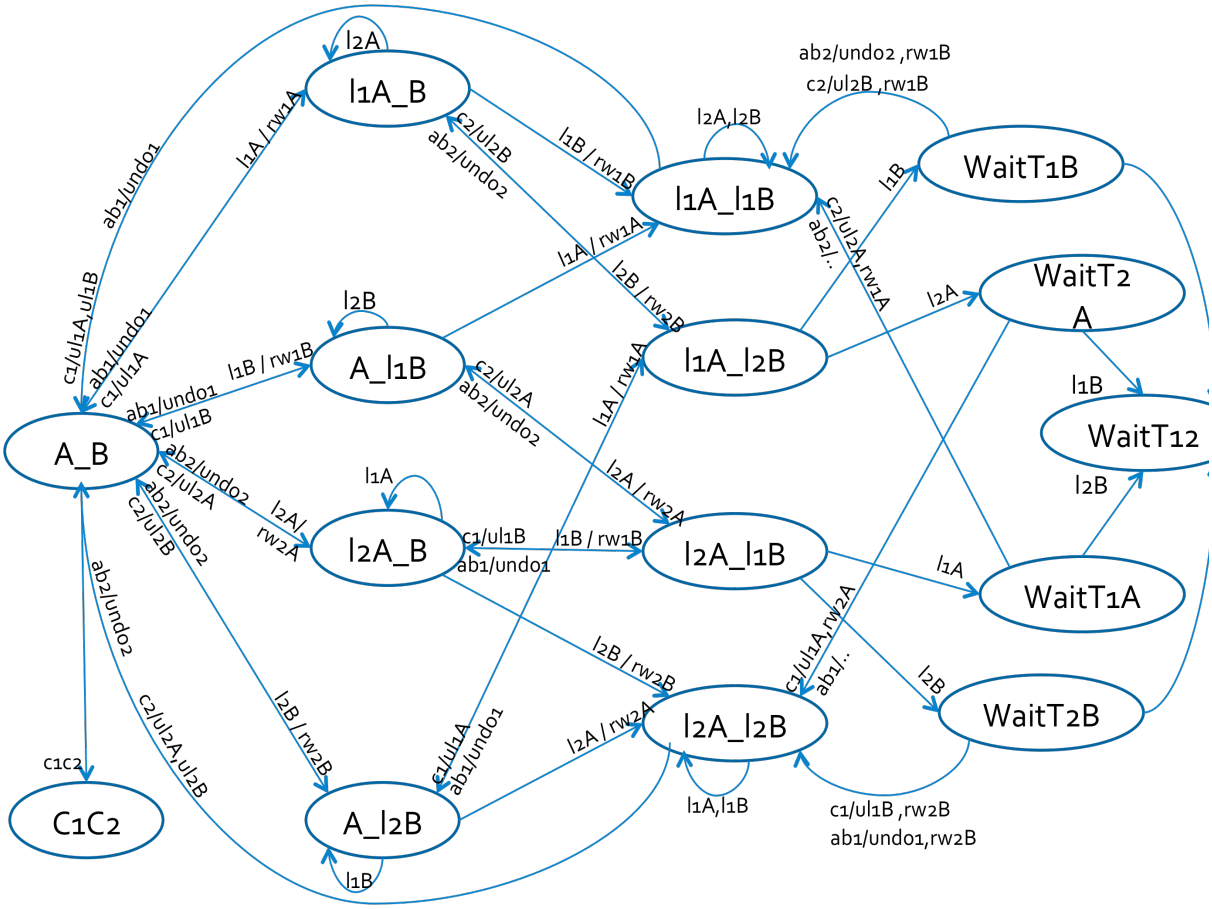
Passiamo ora ad un'interpretazione più operativa di tale protocollo, modelliamo il diagramma UML degli stati e delle transazioni. Ora devo considerare le operazioni di abort e commit quindi questo complicherà ancora di più il disegno. Gli stati rimangono i medesimi mentre abbiamo qualche azione in aggiunta e dei meccanismi diversi per passare da uno stato all'altro. Analizziamo bene la definizione di protocollo strict 2PL, uno schedule è in strict 2PL se segue il protocollo 2PL ed i lock di ogni

transazione Ti vengono mantenuti fino al commit o l'abort di Ti. La prima parte riprende tutti i concetti spiegati nel protocollo 2PL per definizione mentre la seconda parte contiene delle novità essenziali. Il fatto di mantenere di mantenere i lock significa non eseguire l'operazione di unlock fino al commit o abort e questo modifica sostanzialmente i ragionamenti precedenti per unlock. Ora quindi consideriamo come evento il commit o l'abort di una transazione e questo evento porta come conseguenza, come azione rispettivamente l'unlock degli elementi di cui si detiene il lock e l'undo. Mi soffermo su questa cosa perché merita una spiegazione. Al termine di ogni transazione, se questa ha effettuato un commit il sistema deve garantire che i suoi effetti sia registrati nella base di dati mentre se effettua il rollback il sistema deve disfare tutte le operazioni della transazione senza lasciare alcun effetto. Quindi ho considerato in maniera astratta un'azione di undo che disfa il lavoro svolto dalla transazione e quindi ne annulla le operazioni. Ricapitolando il commit o l'abort avrà come effetto l'unlock. Detto questo dobbiamo garantire che le transazioni non effettuino alcuna operazioni di lock dopo il commit (abort). Più o meno quindi possiamo ripetere le considerazioni fatte precedentemente:

- Ogni transazione una volta effettuato il lock su una variabile deve effettuare in un tempo futuro un unlock su tale variabile. Questo per garantire che le transazioni sia ben formate.
- Ogni transazione effettua al massimo una volta l'operazione di lock su una variabile.
- Ogni transazione una volta effettuata un'operazione di commit o abort non può più effettuare un'operazione di lock su qualsiasi elemento. Questo per garantire il protocollo 2PL.

- Se ho effettuato un'operazione di lock su un elemento ed un'altra transazione chiede di fare il lock su tale elemento entro in una stato d'attesa aspettando che l'elemento venga rilasciato. Questo per come è fatto il protocollo.
- I lock vengono mantenuti fino al commit o all'abort quindi nessuna transazione esegue un'operazione di unlock prima di tali eventi. Inoltre come detto le operazioni di commit/abort provocano unlock sugli elementi lockati dalla stessa transazione.

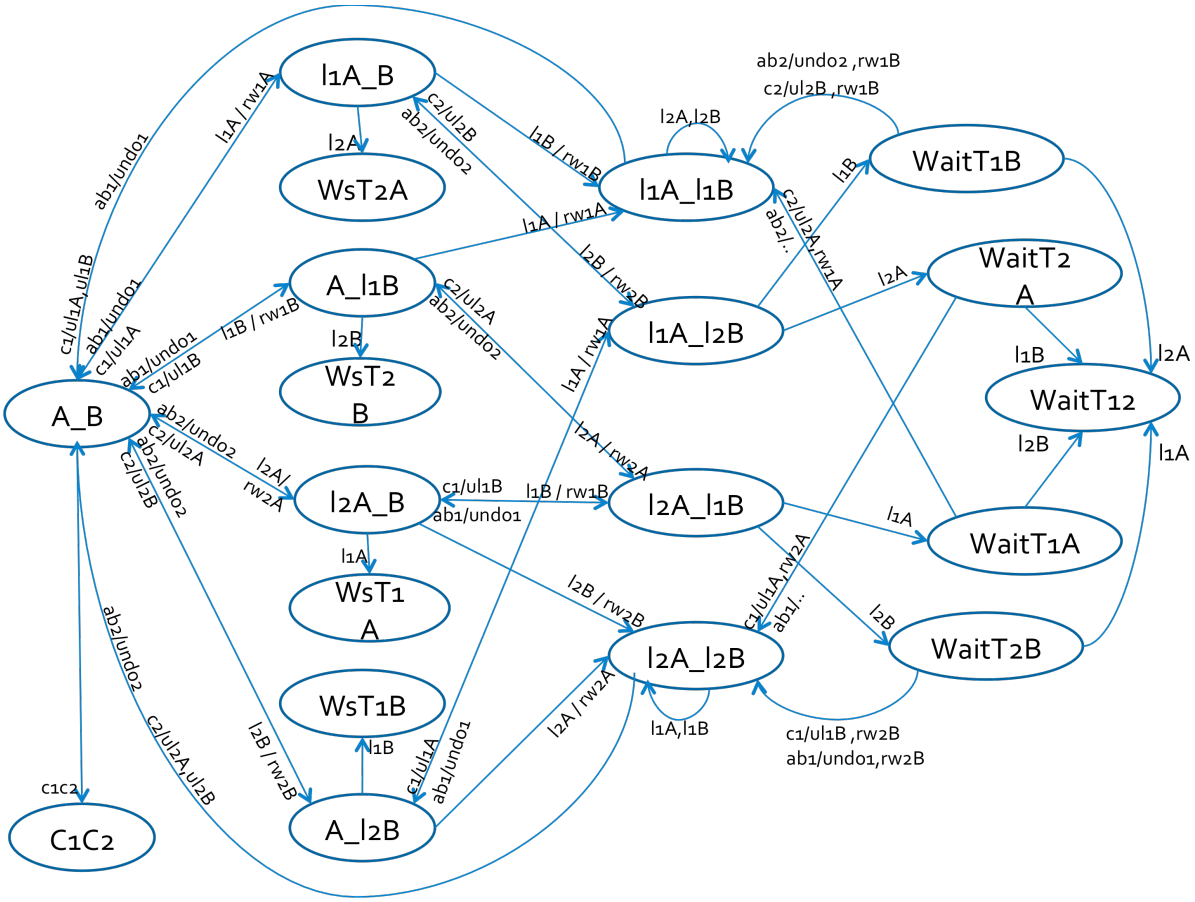
Di seguito mostro il diagramma UML degli stati e delle transazioni:



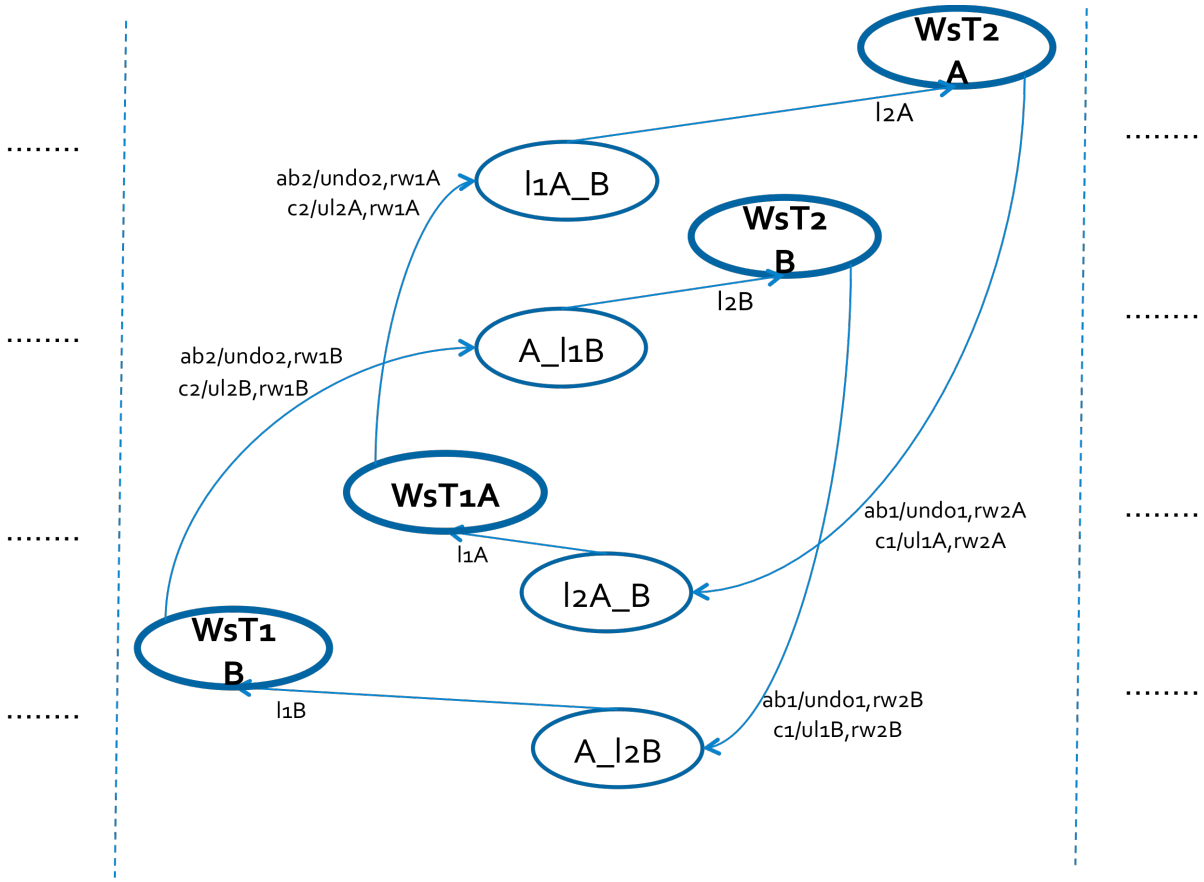
Tale diagramma non è di facile lettura, questo per un elevato numero di stati eventi azioni e perché stiamo modellando un protocollo più stringente del protocollo 2PL. Quindi questo sarà complicato da realizzare e accetterà solo un sottoinsieme degli schedule come visto in precedenza.

Tale modello UML rispetta ciò che era detto ma ad una visita approfondita ci si può rendere conto di un'imprecisione che voglio analizzare. Quando sto nello stato, ad esempio I1A_B, e la transazione T2 chiede il lock su A, ho simulato uno stato d'attesa ciclando sullo stato. Non è corretto perché se ad esempio il prossimo evento è I2B si passa allo stato I1A_I2B ma questo non va bene perché devo mettere T2 in attesa. Quindi in realtà si doveva entrare in uno stato d'attesa diverso dove si permettevano solo le operazione della transazione T1 e si bloccava facendola attendere T2. Ci sono molte considerazioni da fare intorno a questa osservazione. Intanto di seguito descrivo come andrebbe corretto e poi spiego perché il diagramma fin qui considerato non introduce significativi errori a livello di semantica e di schedule accettati.

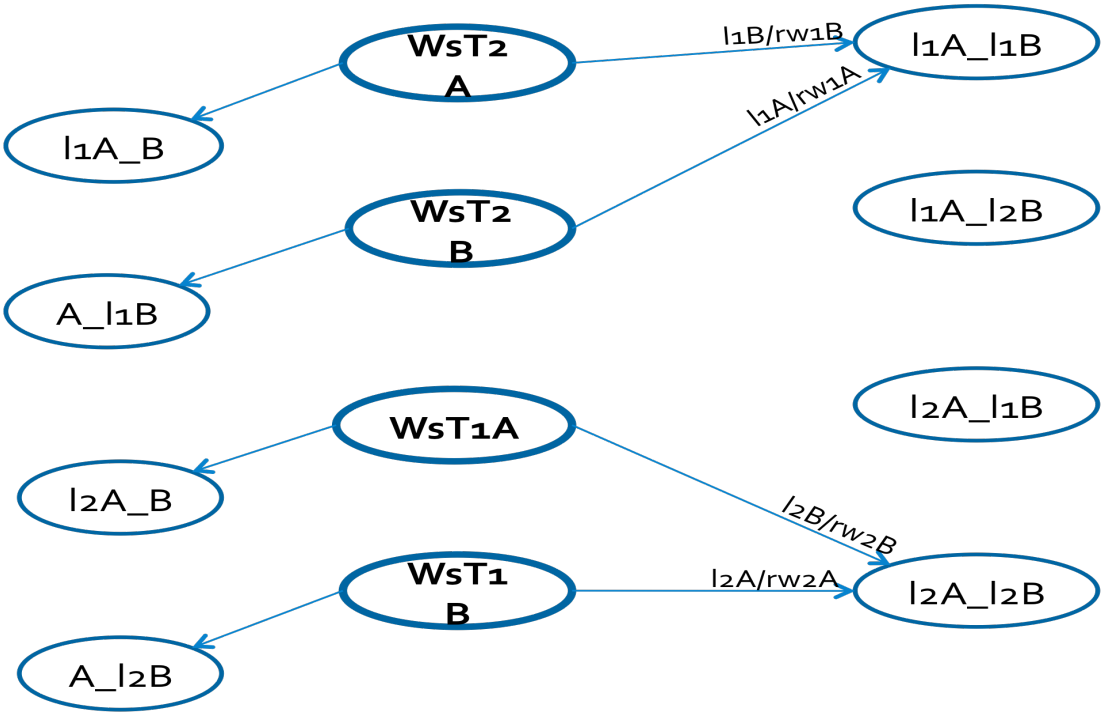
Correggendo l'errore, bisogna chiaramente introdurre degli stati aggiuntivi che permetteno al sistema di mettere realmente in attesa la transazione che richiede il lock su un elemento già lockato da un'altra. Il diagramma UML degli stati e transazioni ottenuto è il seguente:



Per motivi ovvi il diagramma non è completo, altrimenti sarebbe risultato non leggibile e pertanto ho preferito aggiungere solamente gli stati di attesa e poi spiegarli separatamente nel seguito. Gli stati d'attesa aggiunti sono WsT1A, WsT1B, WsT2A e WsT2B. Questi sono stati d'attesa diversi da quelli introdotti in precedenza, la differenza fondamentale sta nel fatto che chi aspetta non detiene nessun lock. Per questo motivo li ho chiamati stati d'attesa semplice WsTiX (simple wait of Ti for X). Analizziamoli nel dettaglio cominciando a considerare cosa succede quando le transazioni che detengono (lock) l'elemento richiesto effettuano il commit o abort.La seguente figura mostra tale scenario:



Come si può vedere se sto nello stato d'attesa semplice ed effettuo il commit o abort passo direttamente ad uno stato lockato. Mi spiego meglio con un esempio. Se sto nello stato l1A_B e l'evento è l2A ora passo a WsT2A, quando la transazione T1 effettua il commit (abort) passo allo stato l2A_B facendo l'unlock di A e permettendo la scrittura/lettura di A da parte di T2. In questi stati solo una transazione può effettuare operazione, l'altra attende. Ad esempio nello stato WsT2A permetto a T1 di operare un lock su B e di passare conseguentemente di l1A_l1B. Qui lo scheduler riproporrà l'evento che ha portato precedentemente allo stato d'attesa l2A ed anche in questo caso la transazione T2 dovrà aspettare il commit (abort) di T1.



Ora vediamo perché anche se non è corretto da un punto di vista rigoroso perché non modella correttamente il protocollo possiamo comunque accettare questa imprecisione e giungere a conclusioni valide. Prima si ciclava sullo stesso stato dando anche la possibilità di procedere di procedere con la transazione che in teoria doveva essere messa in attesa, quindi il nostro stato di attesa era fittizio. In realtà quando lo scheduler mette in attesa una transazione, la blocca partendo da una determinata operazione per poi riprenderla da questa stessa istruzione. Quindi ponendo come requisito che la transazione se non riesce a lockare una risorsa non può più effettuare nessun altra operazione se non il lock su quella risorsa, si risolveva il problema limitando le conseguenze. Cambiando prospettiva del problema e vedendola in una visione più pratica ed ingegneristica, considerando che si tratta di un'astrazione, tale imprecisione non altera le proprietà che vogliamo andare a verificare. In poche parole si sarebbe anche potuto

ignorare il problema sebbene la versione che andrò a modellare ne tiene conto e lo corregge.

IMPLEMENTAZIONE IN NUSMV

Si è proceduto in maniera simile al caso del protocollo Two Phase Locking. La tecnica adottata è la stessa solo che cambiano chiaramente alcuni particolari nella definizione delle variabili e delle pre-condizioni. Come si può vedere facilmente dal diagramma UML ho dovuto aggiungere degli stati oltre che degli eventi e azioni. Le variabili che ho dichiarato sono le seguenti:

stato: {A_B, C1C2, l1A_B, A_l1B, l2A_B, A_l2B, l1A_l1B, l1A_l2B, l2A_l1B, l2A_l2B, WT1B, WT2A, WT1A, WT2B, WsT1A, WsT1B, WsT2A, WsT2B, WT1T2};

evento: {l1A, l1B, l2A, l2B, c1, c2, undo1, undo2, ab1, ab2, c1c2, null};

azione: {rw1A, rw1B, rw2A, rw2B, undo1, undo2, ul1AB, ul2AB, ul1A, ul1B, ul2A, ul2B, undo2_rw1B, ul2B_rw1B, ul2A_rw1A, undo2_rw1A, ul1A_rw2A, undo1_rw2A, ul1B_rw2B, undo1_rw2B, null};

Le precondizioni che ho imposto sono le seguenti:

- Ogni transazione può richiedere con successo al massimo una volta il lock

```
(G( p.azione = rw1A -> X G p.azione != rw1A )          &
G( p.azione = rw1B -> X G p.azione != rw1B )           &
G( p.azione = rw2A -> X G p.azione != rw2A )           &
G( p.azione = rw2B -> X G p.azione != rw2B )
```

- La transazione una volta entrata in attesa deve aspettare:

G(p.stato = l1A_l1B & (p.evento = l2A | p.evento = l2B) -> X G (p.stato = l1A_l1B -> (p.evento != l2A & p.evento != l2B))) &
G(p.stato = l2A_l2B & (p.evento = l1A | p.evento = l1B) -> X G (p.stato = l2A_l2B -> (p.evento != l1A & p.evento != l1B)))

Ogni transazione una volta effettuata un'operazione di commit/abort non può più effettuare un'operazione di lock su qualsiasi elemento:

G(p.evento = c1 | p.evento = ab1 -> X G (p.evento != l1A & p.evento != l1B)) &
G(p.evento = c2 | p.evento = ab2 -> X G (p.evento != l2A & p.evento != l2B)))

Verifica dello stallo

Anche qui ho proceduto in maniera simile a quanto fatto precedentemente, Sono andato a verificare lo stallo andando a controllare se partendo dallo stato iniziale si arrivasse sempre in uno stato finale di commit chiamato C1C2. Quindi sono andato a vedere se soddisfatte tutte le precondizioni fosse soddisfatta la suddetta condizione di 'arrivo' con il metodo PRE_ → ASS_Stallo dove PRE_ sono le precondizioni imposte e dichiarate nella sezione precedente mentre ASS_Stallo è la seguente:

- -> (F G p.stato = C1C2)

Ed ottenendo il seguente risultato come previsto:

-- specification (((((((G (p.azione = rw1A -> X (G p.azione != rw1A)) & G (p.azione = rw1B -> X (G p.azione != rw1B))) & G (p.azione = rw2A -> X (G p.azione != rw2A))) & G (p.azione = rw2B -> X (G p.azione != rw2B))) & G ((p.stato = I1A_I1B & (p.evento = I2A | p.evento = I2B)) -> X (G (p.stato = I1A_I1B -> (p.evento != I2A & p.evento != I2B))))) & G ((p.stato = I2A_I2B & (p.evento = I1A | p.evento = I1B)) -> X (G (p.stato = I2A_I2B -> (p.evento != I1A & p.evento != I1B))))) & G ((p.evento = c1 | p.evento = ab1) -> X (G (p.evento !=I1A & p.evento != I1B)))) & G ((p.evento = c2 | p.evento = ab2) -> X (G (p.evento != I2A & p.evento != I2B)))) -> F (G p.stato = C1C2)) is **false**

-- as demonstrated by the following execution sequence

Trace Description: LTL Counterexample

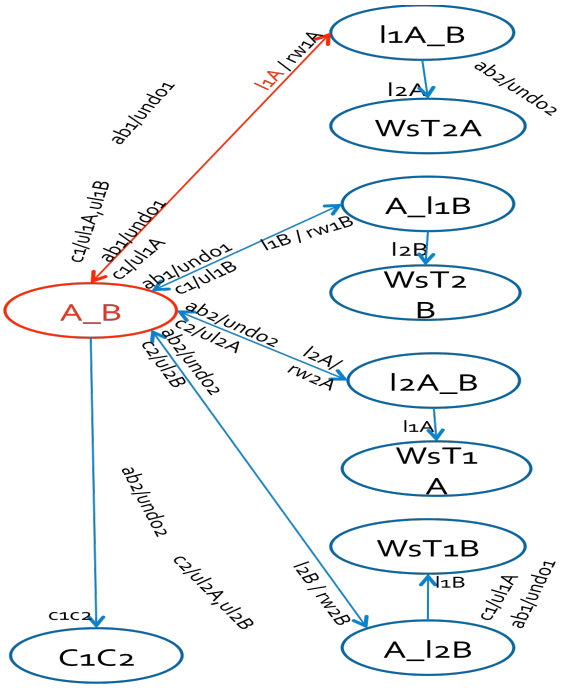
Trace Type: Counterexample

-> State: 1.1 <-

p.stato = A_B

p.evento = I1A

p.azione = null

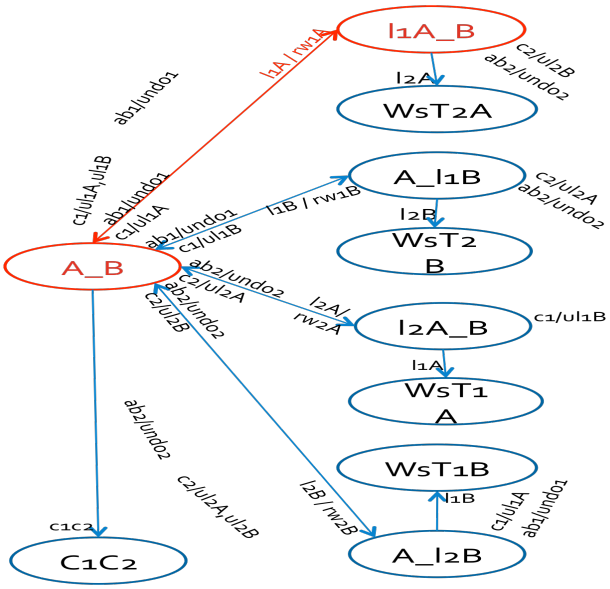


-> Input: 1.2 <-

-> State: 1.2 <-

p.stato = l1A_B

p.azione = rw1A

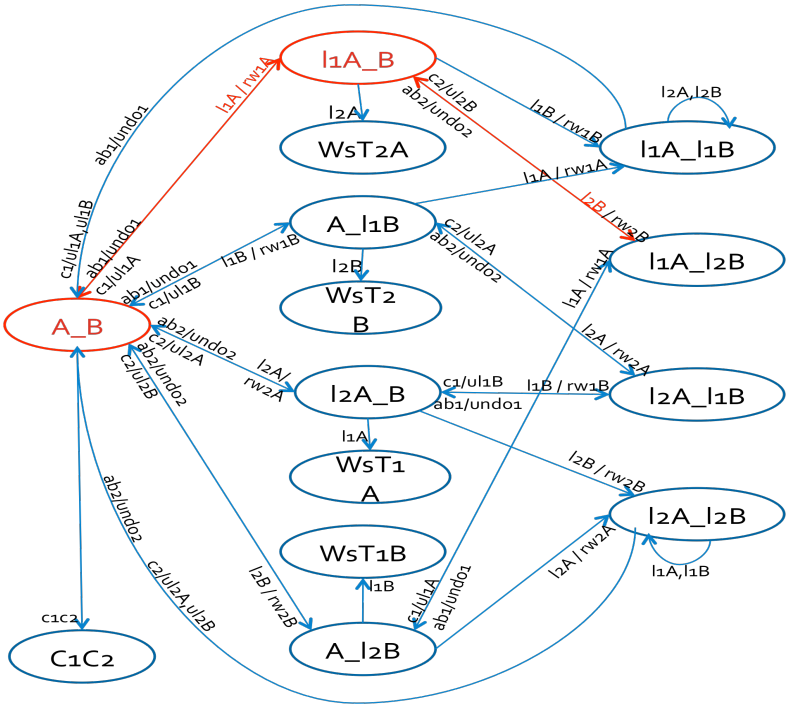


-> Input: 1.3 <-

-> State: 1.3 <-

p.evento = l2B

p.azione = null



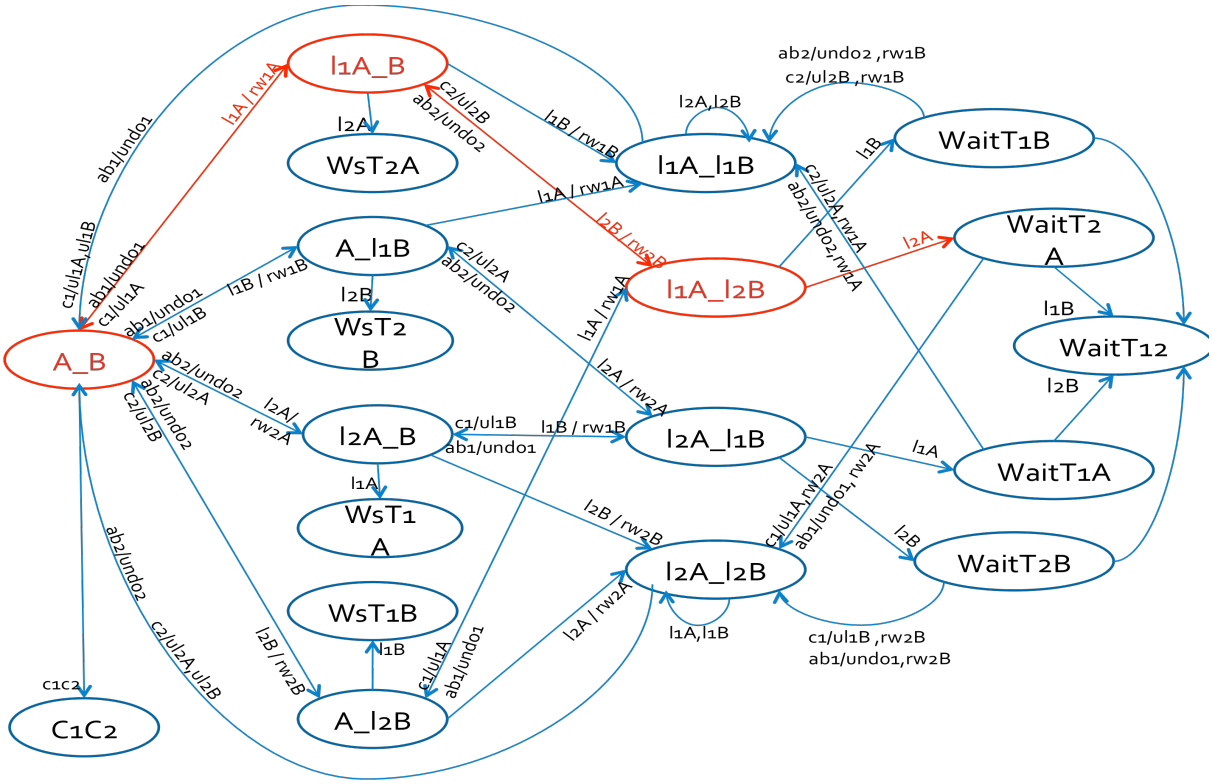
-> Input: 1.4 <-

-> State: 1.4 <-

p.stato = l1A_l2B

p.evento = l2A

p.azione = rw2B

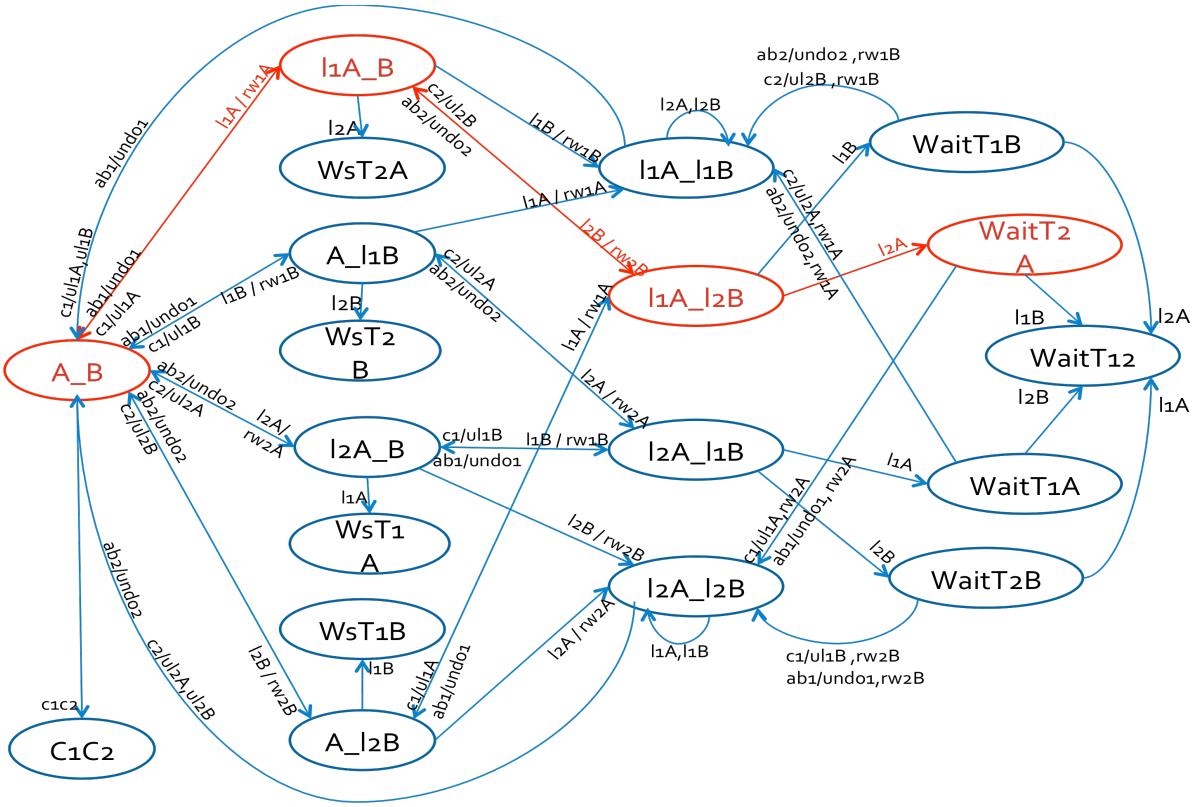


-> Input: 1.5 <-

-> State: 1.5 <-

p.stato = WT2A

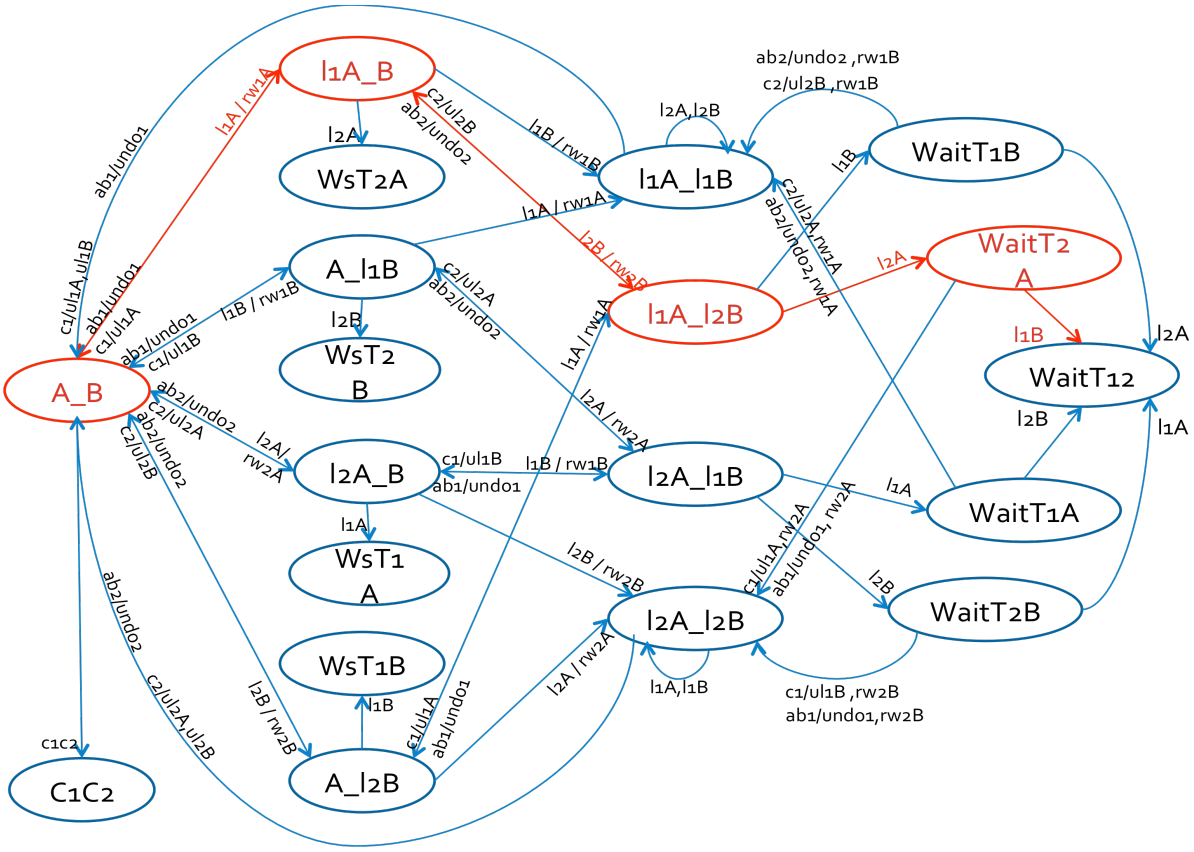
p.azione = null



-> Input: 1.6 <-

-> State: 1.6 <-

p.evento = l1B



-> Input: 1.7 <-

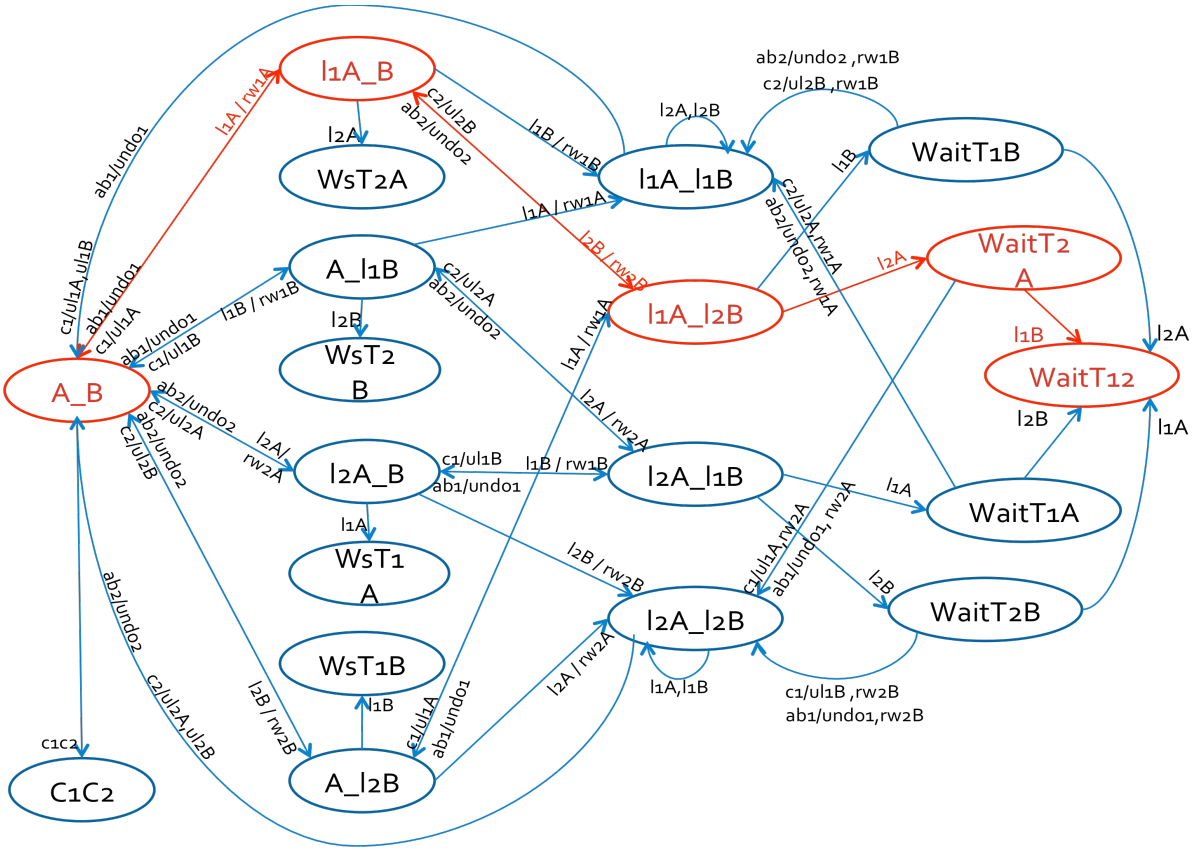
-- Loop starts here

-> State: 1.7 <-

p.stato = WT1T2

-> Input: 1.8 <-

-> State: 1.8 <-



Verifica proprietà ACID

Durabilità

Abbiamo già visto come questa proprietà non è possibile verificare essendo dovuto ad una caratteristica fisica non replicabile del dbms.

Atomicità

Per quanto riguarda questa proprietà sono andato a verificare appunto l'atomicità delle transazioni, quindi le transazioni o non iniziano o se iniziano devo essere completate. Sono andato a verificare PRE → ASS_Atom dove ASS_Atom è G (F (p.stato = C1C2 | p.stato = A_B))).

Quindi o la transazione comincia e finisce oppure non comincia proprio. L'output che ho elencato di seguito è proprio quello che mi aspettavo.

-- specification ((((((((G (p.azione = rw1A ->X (G p.azione != rw1A)) & G (p.azione = rw1B -> X (G p.azione != rw1B))) & G (p.azione = rw2A -> X (G p.azione != rw2A))) & G (p.azione = rw2B -> X (G p.azione != rw2B))) & G ((p.stato = I1A_I1B & (p.evento = I2A | p.evento = I2B)) -> X (G (p.stato = I1A_I1B -> (p.evento != I2A & p.evento != I2B))))) & G ((p.stato = I2A_I2B & (p.evento = I1A | p.evento = I1B)) -> X (G (p.stato = I2A_I2B -> (p.evento != I1A & p.evento != I1B))))) & G ((p.evento = c1 | p.evento = ab1) -> X (G (p.evento !=I1A & p.evento != I1B)))) & G ((p.evento = c2 | p.evento = ab2) -> X (G (p.evento != I2A & p.evento != I2B)))) -> G (F (p.stato = C1C2 | p.stato = A_B)))is **false**

-- as demonstrated by the following execution sequence

Trace Description: LTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

p.stato = A_B

p.evento = I1A

p.azione = null

-> Input: 1.2 <-

-> State: 1.2 <-

p.stato = I1A_B

p.azione = rw1A

-> Input: 1.3 <-

-> State: 1.3 <-

p.evento = I2B

p.azione = null

-> Input: 1.4 <-

-> State: 1.4 <-
p.stato = I1A_I2B
p.evento = I2A
p.azione = rw2B
-> Input: 1.5 <-
-> State: 1.5 <-
p.stato = WT2A
p.azione = null
-> Input: 1.6 <-
-> State: 1.6 <-
p.evento = I1B
-> Input: 1.7 <-
-- Loop starts here
-> State: 1.7 <-
p.stato = WT1T2
-> Input: 1.8 <-
-> State: 1.8 <-

Ovviamente soffrendo tale protocollo di situazioni di stallo tale proprietà non è garantita ma come ho già detto esistono per gestire lo stallo che quindi possono garantire anche tale proprietà

Consistency

Per quanto riguarda tale proprietà ho verificato che l'effetto della transazione sulle risorse fosse coerente. Quindi devo evitare situazioni di lettura sporca e mantenere la coerenza nella lettura e scrittura dei dati. La condizione imposta è: PRE → ASS_Cons dove ASS_Cons è la seguente:

(G ((p.azione = rw1A & F p.azione = rw2A) -> G X p.azione != rw1A) &
G ((p.azione = rw2A & F p.azione = rw1A) -> G X p.azione != rw2A) &
G ((p.azione = rw1B & F p.azione = rw2B) -> G X p.azione != rw1B) &
G ((p.azione = rw2B & F p.azione = rw1B) -> G X p.azione != rw2B))

Tale condizione serve a controllare che non si leggano valori sporchi e che quindi due interrogazioni sulla stessa base di dati possano dare valori differenti in output. Valgono le stesse osservazioni fatte per questa proprietà nel caso di protocollo 2PL con lock esclusivi. Questo è l'output che ottengo:

-- specification ((((((((G (p.azione = rw1A -> X (G p.azione != rw1A)) & G (p.azione = rw1B -> X (G p.azione != rw1B))) & G (p.azione = rw2A -> X (G p.azione != rw2A))) & G (p.azione = rw2B -> X (G p.azione != rw2B))) & G ((p.stato = l1A_l1B & (p.evento = l2A | p.evento = l2B)) -> X (G (p.stato = l1A_l1B -> (p.evento != l2A & p.evento != l2B))))) & G ((p.stato = l2A_l2B & (p.evento = l1A | p.evento = l1B)) -> X (G (p.stato = l2A_l2B -> (p.evento != l1A & p.evento != l1B))))) & G ((p.evento = c1 | p.evento = ab1) -> X (G (p.evento != l1A & p.evento != l1B)))) & G ((p.evento = c2 | p.evento = ab2) -> X (G (p.evento != l2A & p.evento != l2B)))) -> (((G ((p.azione = rw1A & F p.azione = rw2A) -> G (X p.azione != rw1A)) & G ((p.azione = rw2A & F p.azione = rw1A) -> G (X p.azione != rw2A))) & G ((p.azione = rw1B & F p.azione = rw2B) -> G (X p.azione != rw1B))) & G ((p.azione = rw2B & F p.azione = rw1B) -> G (X p.azione != rw2B)))) is **true.**

Isolamento

Per garantire questa proprietà non ci possono essere azioni di scrittura/lettura in contemporanea. Quindi vado a verificare PRE → ASS_Isol dove ASS_Isol è la seguente:

```
( G ( p.azione = rw1A & G (p.evento !=c1 & p.evento != ab1) -> G X p.azione
!= rw2A )
        &
G ( p.azione = rw1B & G (p.evento !=c1 & p.evento != ab1) -> G X p.azione !=
rw2B )
        &
G ( p.azione = rw2A & G (p.evento !=c2 & p.evento != ab2) -> G X p.azione !=
rw2A )
        &
G ( p.azione = rw2B & G (p.evento !=c2 & p.evento != ab2) -> G X p.azione !=
rw2B ))
```

In poche parole vado a controllare che se ottengo il lock su una risorsa r (e lo utilizzo eseguendo un'operazione di lettura/scrittura rw) e non eseguo commit o abort allora non ci sarà nessun altra transazione che legge o scrive su r. L'output che ottengo è il seguente:

```
-- specification (((((((( G (p.azione = rw1A ->  X ( G p.azione != rw1A)) &  G
(p.azione = rw1B ->  X ( G p.azione != rw1B))) &  G (p.azione = rw2A ->  X ( G
p.azione != rw2A))) &  G (p.azione = rw2B ->  X ( G p.azione != rw2B))) &  G
((p.stato = l1A_l1B & (p.evento = l2A | p.evento = l2B)) ->  X ( G (p.stato =
l1A_l1B -> (p.evento != l2A & p.evento != l2B))))) &  G ((p.stato = l2A_l2B &
(p.evento = l1A | p.evento = l1B)) ->  X ( G (p.stato = l2A_l2B -> (p.evento !=
```

l1A & p.evento != l1B))))) & G ((p.evento = c1 | p.evento = ab1) -> X (G (p.evento !=l1A & p.evento != l1B)))) & G ((p.evento = c2 | p.evento = ab2) -> X (G (p.evento != l2A & p.evento != l2B)))) -> (((G ((p.azione = rw1A & G (p.evento != c1 & p.evento != ab1)) -> G (X p.azione != rw2A)) & G ((p.azione = rw1B & G (p.evento != c1 & p.evento != ab1)) -> G (X p.azione != rw2B))) & G ((p.azione = rw2A & G (p.evento != c2 & p.evento != ab2)) -> G (X p.azione != rw2A))) & G ((p.azione = rw2B & G (p.evento != c2 & p.evento != ab2)) -> G (X p.azione != rw2B)))) is **true**