UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

Corso di Tecniche Automatiche per la Correttezza del Software (a.a. 2016/17)

Program Repair as a Game

Prof. Salvatore La Torre

Giovanni De Costanzo

Overview

- Introduzione
- Preliminari
- Program Repair
- Esempi
- Conclusioni

Introduzione

Il *Model Checking* rivela se un programma aderisce alle sue specifiche e, in caso contrario, fornisce un controesempio in cui la specifica è violata.

Tuttavia, localizzare un errore non significa che sia facile correggerlo.

“ Conoscere se un programma ha un bug è buono. Conoscere la sua posizione è ancora meglio, ma solo una correzione ci soddisfa veramente.”

Introduzione

Fault Localization

Anche se è disponibile una traccia di errore, potrebbe essere difficile trovare l'errore contenuto all'interno del sistema.

Diversi approcci sono stati presentati per alleviare questo problema:

1. Rendere le tracce più semplici da capire

2. Aiutare l'utente a capire l'errore considerando delle tracce molto simili, alcune di errore altre di successo

3. Approccio mirato a localizzare l'errore, basato sulla teoria della diagnosi (originariamente sviluppato per sistemi fisici)

Introduzione

Program Repair

- A partire da un insieme di statement sospetti, cerca una modifica del programma che soddisfi la specifica
- Non è necessario fornire una specifica dell’intero programma ma soltanto per la parte da riparare
- La modifica introdotta è limitata solo alla parte interessata dall’errore, dunque non vengono alterate la struttura e la logica del programma

Introduzione

Program Repair

- La specifica del programma è fornita in LTL
- Il gioco del programma è un gioco LTL che cattura le possibili riparazioni, costruendo alcuni valori ‘*unknown*’
- Attenzione focalizzata su programmi a stati finiti
- Il *fault model* presentato assume che sia RHS che LHS di un assignment possano essere sbagliati

Introduzione

Program Repair

- Il gioco avviene tra:
  - L'ambiente esterno che fornisce gli input
  - Il sistema, che fornisce i valori corretti per le espressioni sconosciute
- Il gioco è vinto se per ogni sequenza in input il sistema è in grado di fornire una sequenza di valori per le espressioni sconosciute, tali che la specifica è soddisfatta
- Una *strategia vincente* fissa il corretto valore per l'espressione sconosciuta e quindi corrisponde ad una riparazione

Introduzione

Program Repair

- Per trovare una strategia viene costruito un gioco di Büchi che è il prodotto tra il gioco del programma e l'automa non deterministico standard per la specifica
- Se il gioco prodotto è vinto, allora lo è anche il gioco del programma (non vale il contrario)
- Invece di implementare la riparazione corrispondente alla strategia a stati finiti, aggiungendo degli stati al programma, viene introdotta una strategia memoryless

Preliminari

Definizione di *gioco*

Un *gioco G* su AP è una tupla $(S, s_0, I, C, \delta, \lambda, F)$, dove:

- **S** è un insieme finito di stati
- $\mathbf{s_0} \in S$ è lo stato iniziale
- ***I*** e ***C*** sono insiemi finiti di input esterni e di scelte del sistema
- $\boldsymbol{\delta} : S \times I \times C \rightharpoonup S$ è la funzione di transizione parziale
- $\boldsymbol{\lambda} : S \rightarrow 2^{AP}$ è la funzione di labeling
- $\boldsymbol{F} \subseteq S^{\omega}$ è la condizione di vincita, un insieme di sequenze infinite di stati

La sfida è quella di trovare i corretti valori per *C* tali che *F* è soddisfatta

Preliminari

Definizione di *strategia*

Dato un gioco $G = (S, s_0, I, C, \delta, \lambda, F)$,

una *strategia* (a stati finiti) è una tupla $\sigma = (Q, q_0, \mu)$ dove:

- **Q** è un insieme finito di stati
- $\mathbf{q_0} \in Q$ è lo stato iniziale
- $\boldsymbol{\mu} : Q \times S \times I \rightarrow 2^{C \times Q}$ è la funzione di movimento

Preliminari

Definizione di *strategia*

- Abbiamo bisogno di strategie non deterministiche per garantire la massima libertà quando tentiamo di convertire una strategia a stati finiti in una *strategia memoryless*

- Affinché una strategia sia vinta, deve derivarne una partita vincente a partire da qualunque scelta non deterministica di essa

Preliminari

Definizione di *partita*

Una *partita* su G in accordo ad una strategia σ è una sequenza finita o infinita

$\pi = q_0 s_0 \xrightarrow{i_0 c_0} q_1 s_1 \xrightarrow{i_1 c_1} \dots$

tale che $(c_i, q_{i+1}) \in \mu(q_i, s_i, i_i), s_{i+1} = \delta(s_i, i_i, c_i)$ e:

- o la partita è infinita
- oppure esiste *n* tale che:
  - $\mu(q_n, s_n, i_n) = \varnothing$
  - oppure $\delta(s_n, i_n, c_n)$ non è definito, il che significa che il gioco è finito

Una *partita* è vincente se è infinita e $s_0 s_1 \dots \in F$

Preliminari

Strategia memoryless

- Una *strategia memoryless* è una strategia a stati finiti con un solo stato, definita come una funzione $\sigma : Q \times I \rightarrow 2^C$
- Una partita di una strategia memoryless è definita come una sequenza $s_0 \xrightarrow{i_0 c_0} s_1 \xrightarrow{i_1 c_1}$ … dove lo stato dell'automa strategia è escluso

Preliminari

Safety Game

- Un *gioco safety* ha una condizione $F = \{s_0 s_1 \ldots \mid \forall\, i : s_i \in A\}$ per qualche $A \subseteq S$.

- La condizione di vincita di un gioco LTL è l'insieme di sequenze che soddisfano una formula LTL ϕ. In questo caso scriveremo ϕ per F.

- I giochi di Büchi sono definiti da un insieme $B \subseteq Q$, e richiedono che una partita visiti il vincolo di Büchi B infinite volte.

- Possiamo convertire una formula LTL ϕ su AP in un gioco di Büchi $A = (Q, q_0, 2^{AP}, C, \delta, \lambda, B)$ tale che $l(A)$ è l'insieme di parole che soddisfano ϕ. Le scelte del sistema modellano il non determinismo dell'automa.

Program Repair

Costruzione di un gioco

Supponiamo di avere un programma che non rispetta la sua specifica LTL ϕ e di avere localizzato la variabile o la linea responsabile dell'errore.

Un programma è un gioco LTL $K = (S, s_0, I, \{c\}, \delta, \lambda, \phi)$ ***dove:***

- L'insieme delle scelte di sistema è un singleton
- La condizione di accettazione è la specifica LTL
- e è l'espressione nella quale LHS o RHS sono errate

Program Repair

Costruzione di un gioco

Trasformiamo K in un gioco del programma G ‘liberando’ il valore dell’espressione e, cioè:

- Se Ω è il dominio di e, cambiamo la scelta del sistema in $C' = C \times \Omega$ e lasciamo che la seconda componente della scelta del sistema definisca il valore di e

- Se troviamo una strategia memoryless vincente per G, allora abbiamo determinato una funzione dallo stato del programma al valore corretto per LHS o RHS

Program Repair

Strategie a stati finiti

Dati due giochi $G = (S, s_0, I_G, C_G, \delta_G, \lambda_G, F_G)$ e $A = (Q, q_0, 2^{AP}, C_A, \delta_A, \lambda_A, F_A)$, definiamo il *gioco prodotto* come $G \triangleright A = (S \times Q, (s_0, q_0), I_G, C_G \times C_A, \delta, \lambda, F)$ dove:

- $\delta((s, q), i_G, (c_G, c_A)) = (\delta_G(s, i_G, c_G), \delta_A(q, \lambda_G(s), c_A))$
- $\lambda(s,q) = \lambda_G(s)$
- $F = \{ (s_0, q_0), (s_1, q_1), \ldots \mid s_0, s_1, \ldots \in F_G \text{ e } q_0, q_1, \ldots \in F_A \}$

Intuitivamente, l'output di G viene dato in input ad A, e le condizioni di vincita sono congiunte

Program Repair

Strategie a stati finiti

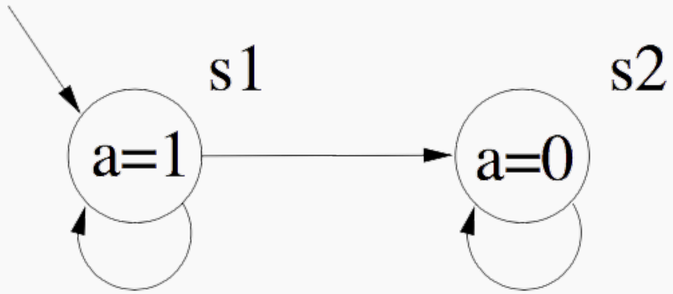
Lemma 1: Dati due giochi G ed A, $L(G \triangleright A) = L(G) \cap I(A)$

Lemma 2: Siano G ed A dei giochi. Se esiste una strategia memoryless per $G \triangleright A$ allora esiste una strategia a stati finiti vincente per G tale che per ogni partita π di G, in accordo a σ, $\lambda(\pi) \in L(G)$ e $\lambda(\pi) \in I(A)$

Teorema 1: Sia $G = (S, s_0, I, C, \delta, \lambda, \phi)$ un gioco LTL, e sia G' come G ma con condizione di vincita S^{ω}, e sia A un gioco di Büchi con $I(A) = L(G)$. Se esiste una strategia memoryless vincente per il gioco di Büchi $G' \triangleright A$ allora esiste una strategia a stati finiti vincente per G

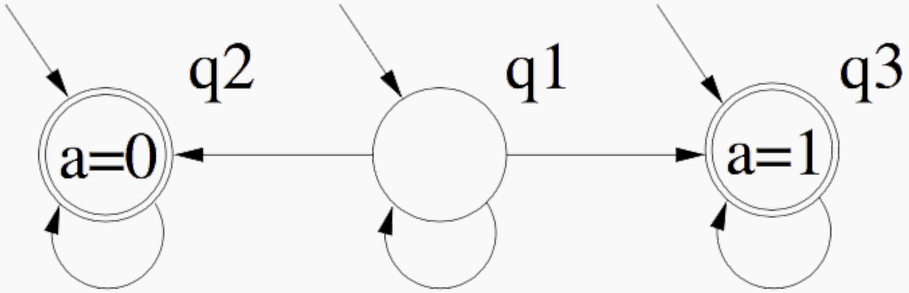
Program Repair

Strategie a stati finiti



(a)

(a) Gioco nel quale l'ambiente può assegnare il valore per la variabile *a*



(b)

(b) Automa per
F G(a = 1) V F G(a = 0)

Program Repair

Strategie a stati finiti

In generale, la trasformazione di una formula LTL in un automa deterministico richiede un blowup doppiamente esponenziale.

Per evitare questo blowup possiamo

- usare un’euristica per ridurre il numero di stati non deterministici nell’automa
- oppure utilizzare un insieme ristretto di LTL [8]

Program Repair

NP-Completezza delle strategie memoryless

Dato un grafo diretto *G* e due nodi *v* e *w*, calcolare se ci sono percorsi di nodi disgiunti da *v* a *w* e viceversa, è un problema NP-Completo.

Supponiamo di costruire un gioco *G'* basato su *G*. La condizione di accettazione è che *v* e *w* siano visitati infinite volte. Poiché l'esistenza di una strategia memoryless di *G'* implica l'esistenza di due percorsi di nodi disgiunti da *v* a *w* e viceversa, possiamo dedurre il seguente teorema.

Teorema 2: *Decidere se un gioco con una strategia vincente definita da un automa di Büchi ha una strategia memoryless vincente è NP-Completo*

Program Repair

Euristiche per strategie memoryless

Data una strategia memoryless per il gioco prodotto:

- Costruiamo una strategia comune a tutti gli stati dell’automa, candidandola come nostra strategia memoryless
- Successivamente calcoliamo se la strategia è vincente (che non è necessariamente il nostro caso)

Program Repair

Euristiche per strategie memoryless

Vogliamo risolvere il gioco $G = (S, s_0, I_G, C_G, \delta_G, \lambda_G, \phi)$ costruendo un automa $A = (Q, q_0, 2^{AP}, C_A, \delta_A, \lambda_A, B)$ con $I(A) = L(\phi)$ e successivamente risolvere $G' \triangleright A$, dove G' è derivato da G facendo accettare tutte le partite.

Sia $\sigma : (S \times Q) \times I_G \rightarrow 2^{C_G \times C_A}$ una strategia vincente per $G' \triangleright A$

Definiamo la nostra strategia memoryless candidata $\tau' = (s, i_G)$ prendendo gli spostamenti comuni a tutti gli stati vincenti raggiungibili dell'automa strategia.

Se τ' è vincente allora anche σ lo è, ma non vale il contrario.

Per controllare se τ' è vincente, costruiamo un gioco G' a partire da G, restringendo la funzione di transizione per aderire a τ': $\delta' = \{ (s, i, c, s') \in \delta \mid c \in \tau'(s, i) \}$

Program Repair

Euristiche per strategie memoryless

La costruzione G’ potrebbe introdurre stati senza un successore, quindi proviamo ad evitarli calcolando l’insieme degli stati vincenti *W’*:

- Se troviamo che $s_0 \notin W'$ allora non possiamo evitare di visitare uno stato pozzo, e quindi rinunciamo a trovare una riparazione
- Se troviamo che $s_0 \in W'$, otteniamo la nostra strategia memoryless finale τ restringendo τ' a W', assicurandoci che un gioco che inizia in W' vi rimanga e non visiti mai uno stato pozzo: $\tau(s, i_G) = \tau'(s, i_G) \cap (W' \times I_G)$

Program Repair

Euristiche per strategie memoryless

Giungiamo così al seguente teorema:

Teorema 3: Se $s_0 \in W'$ allora τ è una strategia vincente di G.

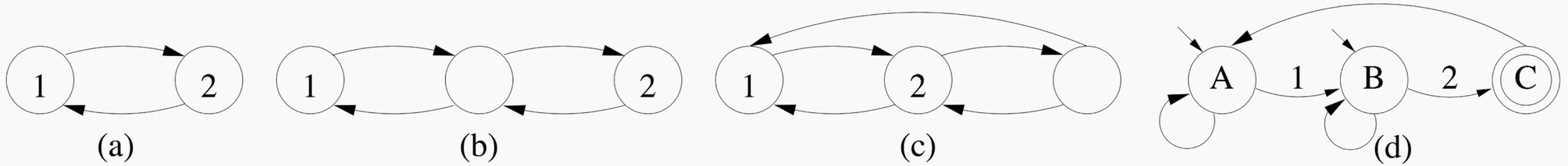
Dimostrazione (idea):

Presa una partita $\pi = s_0 \xrightarrow{i_{G0} c_{G0}} s_1 \xrightarrow{i_{G1} c_{G1}}$ su G in accordo a τ, notiamo che è anche una partita in accordo a τ'.

Costruiamo una partita $\pi' = (s_0, q_0) \xrightarrow{i_{G0}(c_{G0} c_{A0})} (s_1, q_1)$ in accordo a σ.
Siccome σ è vincente, allora π' è una partita vincente di $G \triangleright A$ e quindi $\lambda(s_0 s_1 \ldots) \in I(A) = L(\phi)$, il che significa che π è vincente in accordo alla specifica ϕ di G.

Program Repair

Euristiche per strategie memoryless



Le figure *(a)*, *(b)*, *(c)* mostrano tre giochi in cui la condizione di vincita richiede che gli stati 1 e 2 siano visitati infinite volte. L'automa di Büchi in figura (d) definisce la condizione di vincita.

- Per la fig. *(a)*, le strategie per gli stati A, B e C coincidono ed esiste quindi una strategia memoryless
- Per la fig. *(b)* non esiste una una strategia memoryless
- Per la fig. *(c)* esiste una strategia memoryless ma non corrisponde all'intersezione di tutte le strategie per gli stati A, B, C. (Le strategie sono contraddittorie per lo stato sulla destra)

Program Repair

Estrazione di una riparazione

Viene mostrato un metodo simbolico per estrarre uno statement di riparazione da una strategia memoryless:

- La strategia viene resa deterministica trovando le corrette assegnazioni alle scelte del sistema che possono essere usate nelle locazioni sospette

- Per un dato stato del programma, la strategia proposta può consentire più assegnazioni, dandoci spazio per l’ottimizzazione

Program Repair

Estrazione di una riparazione

Potremmo non volere che la riparazione dipenda da certe variabili del programma (es. perché sono fuori dallo scope delle componenti che devono essere riparate).

In questo caso, possiamo quantificare tali variabili dalla strategia e la sua regione di vincita, e verificare che venga ancora fornita una risposta valida per tutte le combinazioni di stati ed input.

Program Repair

Estrazione di una riparazione

Per ogni assegnazione alle variabili delle scelte di sistema, calcoliamo un insieme $P_j \subseteq S \times I$ per cui l'assegnazione è parte della strategia proposta.

Possiamo usare questi insiemi P_j per suggerire una riparazione: "`if P0 then assign 0 else if P1 then ...`"

Sfruttiamo il fatto che gli insiemi *Pj* possono sovrapporsi per costruire dei nuovi insiemi *Aj*, che sono molto più facili da esprimere. Dobbiamo fare in modo da coprire ancora tutti gli stati vincenti e raggiungibili utilizzando gli insiemi *Aj*.

Program Repair

Estrazione di una riparazione

A_j è ottenuto da P_j aggiungendo o rimuovendo gli stati al di fuori di un *care set*. L'insieme consiste:

- di tutti gli stati che non possono essere coperti da A_j perché non sono in P_j
- di tutti gli stati che devono essere coperti da *Aj* poiché non sono coperti da A_k, con $k < j$, ne da P_k con $k > j$

Per ottenere la riparazione suggerita, sostituiamo infine `Pj` con un'espressione per *Aj*.

Program Repair

Complessità

La complessità dell'algoritmo è polinomiale nel numero degli stati del sistema, ed esponenziale nella lunghezza della formula, come per il model checking.

Un'implementazione simbolica richiede un numero quadratico di calcoli di preimage per calcolare la regione di vincita di un gioco di Büchi.

Esempi

Locking

```
  int got_lock = 0;
  do{
1   if (*) {
2     lock();
3     got_lock++; }
4   if (got_lock != 0) {
5     unlock();}
6   got_lock--;
7 } while(*)

  void lock() {
11  assert(L = 0);
12  L = 1; }

  void unlock(){
21  assert(L = 1);
22  L = 0; }
```

Un programma astratto che realizza semplici operazioni di lock

L'errore è alla riga 6, lo statement `got_lock--` andrebbe messo all'interno dello scope del precedente `if`

Esempi

Locking

```
  int got_lock = 0;
  do{
1   if (*) {
2     lock();
3     got_lock++; }
4   if (got_lock != 0) {
5     unlock();}
6   got_lock--;
7 } while(*)

  void lock() {
11  assert(L = 0);
12  L = 1; }

  void unlock(){
21  assert(L = 1);
22  L = 0; }
```

L'algoritmo localizza dei possibili errori alle righe 1, 6, 7 e suggerisce tre riparazioni:

1. Impostare la condizione dell'if alla prima riga ad 1
2. Impostare la condizione del while alla riga 7 a false
3. Impostare `got_lock = 0` alla riga 6

L'ultimo suggerimento è soddisfacente: è una correzione per il programma indipendentemente dalle condizioni nell'if e nel while.

Esempi

MinMax

```
1   int least = input1;
2   int most = input1;
3   if(most < input2){
4      most = input2; }
5   if(most < input3){
6      most = input3;}
7   if(least > input2){
8      most = input2; }
9   if(least > input3){
10     least = input3;}
11  assert (least <= most);
```

Questo programma assegna i valori minini e massimi di tre valori in input rispettivamente a due variabili `least` e `most`.

L'errore è localizzato alla riga 8, dove `input2` viene assegnato a `most` invece che a `least`.

Esempi

MinMax

```
1   int least = input1;
2   int most = input1;
3   if(most < input2){
4      most = input2; }
5   if(most < input3){
6      most = input3;}
7   if(least > input2){
8      most = input2; }
9   if(least > input3){
10     least = input3;}
11  assert (least <= most);
```

L'algoritmo localizza due possibili errori alle righe 7 ed 8 e fornisce le possibili riparazioni.

- Alla riga 7 suggerisce di impostare la condizione dell'if a false
- Alla riga 8 suggerisce due possibili riparazioni: cambiare l'LHS dell'assegnazione con `least` oppure cambiare l'RHS con `input1` oppure `input3`

Tutte le riparazioni suggerite sono valide per l'asserzione `(least <= most)` che è ovviamente troppo debole.

Esempi

MinMax

L’asserzione viene quindi modificata e resa più precisa:

```
(least <= input1) && (least <= input2) && (least <= input3) &&
(most >= input1) && (most >= input2) && (most >= input3)
```

Con questa specifica troviamo la riparazione corretta: cambiare l’LHS alla riga 8 da `most` a `least`.

Esempi

Sezione Critica

Process A

```
1   flag1A = true;
2   turn1B = false;
3   while(flag1B && turn1B);
4   x = x && y;
5   flag1A = false;
6   if(turn1B){
7     flag2A = true;
8     turn2B = true;
9     while(flag2B && turn2B);
10    y = false;
11    flag2A = false;}
12  goto 1;
```

Process B

```
1   flag1B = true;
2   turn1B = false;
3   while(flag1A && !turn1B);
4   x = x && y;
5   flag2B = true;
6   turn2B = false;
7   while(flag2A && !turn2B);
8   y = !y;
9   x = x || y;
10  flag2B = false;
11  flag1B = false;
12  goto 1;
```

L'esempio mostra due processi che condividono le variabili `flag` e `turn`, usate per evitare l'accesso concorrente alle variabili `x` e `y`.

Un processo arbitro (non mostrato) assegna in modo non deterministico il controllo sia ad A che a B, e registra le sue scelte in una variabile `arbiter`.

Esempi

L’algoritmo presentato non è in grado di trovare una strategia per il gioco prodotto del programma.

Per risolvere il problema viene modificato manualmente il processo arbitro, in modo da scambiare i processi infinite volte.

Liberando `turn1B` alla riga 2 del processo A con dominio {false, true}, arriviamo alla risposta corretta, `turn1B = true`.

Ulteriore Esempio

Dal codice al gioco

```
   unsigned int got_lock = 0;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
4:     got_lock++;
     }
     ...
5:   if (got_lock != 0) {
6:     unlock();
     }
7:   got_lock--;
     ...
   }
```

```
lock()
lock:    {LOCK:=1;}
unlock()
unlock:  {LOCK:=0;}
```

Specifications

P1: do not acquire a lock twice

P2: do not call unlock without holding the lock

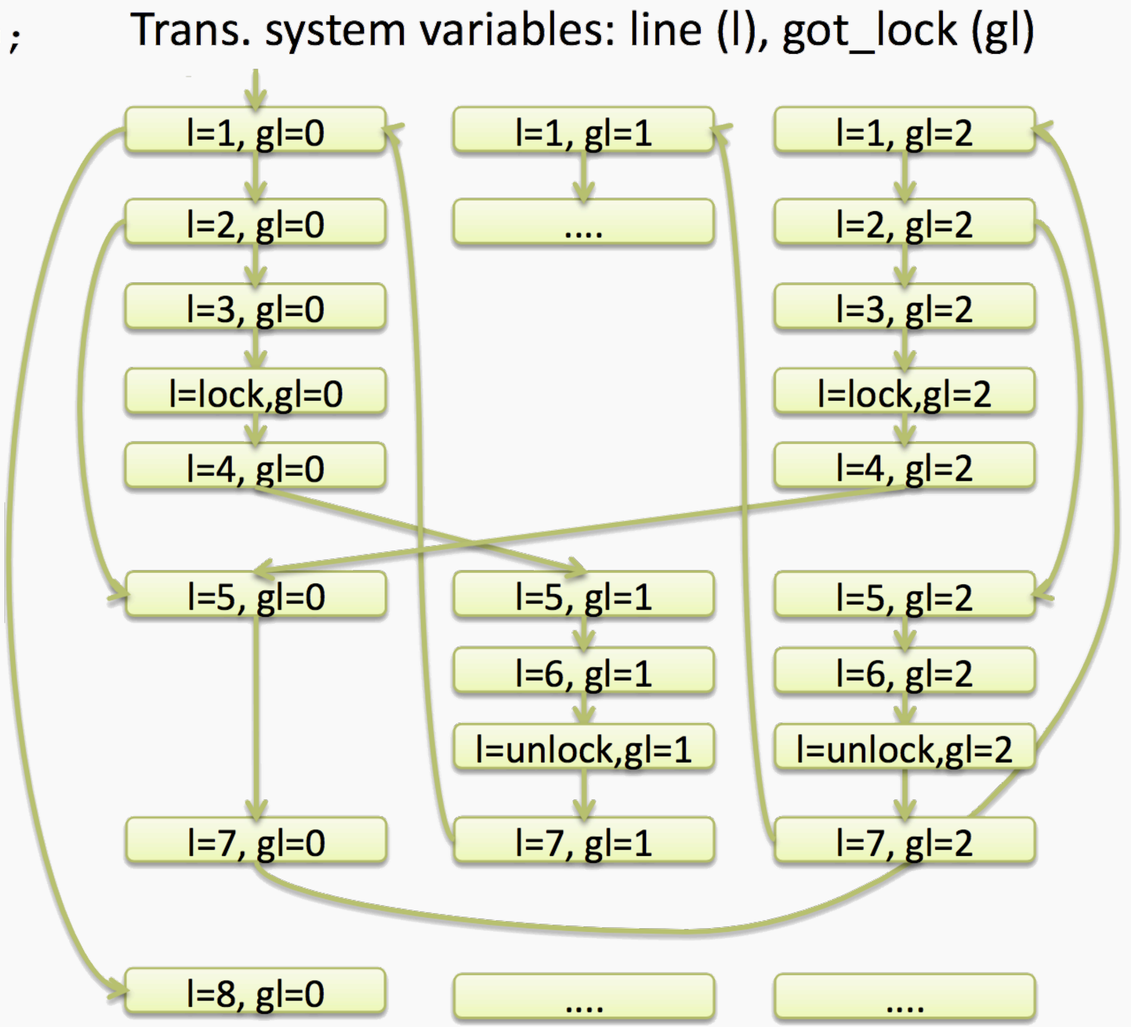
P1: always(line=lock **implies**
next(line!=lock **w-until** line=unlock))

P2: (line!=unlock **w-until** line=lock)) **and**
always(line=unlock **implies**
next(line!=unlock **w-until** line=lock))

Dal codice al gioco

Trasformazione del codice sorgente nel sistema di transizione

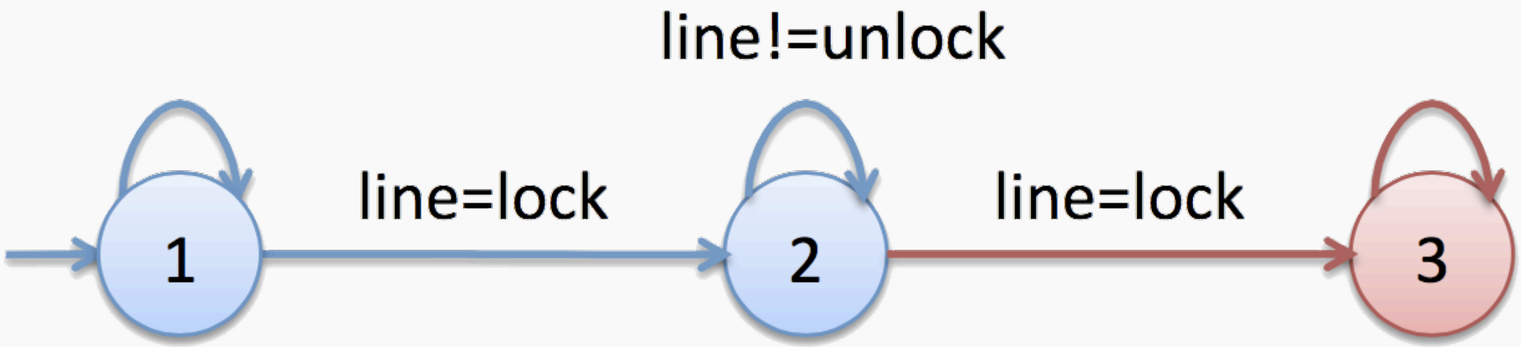
```
    int[0,1,2] got_lock = 0;
    ...
1:  while(*) {
      ...
2:    if (*) {
3:      lock();
lock:     {LOCK:=1;}
4:      got_lock++;
      }
      ...
5:    if (got_lock != 0) {
6:      unlock();
unlock:  {LOCK:=0;}
      }
7:    got_lock--;
      ...
    }
8:
```



Dal codice al gioco

Automa di Büchi non deterministico per la proprietà P1 della specifica

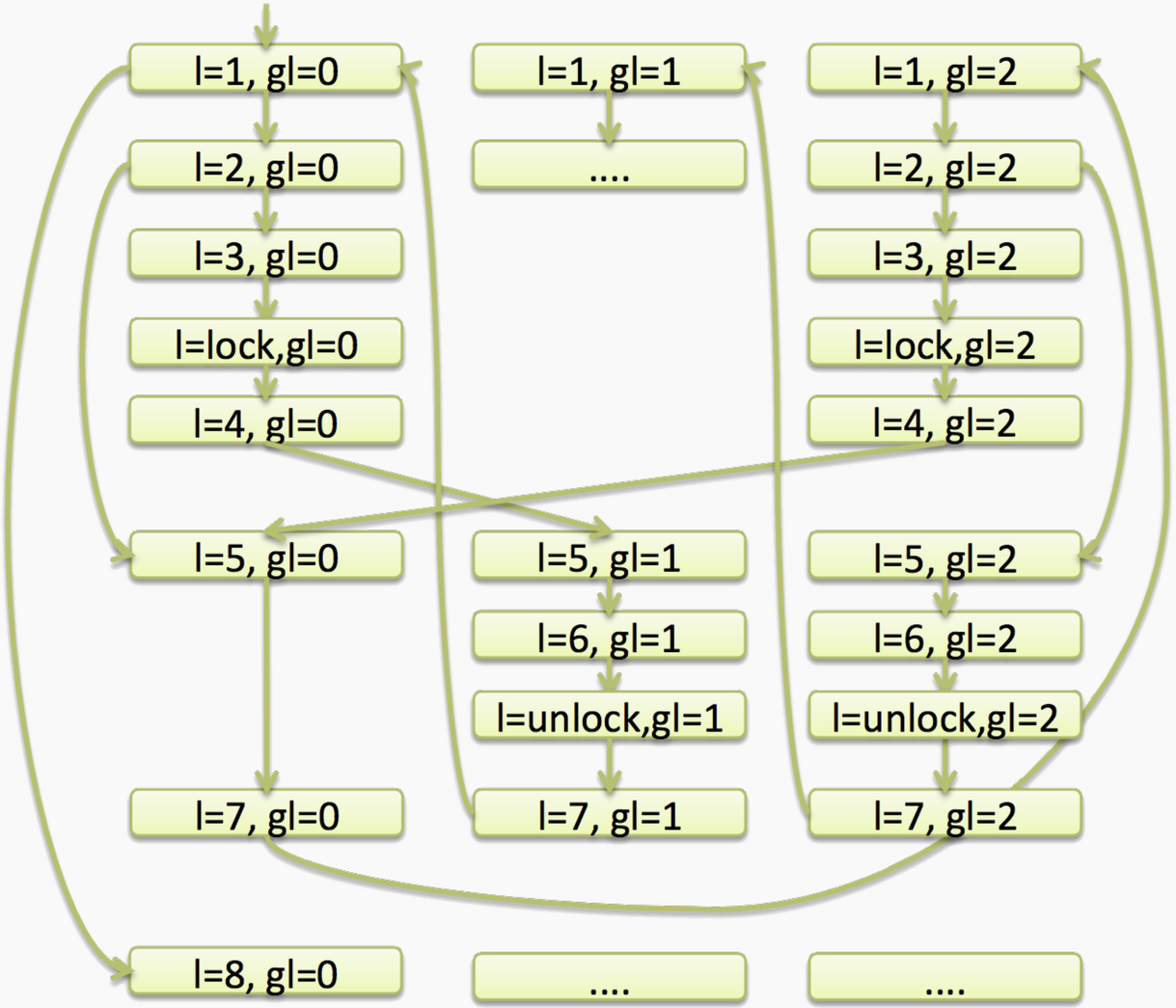
P1: always(line=lock **implies**
next(line!=lock **w-unti** line=unlock))

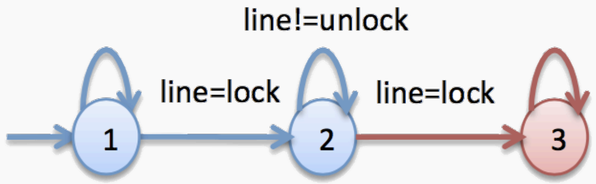


s1: non-deterministic choice **s2**: no edge with line=unlock

Dal codice al gioco

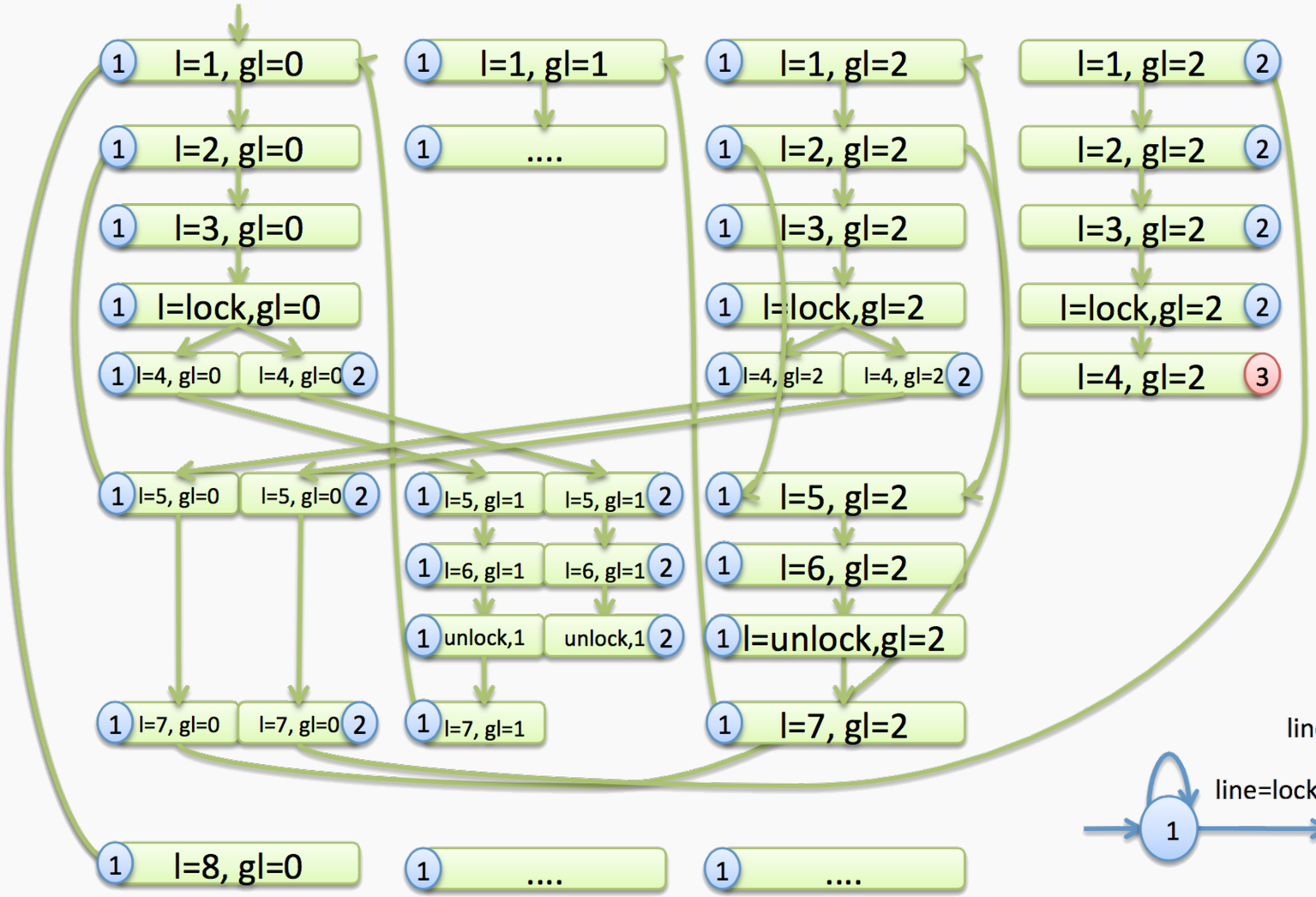
Prodotto

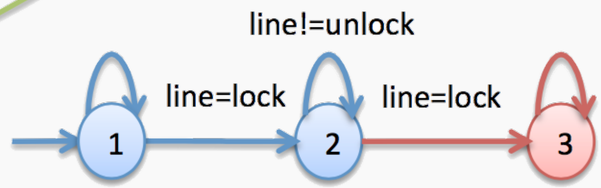




Dal codice al gioco

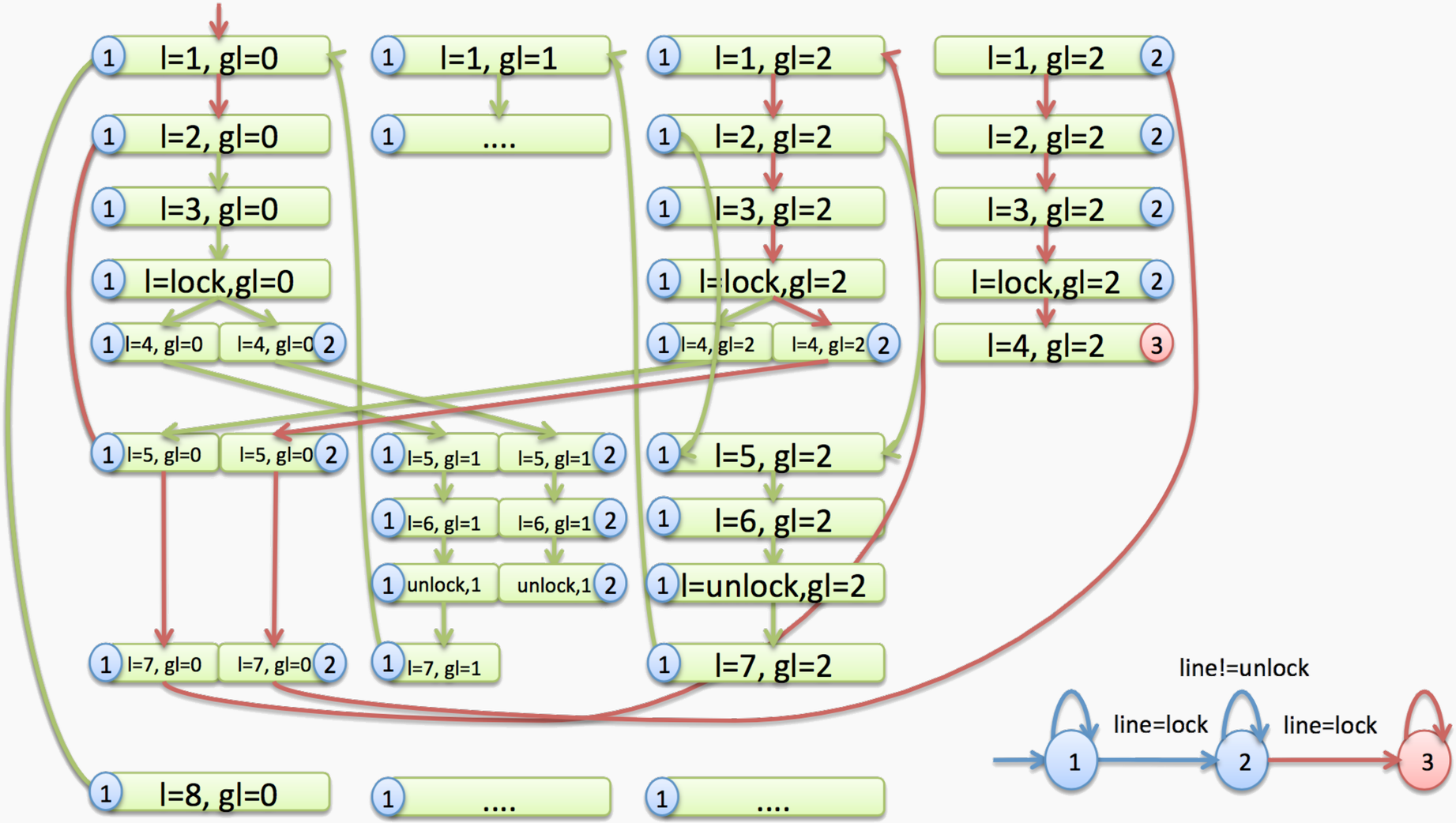
Prodotto





Dal codice al gioco

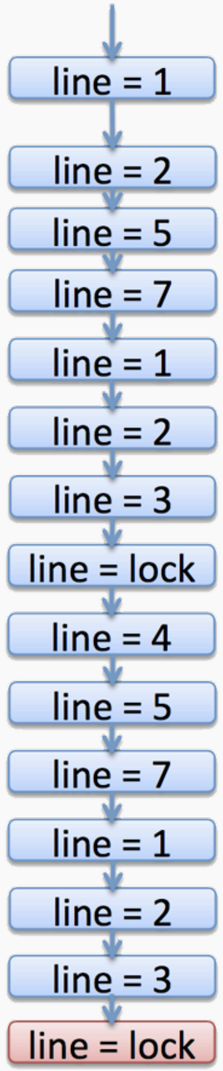
Traccia di violazione della specifica



Dal codice al gioco

Esempio di esecuzione con violazione della specifica

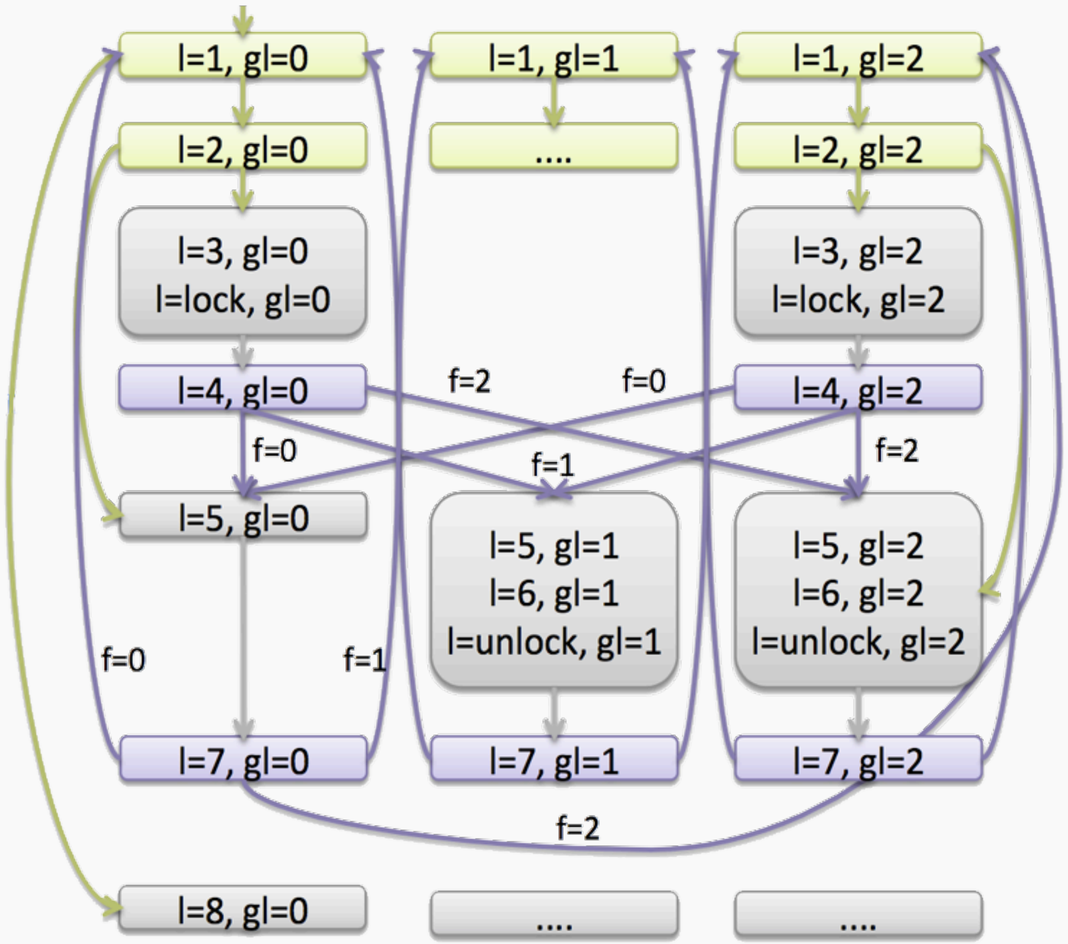
```
    int[0,1,2] got_lock = 0;
    ...
1:  while(*) {
      ...
2:    if (*) {
3:      lock();
lock:    {LOCK:=1;}
4:      got_lock++;
      }
      ...
5:    if (got_lock != 0) {
6:      unlock();
unlock: {LOCK:=0;}
      }
7:    got_lock--;
      ...
    }
8:
```



Dal codice al gioco

Freedom degli assignment sospetti e definizione del gioco

```
   int[0,1,2] got_lock = 0;
   int[0,1,2] freedom;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
lock:   {LOCK:=1;}
4:     got_lock:=freedom;
     }
     ...
5:   if (got_lock != 0){
6:     unlock();
unlock: {LOCK:=0;}
     }
7:   got_lock:=freedom;
     ...
   }
8:
```

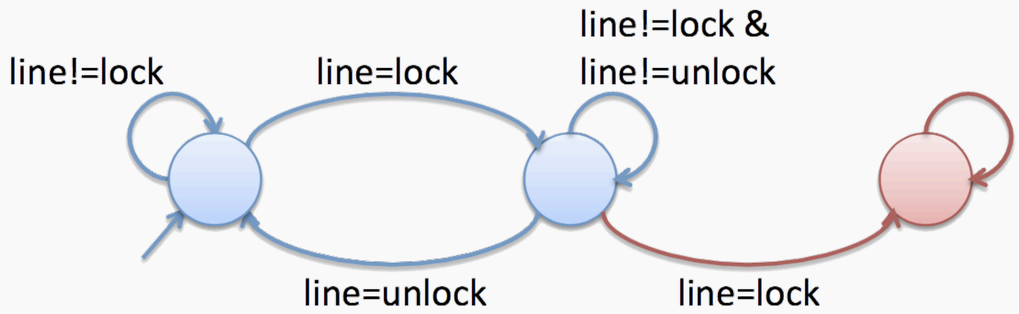


Dal codice al gioco

Automa deterministico per la specifica

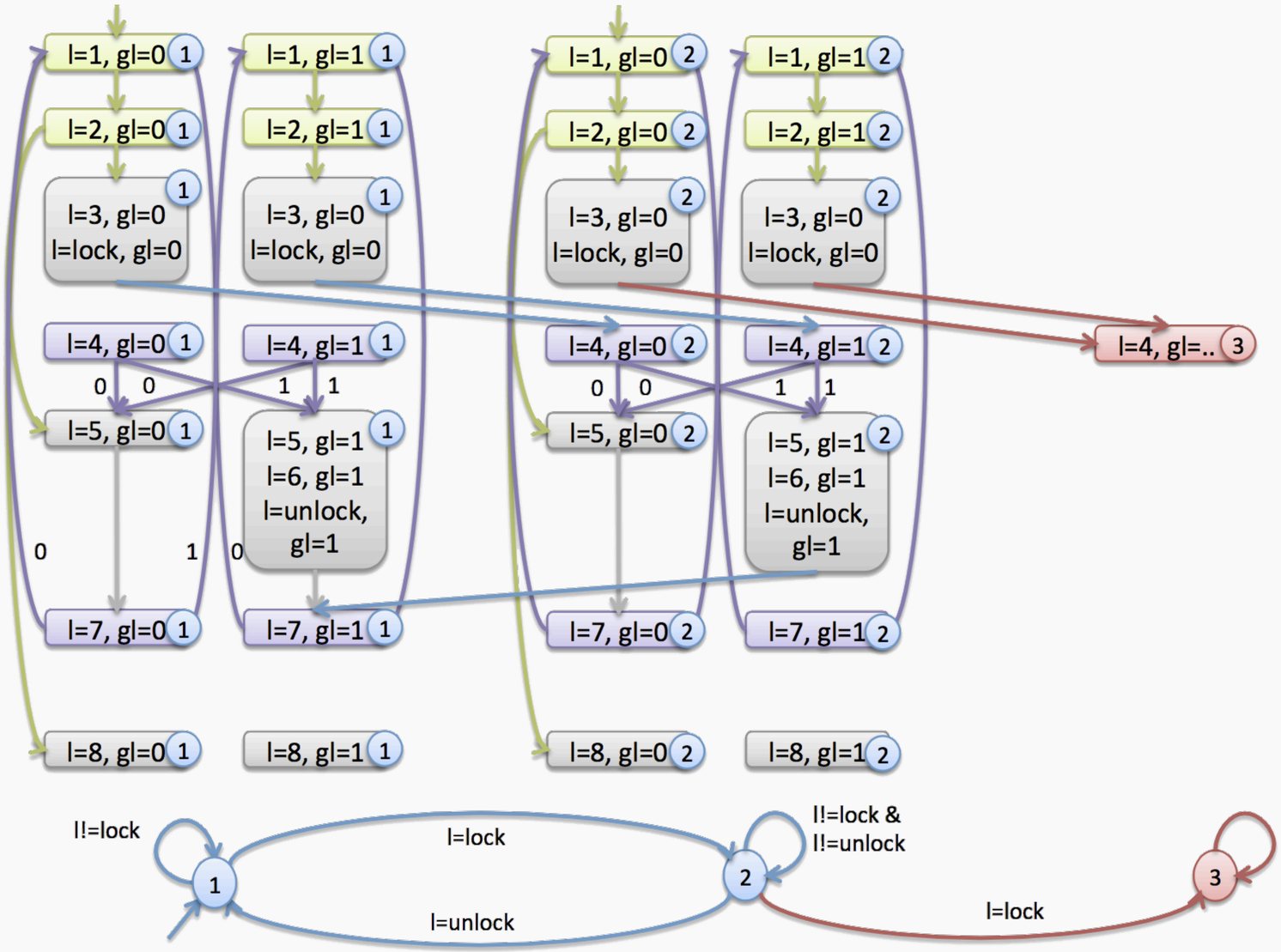
P1: always(line=lock **implies**

next(line!=lock **w-until** line=unlock))



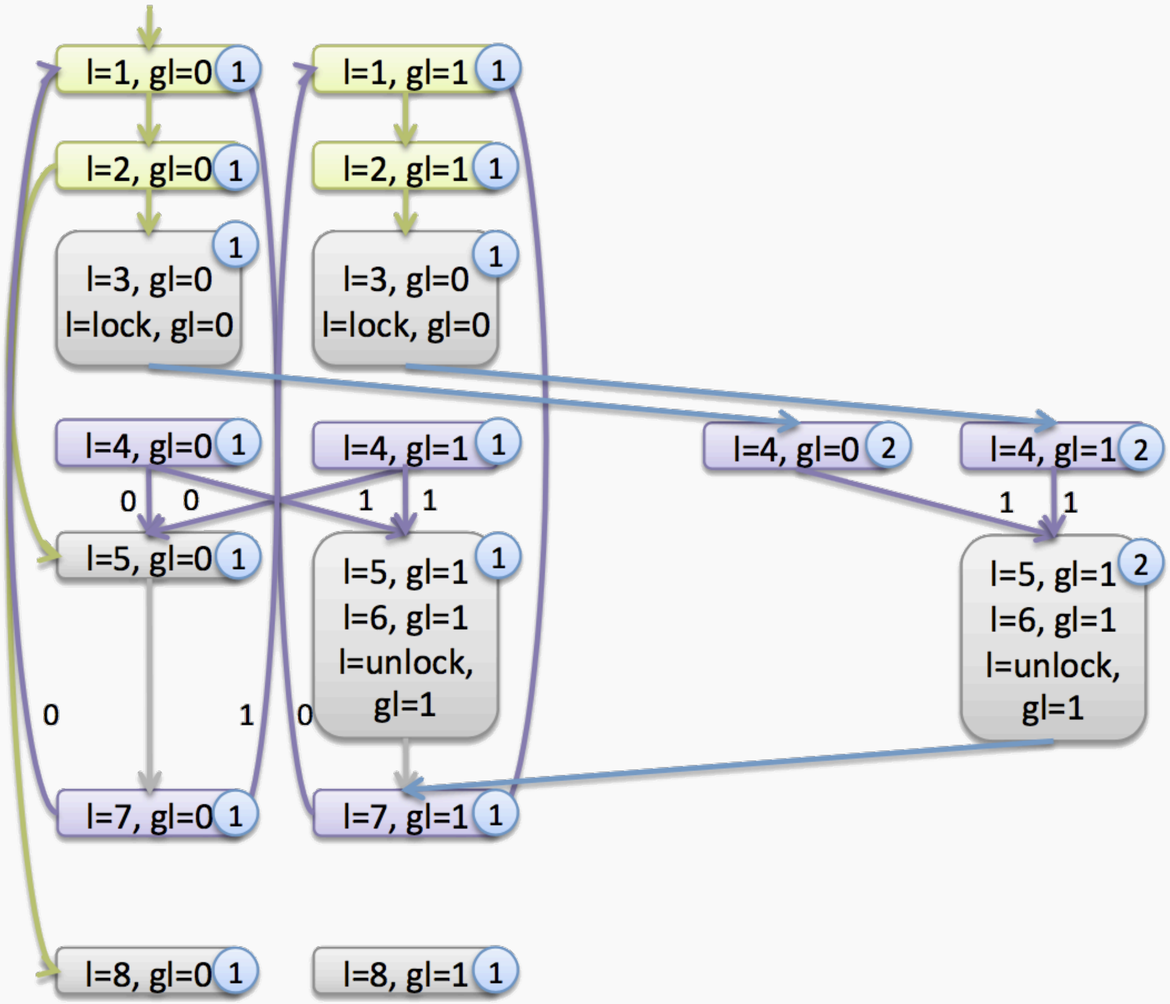
Dal codice al gioco

Prodotto



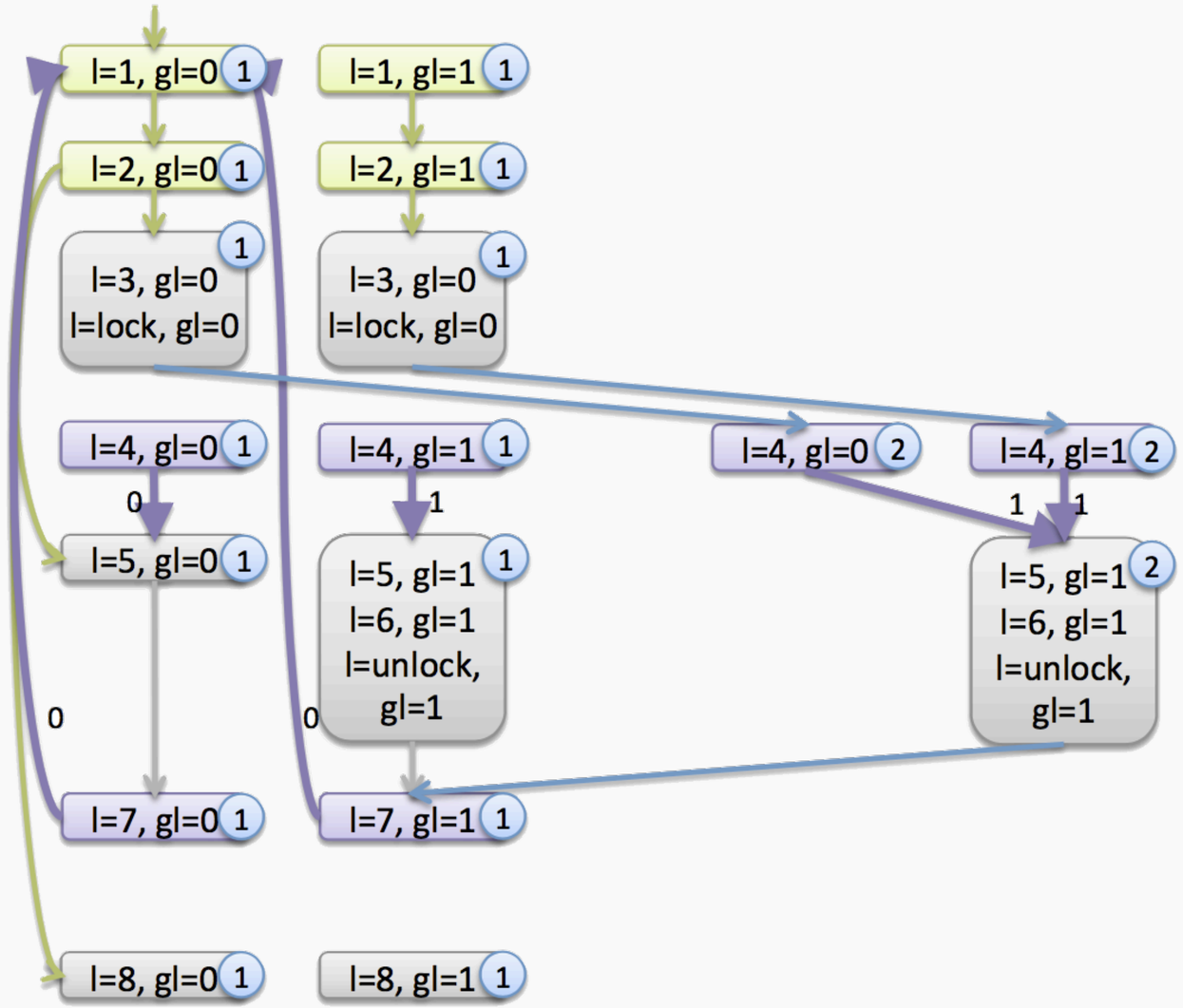
Dal codice al gioco

Estrazione degli stati vincenti



Dal codice al gioco

Strategia vincente



Strategy to Repair:

if (l=4 & gl=0 & s=1) freedom:=0
if (l=4 & gl=1 & s=1) freedom:=1
if (l=4 & gl=0 & s=0) freedom:=1
if (l=7 & gl=0 & s=1) freedom:=0
if (l=7 & gl=1 & s=1) freedom:=0
..
freedom := f(l,gl,s)
if (line=4)
    freedom := (gl=1) | (s=2)
if (line=7)
    freedom := 0

Dal codice al gioco

Programma corretto

```
   unsigned int got_lock = 0;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
4:     got_lock = 1;
     }
     ...
5:   if (got_lock != 0) {
6:     unlock();
     }
7:   got_lock = 0;
     ...
   }
```

```
lock()
lock:   {LOCK=1;}
unlock()
unlock: {LOCK=0;}
```

Prestazioni

Per comparare l'efficienza dell'algoritmo di riparazione con quella del model checking, è stato introdotto un errore in una versione a 16 bit di un semplice processore DLX-style.

Su una macchina Linux con CPU a 2.8GHz e 2GB di RAM:

- L'esecuzione del **model checking** richiede **230 secondi** per trovare che la proprietà non vale
- L'**algoritmo di repair** trova una riparazione in **200 secondi**

Tutte le esecuzioni usano circa 1.2GB di RAM

Conclusioni

- Abbiamo considerato che: dato un sospetto di fault, riparare un programma consiste nel farlo aderire alla propria specifica.
- Abbiamo costruito il prodotto di un gioco corrispondente al programma danneggiato e l'automa che riflette la specifica.
- Se il prodotto del gioco ha una strategia vincente, allora possiamo riparare il programma. Tuttavia una strategia potrebbe non esistere per il prodotto, anche se una riparazione esiste. Questo a causa del non determinismo dell'automa.

Conclusioni

- Per raggirare il problema, possiamo rendere deterministico l'automa, ma il costo sarebbe esponenziale; inoltre per molte combinazioni di programmi e specifiche, il non determinismo non è un problema.

- Una strategia vincente a stati finiti corrisponde ad una riparazione che introduce nuovi stati. Abbiamo respinto questa possibilità di cambiare la logica del programma, quindi cerchiamo una strategia memoryless.

- Abbiamo visto che decidere se una strategia memoryless esiste è un problema NP-Completo.

Conclusioni

- E’ stata presentata un’euristica che congiunge le strategie per i differenti stati dell’automa. L’euristica descritta trova una riparazione efficiente per una strategia memoryless data.

- La complessità dell’algoritmo è comparabile a quella del model checking.

Riferimenti

[1] B. Jobstmann, A. Griesmayer, R. Bloem, Program repair as a game, in: K. Etessami, S. K. Rajamani (Eds.), 17th Conference on Computer Aided Verification (CAV'05), Springer-Verlag, 2005, pp. 226–238, LNCS 3576.

[2] Stefan Staber Barbara Jobstmann Roderick Bloem. Finding and Fixing Faults, 2008.

[3] M. Stumptner and F. Wotawa. A model-based approach to software debugging. In Proceedings on the Seventh International Workshop on Principles of Diagnosis, 1996.

[4] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), pages 445–459, Grenoble, France, April 2002. LNCS 2280.

Riferimenti

[5] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In 30th Symposium on Principles of Programming Languages (POPL 2003), pages 97–105, January 2003.

[6] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In Model Checking of Software: 10th International SPIN Workshop, pages 121–135. Springer-Verlag, May 2003. LNCS 2648.

[7] R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. In *Symposium on Logic in Computer Science (LICS'01)*, pages 291–302, 2001.

[8] Program Synthesis is a Game, Barbara Jobstmann [Slides]