Model Checking e Partial Order Reduction

Angelo Montanari (e Ilaria Sambarino)

Università degli Studi di Udine

1. Introduzione
2. Generazione del grafo degli stati ridotto
3. Indipendenza e invisibilità
4. Partial order reduction per LTL_{-X}
5. Esempio: programma per la gestione della mutua esclusione
6. Calcolo degli ample set
7. SPIN (cenni)

I sistemi asincroni e il problema dell'esplosione degli stati

- I modelli dei sistemi asincroni basati sull'interleaving consentono di ordinare eventi/transizioni concorrenti **indipendenti** in maniera arbitraria.
- Per evitare di trascurare un ordinamento particolare, gli eventi vengono alternati in tutti i modi possibili.
- Ovviamente l'ordinamento di eventi indipendenti è irrilevante.
- Prendere in considerazione tutti i possibili ordinamenti di eventi concorrenti è una delle potenziali cause del problema dell'esplosione degli stati.
- Se consideriamo n transizioni del tutto indipendenti, ci sono $n!$ differenti ordinamenti (sequenze) e 2^n stati differenti (uno stato per ogni sottoinsieme di transizioni).
- Se la specifica non distingue fra tali sequenze, conviene considerarne solo una (**rappresentante**). In tal modo, il numero di stati di riduce a $n + 1$.

Partial Order Reduction: che cos'è? - 1

- La **partial order reduction** è una tecnica che mira a ridurre la dimensione dello spazio degli stati esplorato dagli algoritmi di model checking.
- Sfrutta la commutatività delle transizioni indipendenti eseguite in maniera concorrente, la cui esecuzione porta sempre nello stesso stato, indipendentemente dall'ordine secondo il quale sono eseguite.
- Si presta in modo particolare al trattamento dei sistemi asincroni nei quali non viene richiesta la sincronizzazione delle transizioni concorrenti (tale sincronizzazione è, invece, necessaria per l'esecuzione della comunicazione nei sistemi sincroni).

Partial Order Reduction: che cos'è? - 2

- Il metodo consiste nel costruire un **grafo degli stati ridotto** in sostituzione del grafo degli stati completo, che non viene mai costruito (il grafo ridotto **non** viene costruito a partire dal grafo completo).

- I comportamenti catturati dal grafo ridotto sono un sottoinsieme dei comportamenti modellati dal grafo completo, ma se un comportamento non è presente nel grafo ridotto, allora nel grafo ridotto è presente un comportamento ad esso equivalente.

- La tecnica di partial order reduction può essere meglio definita come **model checking basato sui rappresentanti**, in quanto la verifica del modello viene effettuata utilizzando rappresentanti delle classi di equivalenza dei comportamenti.

Strutture di Kripke modificate

- La tecnica che vedremo si basa sulla **relazione di dipendenza** che intercorre tra le transizioni di un sistema.

  Risulta utile modificare leggermente la definizione di struttura di Kripke.

- Un **sistema di transizioni di stato** è una quadrupla (S, T, S_0, L) dove:
  - l'insieme degli stati S, l'insieme degli stati iniziali S_0 e la funzione di etichettatura L sono definiti come per le strutture di Kripke;
  - T è un insieme di transizioni: per ogni $\alpha \in T, \alpha \subseteq S \times S$.

Definizioni di base

- Una transizione α è **abilitata** in uno stato s se esiste uno stato s' tale che $\alpha(s, s')$. Altrimenti, α è *disabilitata* in s.
- L'insieme delle transizioni abilitate in s è denotato con $enabled(s)$.
- Una transizione α è **deterministica** se per ogni stato s esiste al più uno stato s' tale che $\alpha(s, s')$ (considereremo solo transizioni deterministiche).
- Un **cammino** π che inizia in uno stato s_0 è una sequenza finita o infinita $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ tale che, per ogni $i, \alpha_i(s_i, s_{i+1})$.
- Un prefisso di un cammino è a sua volta un cammino.
- Se π è finito, allora la lunghezza di π, denotata con $|\pi|$, è il numero di transizioni in esso contenute.

Grafo degli stati ridotto

- L'obiettivo è ridurre il numero degli stati che vengono presi in esame nel processo di model checking, preservando la correttezza e completezza della procedura di verifica della proprietà specificata.
- Generiamo un **grafo degli stati ridotto** utilizzando l'algoritmo DFS. A tale grafo verrà applicato l'algoritmo di model checking che
  - terminerà con *risposta positiva* se la proprietà risulta soddisfatta nel grafo degli stati originario (completo);
  - altrimenti, produrrà un *controesempio*.
- Tale soluzione riduce i tempi necessari per la costruzione del grafo e produce un grafo che utilizza meno memoria, rendendo l'algoritmo di model checking più efficiente.

L'algoritmo DFS

```
1   hash(s0);
2   set on_stack(s0);
3   expand_state(s0);
4
5   procedure expand_state(s)
6       work_set(s) := ample(s);
7       while work_set(s) in not empty do
8           let α ∈ work_set(s);
9           work_set(s) := work_set(s) \ {α};
10          s' := α(s);
11          if new(s') then
12              hash(s');
13              set on_stack(s');
14              expand_state(s');
15          end if;
16          create_edge(s, α, s');
17      end while;
18      set completed(s);
19  end procedure
```

Osservazione: ogni stato, quando raggiunto (riga 1), viene memorizzato in una hash table.

Richieste per *ample*(s)

Algoritmo DFS modificato: alla riga 6, si calcola l'insieme *ample*(s) anziché l'insieme *enabled*(s) (*ample*(s) è un sottoinsieme di *enabled*(s)).

Dobbiamo trovare un modo per calcolare *ample*(s) per ogni stato s in modo da soddisfare le seguenti tre condizioni:

1. utilizzare *ample*(s) al posto di *enabled*(s) deve comunque garantire la presenza nel grafo ridotto di un numero di comportamenti che sia sufficiente ad assicurare la correttezza dei risultati;
2. utilizzare *ample*(s) al posto di *enabled*(s) deve garantire che il numero di stati del grafo ridotto sia significativamente più piccolo;
3. il costo computazionale del calcolo di *ample*(s) deve essere ragionevolmente basso.

Indipendenza

Definizione.

Una relazione di **indipendenza** $I \subseteq T \times T$ *è una relazione simmetrica ed antiriflessiva che, per ogni stato* $s \in S$ *e ogni* $(\alpha, \beta) \in I$*, soddisfa le seguenti due condizioni:*

- *enabledness - se* $\alpha, \beta \in enabled(s)$*, allora* $\alpha \in enabled(\beta(s))$*;*
- *commutativity - se* $\alpha, \beta \in enabled(s)$*, allora* $\alpha(\beta(s)) = \beta(\alpha(s))$*.*

La relazione di **dipendenza** D è il complemento di I, ossia $D = (T \times T) \setminus I$.

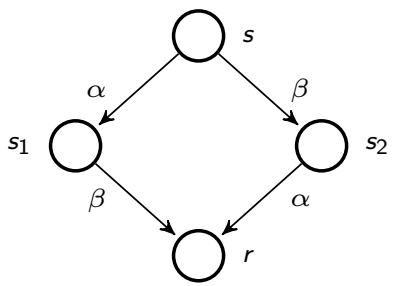
Quando risulta difficile verificare se due transizioni α e β sono indipendenti, assumere che esse siano dipendenti preserva la correttezza della riduzione proposta (assumiamo la dipendenza quale default).

Potenziali problemi

La commutatività suggerisce una possibile riduzione del grafo degli stati: date due transizioni indipendenti α e β, quale che sia l'ordine secondo il quale esse vengono eseguite, viene sempre raggiunto il medesimo stato r.

In realtà, non è possibile, in generale, eliminare uno dei due possibili ordinamenti, in quanto

- **problema 1** - la proprietà che vogliamo verificare potrebbe essere sensibile alla scelta (implicita) tra s_1 e s_2, non solo a r ed s;
- **problema 2** - gli stati s_1 e s_2 potrebbero, accanto ad r, avere altri successori e tali successori risulterebbero non più raggiungibili qualora uno tra s_1 e s_2 venisse eliminato.



Invisibilità

Definizione.

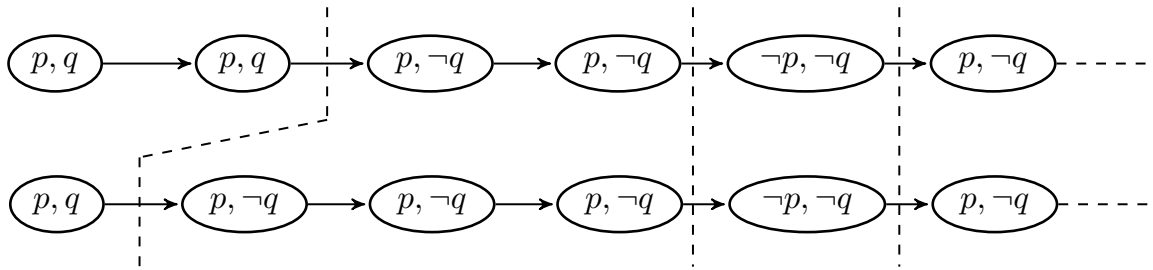
Sia AP l'insieme delle proposizioni atomiche e sia $L : S \to 2^{AP}$ la funzione che etichetta ogni stato con l'insieme delle proposizioni atomiche in esso vere.

Una transizione $\alpha \in T$ è detta **invisibile** *rispetto ad un insieme $AP' \subseteq AP$ se, per ogni coppia di stati $s, s' \in S$ tali che $s' = \alpha(s), L(s) \cap AP' = L(s') \cap AP'$.*

In altri termini, una transizione si dice invisibile se la sua esecuzione, in un qualunque stato, non cambia il valore delle variabili proposizionali in AP'. Una transizione si dice visibile se non è invisibile.

Stuttering e cammini infiniti

Una nozione strettamente collegata alla precedente è quella di **stuttering**.



Definizione.

Due cammini infiniti $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ *e* $\rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$ *si dicono* **equivalenti rispetto allo stuttering**, *denotato* $\sigma \sim_{st} \rho$, *se esistono due sequenze infinite di interi positivi* $0 = i_0 < i_1 < i_2 < \ldots$ *e* $0 = j_0 < j_1 < j_2 < \ldots$ *tali che, per ogni* $k \geq 0$,

$$L(s_{i_k}) = L(s_{i_k+1}) = \ldots = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = \ldots = L(r_{j_{k+1}-1})$$

Stuttering e formule di LTL

Definizione.

Una formula di LTL $\mathbf{A}f$ *si dice* **invariante rispetto allo stuttering** *se e solo se per ogni coppia di cammini* π *e* π' *tali che* $\pi \sim_{st} \pi'$,

$$\pi \models f \text{ se e solo se } \pi' \models f.$$

Denotiamo il frammento della logica LTL privo dell'operatore next con LTL_{-X}.

Teorema.

Ogni formula di LTL_{-X} *è invariante rispetto allo stuttering.*

Teorema.

Ogni formula di LTL invariante rispetto allo stuttering può essere espressa in LTL_{-X}.

Stuttering e strutture

Definizione.

Due **strutture** M *e* M' *si dicono* **equivalenti rispetto allo stuttering** *se e solo se*

- M *e* M' *hanno lo stesso insieme di stati iniziali;*
- *per ogni cammino* σ *in* M *che parte da uno stato iniziale* s *di* M *esiste un cammino* σ' *in* M' *che parte dallo stesso stato iniziale* s *tale che* $\sigma \sim_{st} \sigma'$, *e*
- *per ogni cammino* σ' *in* M' *che parte da uno stato iniziale* s *di* M' *esiste un cammino* σ *in* M *che parte dallo stesso stato iniziale* s *tale che* $\sigma' \sim_{st} \sigma$.

Corollario.

Siano M *e* M' *due strutture equivalenti rispetto allo stuttering. Allora, per ogni formula* $\mathbf{A}f$ *in* LTL_{-X} *e per ogni stato iniziale* $s \in S_0$, $M, s \models \mathbf{A}f$ *se e solo se* $M', s \models \mathbf{A}f$.

Questo risultato verrà sfruttato nel seguito: la partial order reduction genera un grafo degli stati ridotto che è equivalente rispetto allo stuttering al grafo degli stati completo.

Algoritmo DFS e ample set

- Se una specifica (formula) è invariante rispetto a stuttering, commutatività e invisibilità consentono di evitare di generare alcuni degli stati.
- Nel seguito descriveremo un modo sistematico per selezionare un **ample set** di transizioni per ogni dato stato.
- Gli ample set verranno utilizzati dall'algoritmo DFS per costruire un grafo degli stati ridotto in modo tale che per ogni cammino non considerato, ne esista un altro, ad esso equivalente rispetto allo stuttering, preso in considerazione.
- Uno stato s si dice **fully expanded** se $\mathit{ample}(s) = \mathit{enabled}(s)$. In questo caso, tutti i successori dello stato s verranno esplorati dall'algoritmo DFS.

La condizione **C0**

Introduciamo ora quattro condizioni per selezionare $ample(s) \subseteq enabled(s)$ in modo tale che la soddisfacibilità della specifica LTL$_{-X}$ venga preservata.

La riduzione dipenderà dall'insieme delle proposizioni AP' che compaiono nella formula in LTL$_{-X}$.

Condizione C0

Se uno stato s ha almeno un successore, allora il grafo degli stati ridotto contiene almeno un successore per s e viceversa (se uno stato s ha almeno un successore nel grafo degli stati ridotto, allora s ha almeno un successore nel grafo completo):

$$ample(s) = \emptyset \text{ se e solo se } enabled(s) = \emptyset$$

La condizione **C1**

Condizione C1

Per ogni cammino nel grafo degli stati completo che parte da s, vale la seguente condizione: una transizione che dipende da una transizione in $ample(s)$ non può essere eseguita senza che una transizione in $ample(s)$ venga eseguita prima di essa.

Dovremo ovviamente essere in grado di verificare la validità di **C1** senza dover costruire il grafo completo.

Da tale condizione, segue immediatamente il seguente lemma.

Lemma.

Le transizioni in $enabled(s) \setminus ample(s)$ *sono tutte indipendenti da quelle in* $ample(s)$.

Dimostrazione.

Sia $\gamma \in enabled(s) \setminus ample(s)$. Supponiamo che $(\gamma, \delta) \in D$, per qualche $\delta \in ample(s)$. Dato che γ è abilitata in s, nel grafo degli stati completo esiste un cammino che parte da γ. Ma allora una transizione dipendente da qualche transizione in $ample(s)$ viene eseguita prima di una transizione in $ample(s)$, violando la Condizione **C1**. □

La correttezza della riduzione DFS

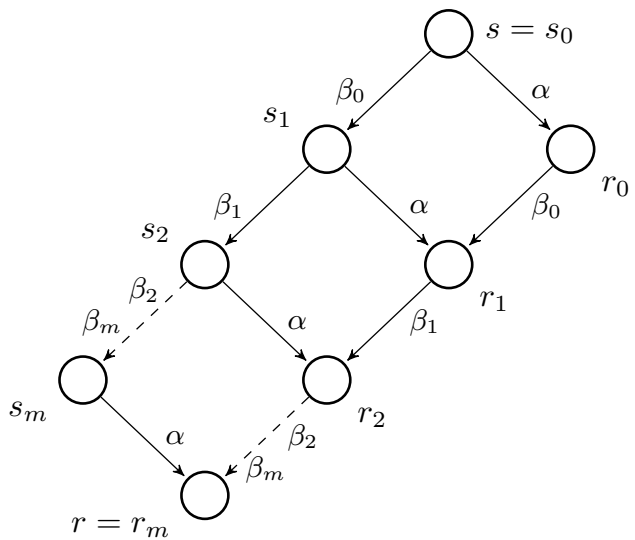
- Occorre mostrare che se scegliamo sempre la prossima transizione da esplorare in *ample*(s) non trascuriamo alcun cammino essenziale per la verifica della correttezza del grafo degli stati (completo).

- Dalla condizione **C1**, segue che i cammini del grafo completo degli stati non presenti nel grafo degli stati ridotto possono avere una delle due seguenti forme:

  1. il cammino ha un prefisso della forma $\beta_0\beta_1 \ldots \beta_m\alpha$, dove $\alpha \in$ *ample*(s) e ogni β_i è indipendente da tutte le transizioni in *ample*(s) inclusa α (si noti come da ciò segua immediatamente che $\beta_i \notin$ *ample*(s));
  2. il cammino è una sequenza infinita di transizioni $\beta_0\beta_1 \ldots$, dove ogni β_i è indipendente da tutte le transizioni in *ample*(s).

La correttezza della riduzione DFS - caso 1

- Dalla condizione **C1** segue anche che se in una sequenza finita di transizioni $\beta_0\beta_1 \ldots \beta_m$, eseguita a partire da s, non risulta presente alcuna transizione in $ample(s)$, allora tutte le transizioni in $ample(s)$ rimangono abilitate.
- Assumiamo che la sequenza di transizioni $\beta_0\beta_1 \ldots \beta_m\alpha$, che non verrà presa in considerazione dall'algoritmo DFS, raggiunga uno stato r.
- Tramite la ripetuta (m volte) applicazione delle condizioni di enabledness e commutatività, possiamo costruire una sequenza finita $\alpha\beta_0\beta_1 \ldots \beta_m$ che raggiunge r.



La correttezza della riduzione DFS - caso 1 - e la condizione **C2**

- Consideriamo le due sequenze di stati $\sigma = s_0 s_1 \ldots s_m r_m (= r)$ e $\rho = s r_0 r_1 \ldots r_m (= r)$ in figura, generate da $\beta_0 \beta_1 \ldots \beta_m \alpha$ e $\alpha \beta_0 \beta_1 \ldots \beta_m$, rispettivamente.
- Per poter ignorare σ, occorre che σ e ρ siano equivalenti rispetto allo stuttering. Ciò viene garantito se α è invisibile (condizione sufficiente), dato che in tal caso $L(s_i) = L(r_i)$, per ogni $0 \leq i \leq m$.
- Tale richiesta porta all'introduzione della condizione **C2**.

Condizione C2 [Invisibilità]

Se s non è fully expanded, allora ogni $\alpha \in \mathit{ample}(s)$ è invisibile.

La correttezza della riduzione DFS - caso 2

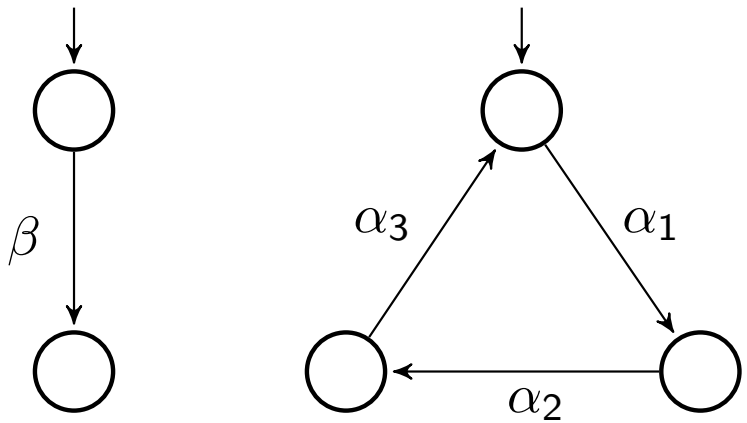
- Nel secondo caso, il cammino infinito $\beta_0\beta_1\beta_2\ldots$ che parte da s non include alcuna transizione contenuta in $ample(s)$.
- Per la condizione **C2**, tutte le transizioni in $ample(s)$ sono invisibili.
- Sia α una transizione contenuta in $ample(s)$. Il cammino generato dalla sequenza infinita di transizioni $\alpha\beta_0\beta_1\beta_2\ldots$ è equivalente rispetto allo stuttering a quello generato dalla sequenza infinita di transizioni $\beta_0\beta_1\beta_2\ldots$.
- Conclusione: il cammino $\beta_0\beta_1\beta_2\ldots$ non viene preservato dal grafo degli stati ridotto, ma c'è un cammino ad esso equivalente rispetto allo stuttering (il cammino $\alpha\beta_0\beta_1\beta_2\ldots$) che è, invece, presente nel grafo degli stati ridotto.

La correttezza della riduzione DFS - ultimo problema

Le condizioni (C1) e (C2) non sono ancora sufficienti a garantire che il grafo degli stati ridotto sia equivalente rispetto allo stuttering al grafo degli stati completo.

Problema: è possibile che qualche transizione venga ritardata indefinitamente a causa di un ciclo presente nel grafo degli stati ridotto costruito.

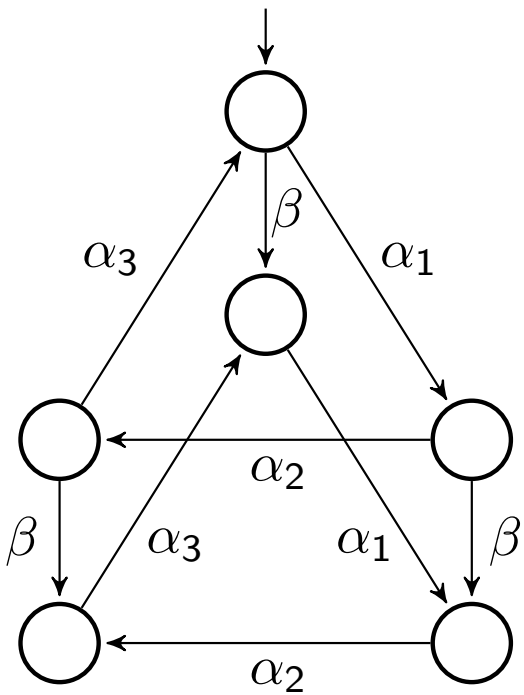
Un **esempio**:

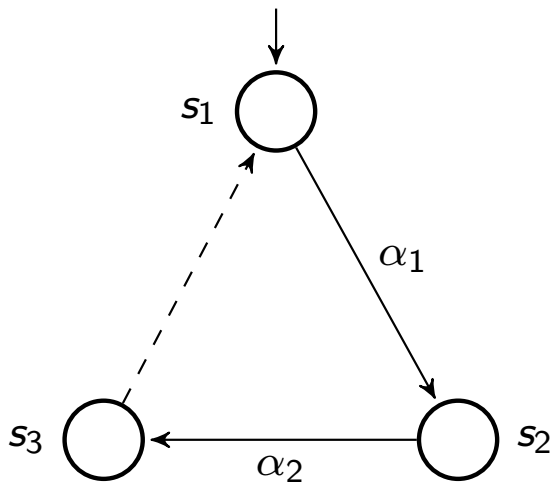


Assumiamo che (i) la transizione β sia indipendente dalle transizioni α_1, α_2 e α_3; (ii) le transizioni α_1, α_2 e α_3 siano interdipendenti; (iii) vi sia una lettera proposizionale p in AP' il cui valore di verità sia cambiato da vero a falso dall'esecuzione β (β sia visibile); (iv) le transizioni α_1, α_2 e α_3 siano invisibili.

Grafo completo e grafo ridotto

A sinistra viene mostrato il grafo completo; a destra le prime fasi della costruzione del grafo ridotto, dove α_1, α_2 e α_3 sono invisibili. Sia s_1 lo stato iniziale. La scelta $\mathit{ample}(s_1) = \{\alpha_1\}$, che porta a generare $s_2 = \alpha_1(s_1)$, rispetta le condizioni **C0**, **C1** e **C2**. Analogamente, la scelta $\mathit{ample}(s_2) = \{\alpha_2\}$, che porta a generare $s_3 = \alpha_2(s_2)$, rispetta le condizioni **C0**, **C1** e **C2**. Infine, le condizioni **C0**, **C1** e **C2** consentono di selezionare $\mathit{ample}(s_3) = \{\alpha_3\}$.





La condizione **C3**

Problema: il grafo degli stati ridotto generato non contiene alcuna sequenza nella quale il valore di verità di p venga cambiato da vero a falso.

Ciò accade perché ogni stato compreso nel ciclo (s_1, s_2 e s_2) differisce l'esecuzione di β ad un eventuale stato successivo. Quando il ciclo viene chiuso, la costruzione termina e la transizione β viene di fatto ignorata.

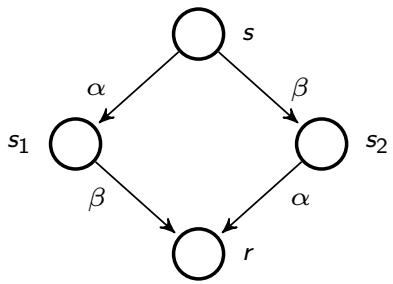
Per escludere grafi degli stati ridotti come quello di destra introduciamo la seguente condizione:

Condizione C3

Non è consentito avere un ciclo (nel grafo degli stati ridotto) che contenga uno stato in cui qualche transizione α risulti abilitata e, per ogni stato s del ciclo, α non venga inclusa in *ample(s)*.

Soluzione al Problema 1

Consideriamo di nuovo lo scenario descritto dalla seguente figura e assumiamo che l'algoritmo di riduzione DFS ponga β (e non α) in $ample(s)$ e non includa lo stato s_1 nel grafo degli stati ridotto.



Problema 1: la proprietà che vogliamo verificare potrebbe essere sensibile alla scelta (implicita) tra s_1 e s_2, non solo a r ed s

Per la condizione **C2**, β deve essere invisibile, quindi s, s_2, r e s, s_1, r sono equivalenti rispetto allo stuttering. Le formule invarianti rispetto allo stuttering non saranno in grado di distinguere tra le due sequenze.

Soluzione al Problema 2

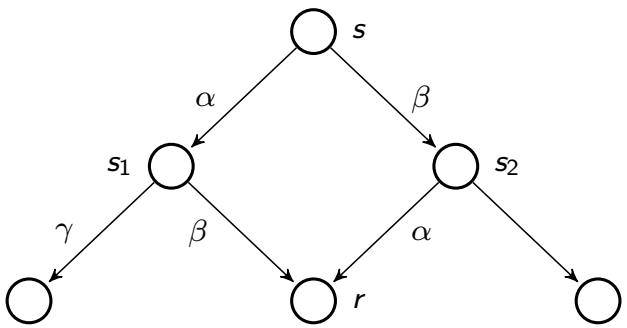
Problema 2: gli stati s_1 e s_2 potrebbero, accanto ad r, avere altri successori e tali successori risulterebbero non più raggiungibili qualora uno tra s_1 e s_2 venisse eliminato.

Assumiamo che esista una transizione γ abilitata a partire da s_1. Mostriamo che γ è ancora abilitata nello stato r e che le sequenze di transizioni $\alpha\gamma$ e $\beta\alpha\gamma$ portano a sequenze di stati equivalenti rispetto allo stuttering.

γ è indipendente da β, altrimenti la sequenza $\alpha\gamma$ violerebbe la condizione **C1**.

Dato che γ è abilitata in s_1, γ deve essere abilitata anche in r (β è invisibile per la condizione **C2**, come già osservato). Assumiamo che γ risulti in r' quando eseguita da r, e in s_1' quando eseguita da s_1.

Dato che β è invisibile, le due sequenze di stati s, s_1, s_1' e s, s_2, r, r' sono equivalenti rispetto allo stuttering.



Esempio: un programma per la mutua esclusione

Consideriamo il seguente programma per la mutua esclusione:

$$P = m : \ \textbf{cobegin}\ P_0 \ ||\ P_1\ \textbf{coend}\ m' :$$

dove i due processi P_0 e P_1 sono definiti nel seguente modo:

$P_0 :: l_0 :$ **while** $True$ **do**
$NC_0 :$ **wait**$(turn = 0)$;
$CR_0 :$ $turn := 1$;
end while;
l_0'

$P_1 :: l_1 :$ **while** $True$ **do**
$NC_1 :$ **wait**$(turn = 1)$;
$CR_1 :$ $turn := 0$;
end while;
l_1'

Esempio: assunzioni

- Il program counter pc del programma P può assumere i valori m, m' (entry point e exit point di P, rispettivamente) e $\perp$ (quando P_0 e P_1 sono attivi).
- Il program counter pc_i, con $i = 0, 1$, del processo P_i può assumere i valori l_i, l_i', NC_i, CR_i e $\perp$.
- $V = V_0 = V_1 = \{turn\}$ e $PC = \{pc, pc_0, pc_1\}$.
- Quando il valore di pc_i è CR_i, il processo P_i è nella sua regione critica (i due processi non possono trovarsi contemporaneamente nella loro regione critica). Quando il valore è NC_i, il processo è nella sua regione non critica. Affinché P_i possa accedere, in modo esclusivo, alla sua regione critica, deve valere $turn = i$ (è possibile una situazione di blocco individuale di P_i qualora l'altro processo rimanga indefinitamente nella propria regione critica).
- Gli stati iniziali di P sono descritti dalla formula:

$$\mathcal{S}_0(V, PC) \equiv pc = m \wedge pc_0 = \perp \wedge pc_1 = \perp$$

  All'inizio della computazione, la variabile $turn$ può assumere indifferentemente il valore 0 o il valore 1.
- Gli stati sono etichettati con $AP = \{NC_i, CR_i, l_i, turn = i, \perp\}$, con $i \in \{0, 1\}$, dove $CR_i \in L(s)$, se $pc_i = CR_i$ nello stato s, e $CR_i \notin L(s)$, se $pc_i \neq CR_i$ in s.

Esempio: la proprietà da verificare

- La proprietà di mutua esclusione è catturata dalla formula LTL$_{-X}$:

$$f = \mathbf{G}\neg(CR_0 \wedge CR_1)$$

- Vedremo ora come utilizzare l'algoritmo DFS per costruire il grafo degli stati ridotto, stuttering-equivalente al grafo degli stati completo rispetto ad un opportuno sottoinsieme AP' di proposizioni atomiche.

- Per verificare se P soddisfa f, scegliamo l'insieme di proposizioni atomiche $AP' = \{CR_0, CR_1\}$ (variabili preposizionali che compaiono nella formula f).

Esempio: le transizioni di P

Le transizioni del programma P che sono abilitate in qualche stato raggiungibile di P sono le seguenti:

- $\alpha \ : \ pc = m \ \wedge \ pc'_0 = l_0 \ \wedge \ pc'_1 = l_1 \ \wedge \ pc' = \perp$
- $\beta_i \ : \ pc_i = l_i \ \wedge \ pc'_i = NC_i \ \wedge \ True \ \wedge \ same(turn) \ \wedge \ same(pc_j)$
- $\gamma_i \ : \ pc_i = NC_i \ \wedge \ pc'_i = CR_i \ \wedge \ turn = i \ \wedge \ same(turn) \ \wedge \ same(pc_j)$
- $\delta_i \ : \ pc_i = CR_i \ \wedge \ pc'_i = l_i \ \wedge \ turn' := (i+1) \ mod \ 2 \ \wedge \ same(pc_j)$
- $\epsilon_i \ : \ pc_i = NC_i \ \wedge \ pc'_i = NC_i \ \wedge \ turn \neq i \ \wedge \ same(turn) \ \wedge \ same(pc_j)$

dove $i \in \{0, 1\}$ e $j = (i+1) \ mod \ 2$.

La condizione *True* presente nella definizione di β_i è quella del ciclo *while*.

Le transizioni visibili rispetto ad AP' sono quelle in cui CR_0 o CR_1 hanno valori differenti prima e dopo la transizione. Quindi, $\{\gamma_0, \gamma_1, \delta_0, \delta_1\}$ sono visibili.

Esempio: dipendenza delle transizioni di P

- Ogni transizione è dipendente da se stessa perché la relazione di dipendenza è riflessiva.
- Tutte le transizioni sono dipendenti da α dato che α deve essere eseguita prima di qualsiasi altra transizione del programma.
- La **relazione di dipendenza** per le altre transizioni è calcolata con le seguenti due regole:
  - due transizioni che cambiano la stessa variabile (inclusi i program counter) sono dipendenti;
  - se una transizione assegna un valore a una variabile e un'altra transizione controlla il valore di tale variabile, allora le transizioni sono dipendenti.
- Tutte le transizioni che appartengono allo stesso processo sono fra loro dipendenti.
- $(\gamma_1, \delta_0), (\gamma_0, \delta_1), (\epsilon_1, \delta_0), (\epsilon_0, \delta_1), (\delta_0, \delta_1)$ sono in D dato che δ_i cambia la variabile *turn*, mentre γ_i ed ϵ_i ne controllano il valore.
- Vanno, infine, aggiunte le coppie simmetriche.

Esempio: grafo degli stati

Gli stati e gli archi inclusi nel grafo ridotto sono in grassetto.

L'algoritmo DFS visita gli stati del grafo ridotto nel seguente ordine:
$s_0, s_1, s_3, s_4, s_6, s_{10}, s_{11}, s_{13}, s_7, s_8$.

Grafi completo e ridotto (in grassetto):

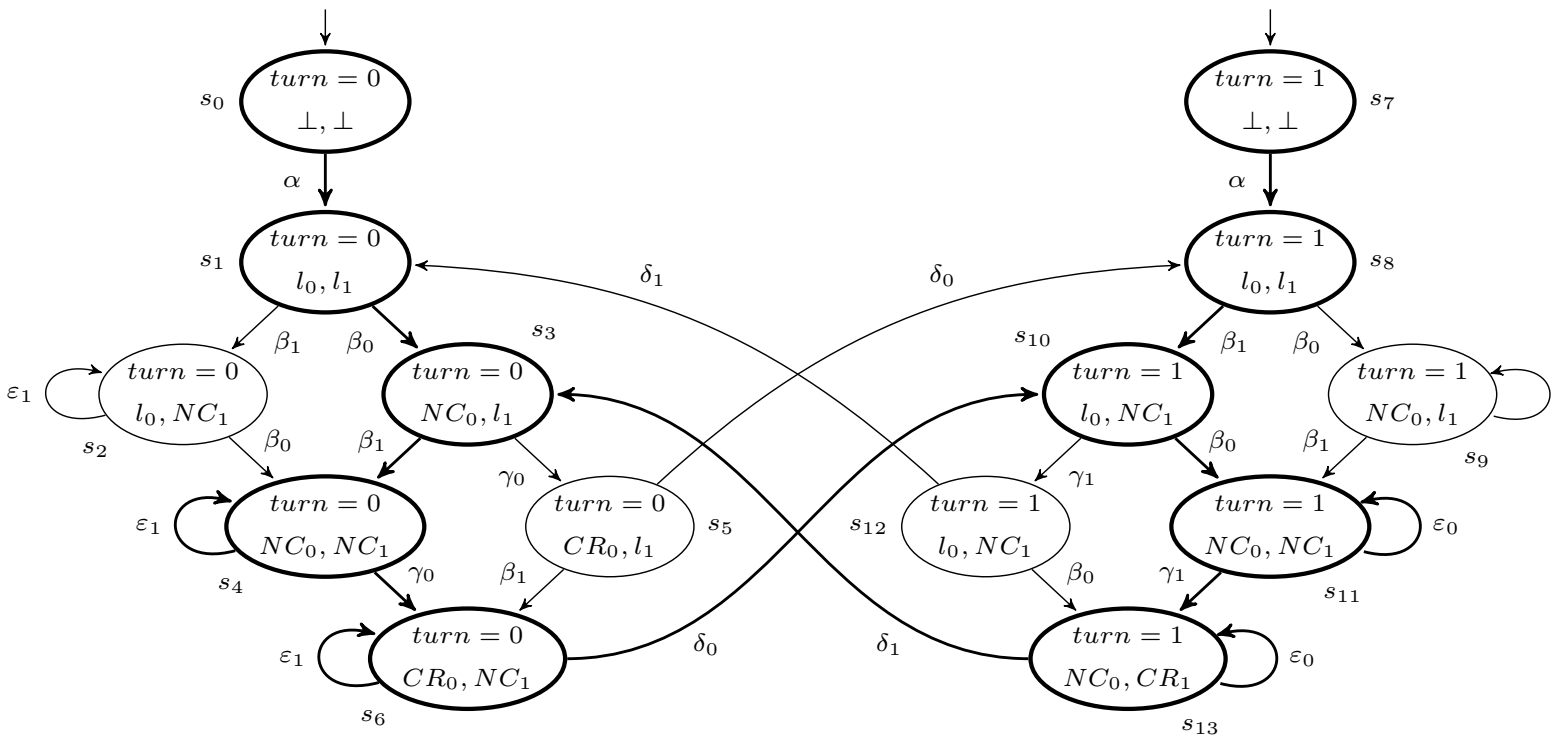


Figura 1: Grafo degli stati ridotto per un programma di mutua esclusione

Esempio: l'esecuzione dell'algoritmo DFS - 1

1. L'esecuzione di DFS inizia da s_0, uno dei due stati iniziali. Per tale stato, si ha $\mathit{ample}(s_0) = \mathit{enabled}(s_0) = \{\alpha\}$ (condizione **C0**).
2. Le possibili scelte per $\mathit{ample}(s_1)$ sono: $\mathit{ample}(s_1) = \{\beta_0\}$, $\mathit{ample}(s_1) = \{\beta_1\}$ e $\mathit{ample}(s_1) = \{\beta_0, \beta_1\}$. Tralasciamo l'ultima, che comporta una minore riduzione. Le prime due opzioni corrispondono alla selezione delle transizioni abilitate in P_0 e in P_1, rispettivamente.

   Le condizioni **C0** e **C1** sono soddisfatte. Supponiamo che $\mathit{ample}(s_1) = \{\beta_0\}$. Lungo tutti i cammini da s_1, β_0 viene immediatamente eseguita oppure β_1 viene eseguita prima di β_0, ma β_1 è indipendente da β_0.

   Anche la condizione **C2** è soddisfatta, dato che β_0 e β_1 sono invisibili. Infine, la condizione **C3** è soddisfatta perché non viene formato (ancora) alcun ciclo. La scelta tra i due insiemi è arbitraria, selezioniamo $\mathit{ample}(s_1) = \{\beta_0\}$.
3. L'esecuzione di β_0 porta da s_1 a s_3. Usando un argomento in parte simile al precedente (inclusa l'invisibilità di $\{\beta_1\}$), possiamo selezionare $\mathit{ample}(s_3) = \{\beta_1\}$.
4. $\mathit{ample}(s_4) = \{\gamma_0, \epsilon_1\}$. Per s_4 non possiamo selezionare l'insieme $\{\gamma_0\}$, perché γ_0 è visibile. Non possiamo nemmeno selezionare il singoletto $\{\epsilon_1\}$, perché avremmo un (self) loop in cui la transizione γ_0 risulta abilitata ma non inclusa in un ample set, violando così la condizione **C3**.

Esempio: l'algoritmo DFS

5. $ample(s_6) = \{\epsilon_1, \delta_0\}$ (dato che sono dipendenti, vanno scelte entrambe, per non violare **C1**).
6. $ample(s_{10}) = \{\beta_0\}$ e $ample(s_{11}) = \{\gamma_1, \epsilon_0\}$ (in analogia con s_3 e s_4).
7. La transizione δ_1 presa da s_{13} chiude il ciclo s_3 s_4 s_6 s_{10} s_{11} s_{13}, che soddisfa la condizione **C3** ($\gamma_0 \in enabled(s_3)$ e $\gamma_0 \notin ample(s_3)$, ma $\gamma_0 \in ample(s_4)$; analogamente per γ_1 rispetto a s_{10} e s_{11}).
8. L'algoritmo DFS continua la ricerca dall'altro stato iniziale s_7. Selezioniamo $ample(s_7) = \{\alpha\}$.
9. $ample(s_8) = \{\beta_1\}$ (in analogia con quanto fatto per s_1).
10. Eseguendo β_1 da s_8, raggiungiamo solo lo stato s_{10} che è già stato visitato. Quindi, l'algoritmo termina.

Ora, applicando un algoritmo di model checking per LTL, si può verificare se il grafo degli stati ridotto soddisfa la formula f. Si ha, infatti, che il grafo degli stati completo soddisfa la formula se e solo se il grafo degli stati ridotto la soddisfa.

Come verificare le condizioni **C0** - **C3**?

- Come verificare se un dato insieme di transizioni, abilitate in un certo stato, soddisfa le condizioni **C0** - **C3**?
- Il rispetto della condizione **C0** per un dato stato può essere verificata in tempo costante.
- La condizione **C2** si può verificare esaminando le transizioni presenti nell'insieme considerato.
- La condizione **C1** si riferisce a stati futuri (alcuni dei quali nemmeno presenti nel grafo degli stati ridotto) e non è quindi immediatamente verificabile.

Teorema.

Il problema di verificare il rispetto della condizione **C1** *per uno stato s ed un insieme di transizioni $T \subseteq enabled(s)$ è difficile almeno quanto il problema della raggiungibilità nello spazio degli stati completo.*

Come procedere? Eviteremo di verificare la condizione **C1** per un sottoinsieme arbitrario di transizioni abilitate. Forniremo, invece, una procedura per calcolare un insieme di transizioni che garantisca, per costruzione, la soddisfacibilità di **C1**, sebbene essa possa produrre degli ample set che non consentono la più grande riduzione possibile.

Come verificare le condizioni **C0 - C3**?

Anche la condizione **C3** è definita in termini globali, anche se fa riferimento al grafo degli stati ridotto. Rimpiazziamo **C3** con una condizione più forte che può essere verificata direttamente sullo stato corrente.

Lemma.

Una condizione sufficiente per **C3** *è che in ogni ciclo sia presente almeno uno stato fully expanded.*

Dimostrazione per assurdo.
Modi efficienti per forzare la condizione **C3** si possono ottenere sfruttando la specifica strategia di ricerca/visita usata per generare lo spazio degli stati ridotto (nel nostro caso, DSF).

Condizione C3′

Se s non è fully expanded, allora nessuna transizione in $ample(s)$ può raggiungere uno stato che si trova sullo stack di ricerca.

Strategia: cerchiamo di selezionare un ample set che non includa un back edge; in caso non sia possibile, lo stato corrente viene fully expanded.

Euristiche per gli ample set: notazione

Sulla base delle osservazione precedenti, è possibile fornire un **algoritmo** per il calcolo degli ample set.

Per presentare l'algoritmo, utilizziamo la seguente notazione:

- per ogni i, indichiamo il valore del program counter del processo P_i in uno stato s con $pc_i(s)$;
- $pre(\alpha)$ contiene tutte le transizioni la cui esecuzione abilita la transizione α, ossia $pre(\alpha)$ contiene tutte le transizioni β tali che esista uno stato s per cui $\alpha \notin enabled(s), \beta \in enabled(s)$ e $\alpha \in enabled(\beta(s))$;
- $dep(\alpha)$ è l'insieme delle transizioni che dipendono da α, vale a dire $\{\beta \mid (\beta, \alpha) \in D\}$;
- T_i è l'insieme delle transizioni del processo P_i e $T_i(s) = T_i \cap enabled(s)$ è l'insieme delle transizioni di P_i che sono abilitate nello stato s;
- $current_i(s)$ è l'insieme delle transizioni di P_i che sono abilitate in qualche stato s' tale che $pc_i(s') = pc_i(s)$. Tale insieme contiene sempre $T_i(s)$; inoltre, esso può contenere transizioni che sono abilitate in stati in cui il program counter ha il valore $pc_i(s)$, ma non sono abilitate in s.

Euristiche per gli ample set: osservazioni

Alcune osservazioni:

- in ogni cammino che inizia da s, qualche transizione in $current_i(s)$ deve essere eseguita prima che altre transizioni in T_i possano essere eseguite;
- le definizioni di $pre(\alpha)$ e della relazione D (a partire dalla quale viene definito $dep(\alpha)$) possono non essere esatte: $pre(\alpha)$ potrebbe contenere transizioni che non abilitano α e D potrebbe includere coppie di transizioni che in realtà sono indipendenti; questa flessibilità consente di calcolare gli ample set in modo efficiente, preservando la correttezza della riduzione;
- le definizioni date si possono facilmente generalizzare ad insiemi di transizioni. Ad esempio, è possibile far riferimento a $dep(T_i(s))$.

In generale, a modelli di computazione diversi corrispondono modalità diverse per determinare gli elementi di $pre(\alpha)$ e di D.

Come costruire $pre(\alpha)$?

La definizione di $pre(\alpha)$ dipende dal modello di computazione adottato:

- $pre(\alpha)$ conterrà le transizioni dei processi che contengono α e possono modificare il program counter assegnando ad esso un valore che rende α eseguibile;
- se la condizione abilitante per α coinvolge variabili condivise, allora $pre(\alpha)$ includerà anche tutte le altre transizioni che possono modificare il valore di tali variabili;
- se α è/comprende operazioni di scambio messaggi con code, ossia se α spedisce o riceve dati su una qualche coda q, allora $pre(\alpha)$ includerà le transizioni di altri processi che rispettivamente ricevono o spediscono dati attraverso q.

Come costruire D?

La definizione di D dipende dal modello di computazione adottato:

- coppie di transizioni che condividono una variabile, che viene modificata da almeno una di esse, sono dipendenti;
- coppie di transizioni che appartengono allo stesso processo sono dipendenti. In particolare, tali sono coppie di transizioni contenute in $current_i(s)$ per qualche stato s e processo P_i;
- coppie di transizioni di tipo *send*, che utilizzano la stessa coda di messaggi, sono dipendenti, perché l'invio di un messaggio da parte di una di esse può causare il riempimento della coda, disabilitando l'esecuzione dell'altra transizione. Inoltre, il contenuto della coda dipende, in generale, dall'ordine di esecuzione delle due transizioni. Per ragioni analoghe, coppie coppie di transizioni di tipo *receive*, che utilizzano la stessa coda di messaggi, sono dipendenti.

Euristiche per gli ample set: la funzione *check_C1*

- Un ovvio candidato per $ample(s)$ è $T_i(s)$. Dato che le transizioni in esso contenute sono interdipendenti, un ample set per s o le include tutte o non ne include nessuna.
- Per costruire un ample set per lo stato corrente s, partiamo da un processo P_i tale che $T_i(s) \neq \emptyset$.
- Per accertarci che $ample(s) = T_i(s)$ soddisfi la condizione **C1** osserviamo quanto segue.

  Come vedremo, vi sono due casi in cui tale scelta per $ample(s)$ potrebbe violare **C1**.

  In entrambi i casi, vengono eseguite alcune transizioni indipendenti da quelle in $T_i(s)$, ottenendo, quale effetto finale, l'abilitazione di una transizione α che dipende da $T_i(s)$.

  Le transizioni indipendenti presenti nella sequenza non possono ovviamente appartenere a T_i perché tutte le transizioni in P_i sono interdipendenti.

Euristiche per gli ample set: la funzione *check_C1*

- Vediamo nel dettaglio i due casi in cui **C1** viene violata.

1. α appartiene a qualche altro processo P_j. Affinché ciò avvenga è necessario che $dep(T_i(s))$ includa una transizione del processo P_j. Questa condizione può essere verificata esaminando la relazione di dipendenza.

2. α appartiene a P_i. Supponiamo che la transizione $\alpha \in T_i$ che viola **C1** sia eseguita in uno stato s'. Le transizioni eseguite da s a s' sono indipendenti da $T_i(s)$ e, quindi, appartengono ad altri processi. Perciò, $pc_i(s') = pc_i(s)$ e $\alpha \in current_i(s)$. Inoltre, $\alpha \notin T_i(s)$ (altrimenti non violerebbe **C1**). Quindi, $\alpha \in current_i(s) \setminus T_i(s)$.

   Dato che α è disabilitata in s ($\alpha \notin T_i(s)$), la sequenza di transizioni che porta da s a s' deve includere una transizione in $pre(\alpha)$. Una condizione necessaria è che $pre(current_i(s) \setminus T_i(s))$ includa transizioni di processi diversi da P_i. Anche questa condizione può essere verificata in modo effettivo.

Euristiche per gli ample set: la funzione *check_C1*

- In entrambi i casi, $T_i(s)$ viene abbandonato e sostituito con un insieme candidato $T_j(s)$ di un qualche altro processo P_j.
  Va osservato come si tratti di un approccio prudenziale, in quanto non è detto che la condizione **C1** alla fine risulti effettivamente non rispettata.

```
1  function check_C1(s, P_i)
2      for all P_j ≠ P_i do
3          if dep(T_i(s)) ∩ T_j ≠ ∅ or pre(current_i(s)\T_i(s)) ∩ T_j ≠ ∅ then
4              return False;
5          end if;
6      end for all;
7      return True;
8  end function
```

Euristiche per gli ample set: la funzione *check_C2*

La funzione *check_C2* prende in input un insieme di transizioni e restituisce *True* se tutte le transizioni nell'insieme sono invisibili. Altrimenti restituisce *False*.

```
1  function check_C2(X)
2      for all α ∈ X do
3          if visible(α) then return False;
4      return True;
5  end function
```

Euristiche per gli ample set: la funzione *check_C3'*

La funzione *check_C3'* verifica se lo stato ottenuto tramite l'esecuzione di una transizione in un dato insieme $X \subseteq enabled(s)$ è ancora nello stack di ricerca (tale funzione sfrutta il marking degli stati come *on_stack* o *completed*).

```
1  function check_C3'(s,X)
2      for all α ∈ X do
3          if on_stack(α(s)) then return False;
4      return True;
5  end function
```

Euristiche per gli ample set: l'algoritmo per *ample*(s)

L'algoritmo per *ample*(s) cerca un processo P_i tale che $T_i(s)$ soddisfi tutte le condizioni da **C0** a **C3**. Se un tale processo non esiste, *ample* restituisce l'insieme *enabled*(s).

```
1  function ample(s)
2      for all P_i such that T_i(s) ≠ ∅ do
3          if check_C1(s, P_i) and check_C2(T_i(s)) and check_C3'(s, T_i(s)) then
4              return T_i(s);
5          end if;
6      end for all;
7      return enabled(s);
8  end function
```

Introduzione

- SPIN (*S*imple *P*roMeLa *In*terpreter) è un tool di verifica per modelli di sistemi software distribuiti, sviluppato a partire dagli anni '80 presso i Bell Labs (Unix group of the Computing Sciences Research Center).
- Più precisamente, SPIN è un model checker per LTL$_{-X}$ on-the-fly: combina la costruzione del grafo ridotto degli stati con la verifica del soddisfacimento della specifica.
- SPIN prende in input un modello del sistema ed un requisito e verifica se il sistema soddisfa o meno il requisito.
- Se il requisito non viene rispettato, SPIN produce un run che lo viola (controesempio).
- La verifica di SPIN si focalizza sulla dimostrazione della correttezza delle interazioni fra processi.
- La comunicazione fra processi avviene tramite rendevous (sincrono), scambio di messaggi asincroni attraverso canali con buffer, accesso a variabili condivise o una combinazione di tali modalità.

Simulazione vs Verifica

SPIN può essere utilizzato in due modi: come simulatore oppure come verificatore.

- Nella modalità di **simulazione**, SPIN consente di riprodurre i comportamenti codificati dal modello di sistema. Le interfacce grafiche di SPIN forniscono una controparte visuale della simulazione che risulta estremamente utile nel processo di debugging dei modelli.
- Nella modalità di **verifica**, SPIN consente di verificare proprietà di interesse del sistema.
- Simulazione e verifica possono essere accoppiate: nel caso in cui il verificatore trovi un controesempio, il simulatore può essere utilizzato per mostrare la traccia d'errore mediante una simulazione guidata.

PROMELA

- SPIN si avvale del linguaggio di specifica PROMELA (*PRO*cess *ME*ta-*LA*nguage).
- È un linguaggio C-like per descrivere modelli di sistemi distribuiti. Mutua alcuni costrutti dal Dijkstra's guarded language e dal linguaggio CSP (Communicating Sequential Processes) di Hoare per le interazioni tra processi.
- Un modello specificato in PROMELA è non-deterministico e a stati finiti. Esso consiste di:
  - dichiarazioni di variabili e loro tipi
  - dichiarazioni di canali
  - dichiarazioni di tipi
  - dichiarazioni di processi
  - processo init (opzionale)
- Per implementazioni di SPIN e, più in generale, documentazione su SPIN, si rimanda al sito `http://www.spinroot.com`.