

# OTTIMIZZAZIONI IN SPIN

---

Introduzione su SPIN

---

Esplosione degli stati

---

Partial order reduction

---

Verifica approssimata tramite  
hashing



LUCIANO BERCINI (0522501684)

PROF. SALVATORE LA TORRE

# INTRODUZIONE SPIN

Creato da Gerard J. Holzmann negli anni 80, SPIN è un model checker esplicito basato sul linguaggio di modellazione **PROMELA** (Process **Meta** Language).

SPIN viene utilizzato in particolare per verificare in maniera automatica proprietà come l'assenza di deadlock e race condition in sistemi concorrenti in cui l'affidabilità è fondamentale (e.g. sistemi aerospaziali).



# SPIN BASICS

Si modella il sistema in PROMELA.

Si definiscono le proprietà da verificare tramite asserzioni sullo stato del modello, never claim, o LTL.

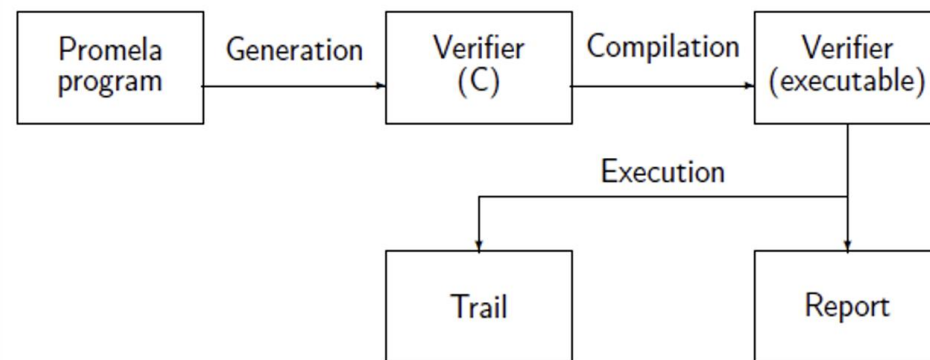
SPIN esplora lo spazio degli stati del modello.

Se sono trovate delle violazioni della proprietà, SPIN restituisce un controesempio (trail).

```
assert(x > 0);
```

```
never {  
  do  
    :: (x <= 0) -> break  
    :: else -> skip  
  od  
}
```

```
[](x > 0)
```



# IL PROBLEMA DELL'ESPLOSIONE DEGLI STATI

Il principale limite nella verifica con SPIN è l'esplosione degli stati.

Lo **spazio degli stati** è l'insieme di stati possibili, dato dal **prodotto cartesiano** dell'insieme dei valori che possono assumere le variabili e le control location. Un'espressione condizionale partiziona lo spazio degli stati in due.

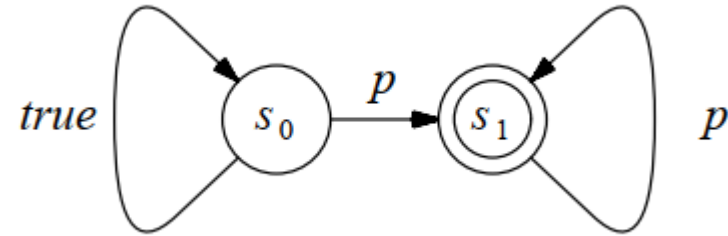
```
proctype P() {  
    int i = 0;  
    do  
        :: i < 3 -> i++  
        :: else -> break  
    od  
}  
  
init {  
    run P();  
    run P();  
    run P();  
    run P();  
}
```

L'esplosione accade perché bisogna:


1. Esplorare tutte le combinazioni di valori possibili;
2. Esplorare tutti gli ordini di esecuzione dei processi;
3. Memorizzare gli stati esplorati per evitare esplorazioni duplicate.

# MODEL CHECKING: LTL E AUTOMI DI BÜCHI

1. Si costruiscono gli automi di Büchi  $S$  e  $A$ , dati rispettivamente dal sistema e della negazione della proprietà LTL ( $\neg f$ ).
2. Si calcola l'automa  $T$  dato dal prodotto  $S \times A$ .
3. Qualsiasi esecuzione infinita accettata da  $T$  corrisponde ad un'esecuzione di  $S$  in cui  $\neg f$  è soddisfatta, e per questo la proprietà originale  $f$  è violata.



*Non-Deterministic Büchi Automaton for LTL formula  $\Diamond \Box p$ .*



## EFFICIENTE IN TEORIA - COSTOSO IN PRATICA

La complessità computazionale può sembrare bassa, di  $O(|S \times A|)$ , ma in realtà:

- La dimensione di  $A$  può essere esponenziale rispetto alla dimensione della formula LTL (numero di operatori temporali). *Questo è un problema minore perché le formule LTL tendono ad avere piccole dimensioni.*
- La dimensione di  $S$  è spesso data dal prodotto di più automi che modellano diversi processi concorrenti. Se abbiamo  $n$  automi con  $k$  stati ciascuno, il sistema può avere fino a  $k^n$  stati.



## VERIFICA ON- THE-FLY

Non appena troviamo una violazione, possiamo fermare la verifica, evitando inutili computazioni. Il prodotto è costruito on-the-fly tramite una nested DFS, consistente di due DFS, una interna e una esterna:

- La DFS esterna: Trova gli stati accettanti raggiungibili dagli stati iniziali.
- La DFS interna: Controlla se uno degli stati accettanti può raggiungere sè stesso (i.e. trova un ciclo).



# ASTRAZIONE

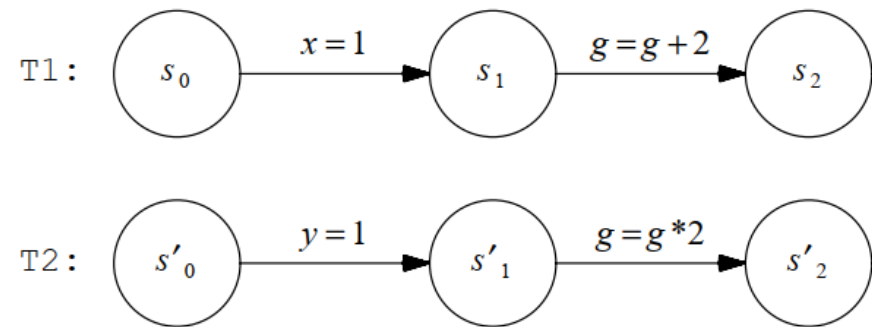
**La rimozione di dettagli può spesso risultare in meno stati da esplorare.**

Ad esempio, un modello di un file server non ha bisogno di entrare nel dettaglio degli errori come "scrittura parziale", "settori danneggiati", etc... ma possiamo usare un'astrazione come "operazione fallita" o "operazione ritentata" senza entrare nel dettaglio. Facendo ciò, semplifichiamo il modello senza aggiungere falsi positivi.



# PROBLEMA DELL'ESPLORAZIONE COMPLETA

**Problema:** Nei sistemi concorrenti, l'esplorazione di tutti i possibili ordini delle azioni porta a un'esplosione combinatoria. Molti ordini, però, portano allo stesso stato finale. Questo perché alcune operazioni sono indipendenti.



**Fig. 10** — Automata  $T1$  and  $T2$ .

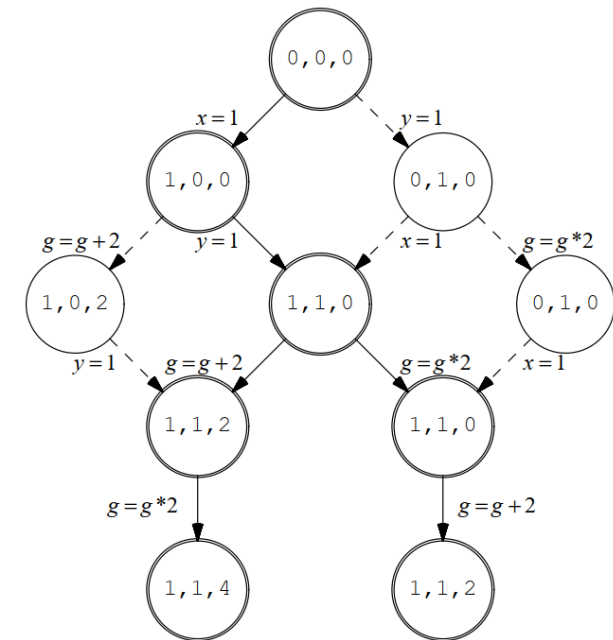
Da notare come, mettendo  $x=1$  e  $y=1$  in ordini diversi, il risultato non cambia essendo operazioni indipendenti.

# PARTIAL ORDER REDUCTION - SOLUZIONE

La Partial Order Reduction evita di esplorare ordini di esecuzione che **non influenzano il risultato**, riducendo così il numero di stati da analizzare.

1:  $x = 1; g = g+2; y = 1; g = g*2;$   
2:  $x = 1; y = 1; g = g+2; g = g*2;$   
3:  $x = 1; y = 1; g = g*2; g = g+2;$   
4:  $y = 1; g = g*2; x = 1; g = g+2;$   
5:  $y = 1; x = 1; g = g*2; g = g+2;$   
6:  $y = 1; x = 1; g = g+2; g = g*2;$

2:  $x = 1; y = 1; g = g+2; g = g*2;$   
3:  $x = 1; y = 1; g = g*2; g = g+2;$



**Fig. 11** — Full and Reduced Depth-First Search for  $T1 \times T2$ .

Gli stati in grassetto e le transizioni con frecce non tratteggiate rappresentano la DFS ridotta.

# PARTIAL ORDER REDUCTION - LTL

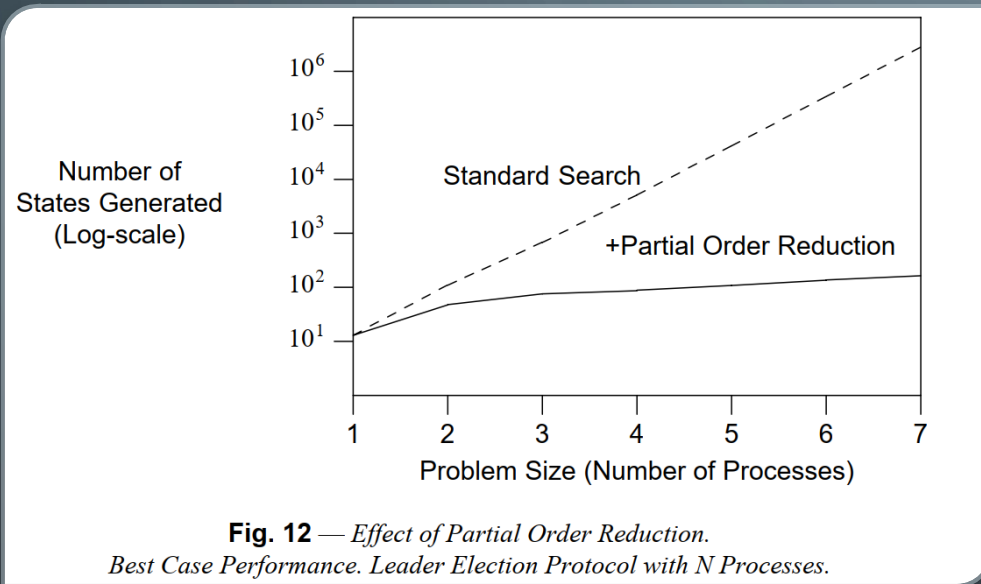
Vi possono essere formule LTL che sono valide nel grafo ridotto da POR, ma che non sono valide nel grafo originale. Le formule LTL quindi possono introdurre anch'esse delle dipendenze. Questo perché la dipendenza delle operazioni non dipende solo dalla struttura dell'automa e dall'accesso ai dati, ma anche dalla LTL che ci interessa.

```
1:  x = 1; g = g+2; y = 1; g = g*2;  
2:  x = 1; y = 1; g = g+2; g = g*2;  
3:  x = 1; y = 1; g = g*2; g = g+2;  
4:  y = 1; g = g*2; x = 1; g = g+2;  
5:  y = 1; x = 1; g = g*2; g = g+2;  
6:  y = 1; x = 1; g = g+2; g = g*2;
```

```
2:  x = 1; y = 1; g = g+2; g = g*2;  
3:  x = 1; y = 1; g = g*2; g = g+2;
```

$\Box (x \geq y).$

# PARTIAL ORDER REDUCTION



Tramite POR, abbiamo una riduzione esponenziale sul numero di stati generati. Questa riduzione ovviamente dipende dal numero di transizioni indipendenti. Anche in casi peggiori, il POR riesce a costruire il grafo avendo virtualmente zero overhead.

# POR IN DETTAGLIO - LTS

Consideriamo qualsiasi problema di verifica che possa essere formalizzato come un problema di analisi della raggiungibilità in un sistema di transizione etichettato (LTS) finito.

Un LTS è definita come una tripla  $\{S, s_0, T\}$ , dove:

- $S$  è un insieme finito di stati;
- $s_0$  è lo stato iniziale in  $S$ ;
- $T$  è un insieme finito di transizioni, con  $T \subseteq (S \times S)$ .

La LTS viene utilizzata per formalizzare il comportamento di un singolo processo sequenziale, o un sistema concorrente composto da più processi sequenziali che eseguono in modo asincrono.

L'LTS può essere visualizzato come un grafo dove:

- I nodi rappresentano gli stati.
- Gli archi diretti rappresentano le transizioni.
- Un cammino nel grafo corrisponde a una possibile esecuzione del sistema concorrente.

# POR - SEMANTICA DELLE TRANSIZIONI LTS

Per ogni transizione  $t \in T$ :

- $\text{Label}(t)$  rappresenta l'istruzione associata a  $t$ .
- $\text{Pid}(t)$  è il processo sequenziale a cui appartiene  $\text{Label}(t)$ .

La semantica di un'istruzione  $a = \text{Label}(t)$  è definita dalle funzioni:

- $\text{Conds}(a)$ : insiemi di stati in cui  $a$  è abilitata (eseguibile).
- $\text{Act}(a, s)$ : stato risultante dopo l'esecuzione di  $a$  in uno stato  $s \in \text{Conds}(a)$ .

Uno stato è "abilitato" o "eseguibile" solo se riferito nel program counter corrente di quel processo sequenziale.

Due istruzioni  $a$  e  $b$  sono *indipendenti* in uno stato  $s \in S$  (scritto come  $\{a, b\} \in \text{Ind}(s)$ ) sse soddisfano le seguenti condizioni:

1.  $s \in \text{Conds}(a)$  e  $s \in \text{Conds}(b)$ , ovvero entrambe sono abilitate in  $s$ .
2.  $\text{Act}(a, s) \in \text{Conds}(b)$  e  $\text{Act}(b, s) \in \text{Conds}(a)$ , ovvero l'esecuzione di una non disabilita l'altra.
3.  $\text{Act}(b, \text{Act}(a, s)) = \text{Act}(a, \text{Act}(b, s))$ , ovvero l'effetto dell'esecuzione di  $a$  seguito da  $b$  lo stesso di  $b$  seguito da  $a$ .

Due istruzioni  $a$  e  $b$  sono globalmente indipendenti sse sono indipendenti in ogni possibile stato  $s$  in cui sono abilitate:

$$s \in (\text{Conds}(a) \cap \text{Conds}(b)) \rightarrow \{a, b\} \in \text{Ind}(s)$$



# ALGORITMO DI RIDUZIONE - OBIETTIVO

L'obiettivo è **ridurre il numero di stati da esplorare durante la verifica di proprietà nei sistemi concorrenti**, mantenendo valide le conclusioni del *model checking*. L'algoritmo utilizza:

- **Ricerca in profondità (DFS)** per esplorare gli stati del sistema.
- **Costruzione del prodotto sincrono** tra il sistema (LTS) e un *automa di Büchi* (che rappresenta una formula LTL).
- **Rilevamento di cicli di accettazione** per verificare se una proprietà è soddisfatta o meno.
- **Passi di riduzione.**



# GENERAZIONE LTS CON DFS

Consideriamo prima la **DFS standard** che genera la **LTS** a partire dal sistema.

Inizializzazione:

- Si crea e si ottimizza la struttura delle transizioni del sistema tramite analisi statica.
- Si inizializza:
  - **State Space**: insieme degli stati esplorati.
  - **Stack**: pila usata dal DFS.
- Si avvia la DFS dallo stato iniziale  $s_0$ .

```
1 start_search( $s_0$ )
2 { derive and optimize transition structures
3   enter  $s_0$  into Statespace;
4   push  $s_0$  onto Stack;
5   Dfs(1); /* see Figure 1c */
6 }
```



# GENERAZIONE LTS CON DFS - ESPANSIONE DEGLI STATI CON DFS

Dopo l'inizializzazione, la ricerca in profondità **visita gli stati** e segue le transizioni per costruire l'LTS.

1. Estrae lo stato in cima alla pila.
2. Andando a cercare tutte le transizioni abilitate in ogni processo, andremo a popolare lo statespace.

```
7 dfs(N)
8 { s = top(Stack);
9   for each sequential process i
10  {   nxt = all transitions in  $F$  enabled in  $s$  with  $Pid(t)=i$ 
11      for all  $t$  in  $nxt$ 
12      {    $s'$  = successor of  $s$  after  $t$ ;
13          if  $\{s', N\}$  NOT in Statespace
14          {   enter  $\{s', N\}$  into Statespace;
15              push  $s'$  onto Stack;
16              Dfs (N);
17          }   }   }
18  pop  $s$  from Stack
19 }
```

# COSTRUZIONE DEL PRODOTTO SINCRONO CON L'AUTOMA DI BÜCHI

```
20 Dfs(N)
21 { s = top(Stack);
22   nxt = all transitions in G enabled in s; /* the Büchi Automaton */
23   for all t in nxt
24   {   s' = successor of s after t;
25       if {s',N} NOT in Statespace
26       {   enter {s',N} into Statespace;
27           push s' onto Stack;
28           dfs(N);
29       }
30   }
31   pop s from Stack
32 }
```

L'LTS viene combinata con un automa di Büchi  $G$  (che rappresenta la formula LTL da verificare). L'accoppiamento sincrono tra il sistema  $F$  e l'automata di Büchi  $G$  viene realizzato alternando le chiamate a **Dfs(N)** (alla riga 16) e **dfs(N)** (alla riga 28). Ogni coppia di chiamate successive esplora una transizione sincrona di  $F \times G$ .

Le transizioni dell'automata vengono abilitate solo se il loro predicato è vero nello stato corrente. Quindi, il codice trova tutte le transizioni di  $G$  che possono essere seguite in base allo stato  $s$  dell'LTS del sistema concorrente.

# RILEVAMENTO DEI CICLI DI ACCETTAZIONE

Una proprietà LTL è soddisfatta se esiste un ciclo di accettazione nell'automa di Büchi. Per verificare ciò:

- Quando si visita uno stato di accettazione, si avvia una **seconda DFS** per vedere se è possibile tornare a quello stato.
- Se sì, si è trovato un **ciclo di accettazione**, e quindi la proprietà è verificata.

Da notare che  $N=1$  allora stiamo esplorando, e quando  $N=2$  stiamo rilevando i cicli di accettazione.

Nel "report acceptance cycle", le informazioni sono contenute nello stack, e possono essere usate per generare il controesempio (il trail).

```
20 Dfs(N)
21 { s = top(Stack);
22   nxt = all transitions in G enabled in s; /* the Büchi Automaton */
23   for all t in nxt
24   {   s' = successor of s after t;
24a     if N == 2 and s' == seed
24b     {   report acceptance cycle
24c         return
24d     }
25     if {s',N} NOT in Statespace
26     {   enter {s',N} into Statespace;
27         push s' onto Stack;
28         dfs(N);
28a         if N == 1 and s is an accepting state in G
28b         {   seed = s
28c             dfs(2)
28d         }
29     }   }
30   pop s from Stack
31 }
```



## RIDUZIONE STATICA

La riduzione statica elimina le transizioni "inutili", rendendo l'esplorazione più veloce.

Il metodo ha l'obiettivo di trovare il più piccolo insieme di transizioni sufficiente per l'espansione, sapendo che vogliamo mantenere le proprietà di safety e liveness.

Viene detta statica perché lo step di ordinamento dell'algoritmo che vedremo virtualmente non incide a runtime.

```

7 dfs(N)
8 { s = top(Stack);
8a order processes; /* using safety as ordering principle - see text */
9  for each sequential process i
9a  {   boolean NotInStack = true
9b      boolean AtLeastOneSuccessor = false
10      nxt = all transitions  $t$  in  $F$  enabled in  $s$  with  $Pid(t)=i$ 
11      for all  $t$  in  $nxt$ 
12      {    $s'$  = successor of  $s$  after  $t$ ;
13          if  $\{s', N\}$  NOT in Statespace
14          {   enter  $\{s', N\}$  into Statespace;
15              push  $s'$  onto Stack;
16              Dfs(N);
16a          } else if  $s'$  in Stack /* reduction proviso */
16b              NotInStack = false
16c              AtLeastOneSuccessor = true
16d          }
16e      if  $AtLeastOneSuccessor \wedge NotInStack$ 
16f          break /* from the loop over processes */
17  }
18  pop  $s$  from Stack
19 }

```

*Figure 1e – Reduced Expansion Step*

# RIDUZIONE STATICA

L'algoritmo mostra la variante con espansione ridotta. I due punti chiavi sono:

1. Linea 8a: Se un processo è safe, ovvero non influenza lo stato globale in modo critico, lo esploriamo per prima, evitando poi di esplorare transizioni che potrebbero non essere necessarie.
2. Linea 16e: Se la condizione è soddisfatta non consideriamo le transizioni degli altri processi. Se questa condizione è soddisfatta significa che:
  - Abbiamo trovato almeno un successore nuovo.
  - Nessuna transizione ha riportato a uno stato già esplorato.

Quindi, non esploriamo tutto, ma selezioniamo solo una transizione sufficiente a dimostrare la proprietà che stiamo verificando.

# PROCESSI SAFE E NON NELL'ORDINAMENTO

Un processo è "**sicuro incondizionatamente**" quando le sue transizioni sono sempre indipendenti e non influenzano altri processi.

Un processo è "**sicuro condizionalmente**" se è sicuro solo in certe condizioni specifiche dello stato del sistema.

In SPIN, ci sono 5 istruzioni che possono essere marchate staticamente come incondizionalmente safe, e come condizionalmente safe quando appaiono in costrutti di selezione.

1. Accesso a variabili esclusivamente locali.
2. Una operazione di receive in una coda di messaggi  $q$ , se solo un processo può ricevere i messaggi, testare il contenuto di  $q$  o la sua dimensione (exclusive receive-access queue).
3. Equivalente per la scrittura in una coda di messaggi  $q$  (exclusive send-access queue).
4. Il test booleano  $nfull(q)$ , che ritorna true se la coda  $q$  non è piena, se questo test viene fatto da un processo che ha exclusive send-access in quella coda.
5. Il test booleano  $nempty(q)$ , se questo test viene fatto da un processo che ha exclusive read-access in quella coda.

Protocol	Algorithm	States	Transitions	Time(sec.)	Memory (Mb)
Best-Case	Non-Reduced	100,001	450,002	13.2	4.3
	Static Reduction	47	47	(<0.1)	1.0
	Dynamic Reduction	47	47	0.1	1.4
Worst-Case	Non-Reduced	100,001	450,002	14.5	5.0
	Static Reduction	100,001	450,002	16.7	5.1
	Dynamic Reduction	100,001	450,002	84.5	5.3
Tpc	Non-Reduced	3,918,286	11,762,426	630.6	268.4
	Static Reduction	391,534	466,753	30.6	26.2
	Dynamic Reduction	267,204	295,395	131.4	18.9
Snoopy	Non-Reduced	91,920	305,460	14.4	11.5
	Static Reduction	16,279	23,532	1.7	3.2
	Dynamic Reduction	7,158	8,459	6.8	2.6
Pftp	Non-Reduced	417,321	1,244,865	73.2	62.3
	Static Reduction	53,244	67,901	6.8	9.3
	Dynamic Reduction	125,718	163,459	105.5	20.6
Leader	Non-Reduced	45,885	185,032	8.1	9.6
	Static Reduction	79	79	0.1	1.1
	Dynamic Reduction	79	79	0.2	1.4

## MISURE PRATICHE

I best e worst case sono semplicemente due esempi "artificiali", in cui nel primo sono utilizzate solo variabili locali mentre nel secondo solo variabili globali. Tpc è un modello dello switch telefonico, Snoopy è un protocollo di coerenza cache, Pftp è una versione del file transfer protocol, e infine Leader è un protocollo di elezione di un leader con cinque processi. Le misure sono state fatte in una workstation Spac-10 con 128Mbyte di RAM.



A decorative graphic on the left side of the slide, consisting of white lines and circles on a dark background, resembling a circuit board or a network diagram.

## VERIFICA APPROSSIMATA

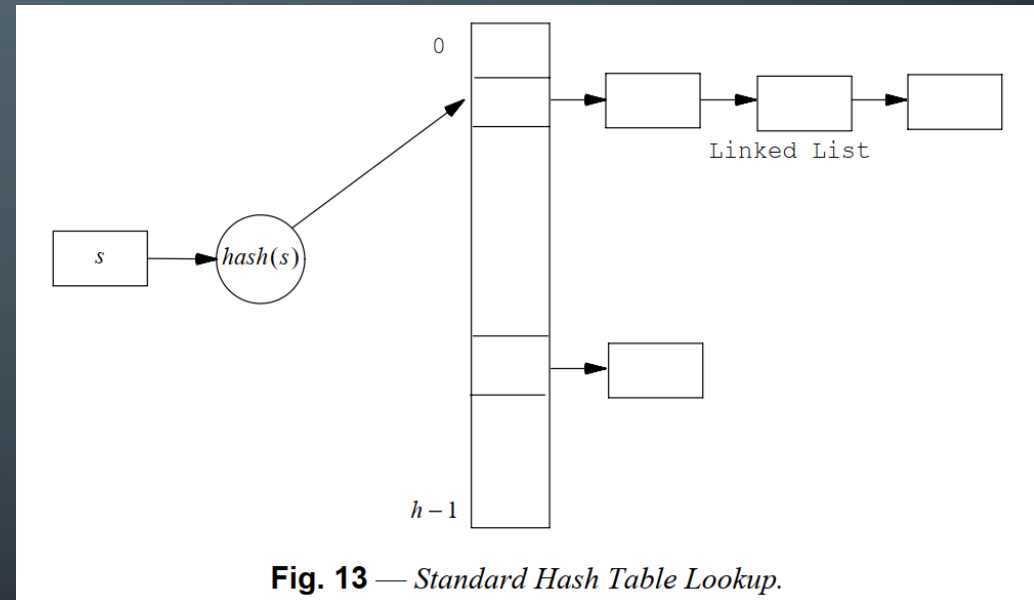
Il model checking si basa sull'esplorazione degli stati, ovvero sulla generazione di tutti i possibili stati di un sistema e sulla verifica se siano già visitati. Questo controllo della visita porta il problema da un problema esponenziale ad uno lineare al numero di stati, dove ogni stato raggiungibile viene visitato solo una volta.

**Quindi verificare velocemente se uno stato è già stato visitato è fondamentale.**



# VERIFICA APPROSSIMATA

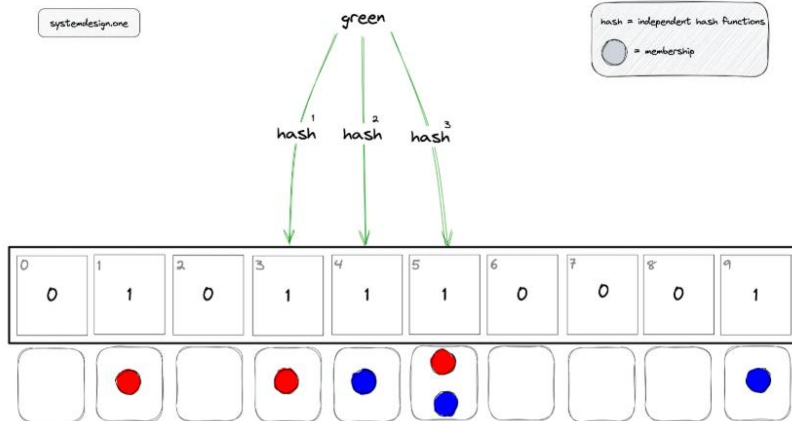
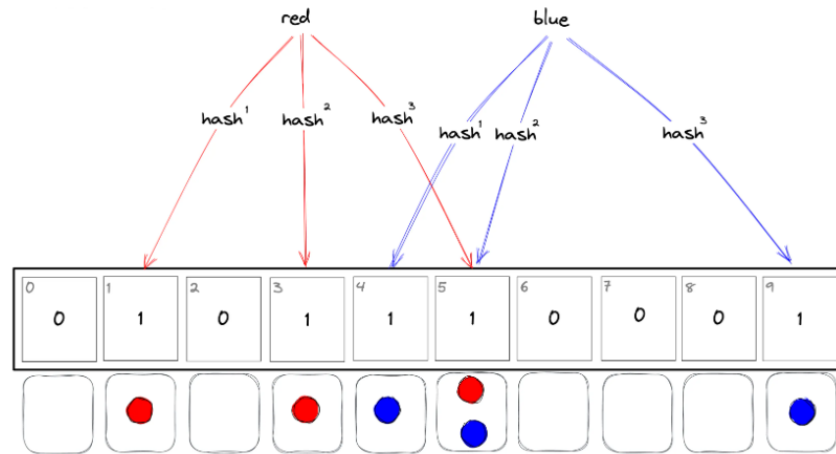
Per tenere traccia degli stati, i model checker utilizzano una tabella hash per controllare rapidamente se uno stato è già stato visitato. L'efficienza del model checking dipende dalla dimensione della tabella hash.



# VERIFICA APPROSSIMATA

Per risparmiare memoria, possiamo utilizzare **un singolo bit per stato** invece di tutti i suoi dati (e.g. valori delle variabili del sistema, stato del programma, ecc...). Questo approccio può portare alla mancata esplorazione di alcuni stati (riducendo la copertura), ma consente di verificare sistemi molto più grandi rispetto all'approccio tradizionale.

Quando si usa una **tabella hash con un solo bit per stato**, si perde quasi tutta l'informazione. La tabella non memorizza lo stato vero e proprio, ma solo il fatto che sia stato visitato o meno. Questo riduce il consumo di memoria, ma significa che alcuni stati potrebbero essere trattati erroneamente come già esplorati (a causa di collisioni di hash), portando a una copertura incompleta.



# BLOOM FILTER

Per testare se uno stato è già stato visitato, o più in generale se un elemento fa parte di un insieme di elementi, è possibile usare la struttura data "filtro bloom".

**L'idea è quella di rappresentare ogni stato con k bit, determinati da k funzioni di hash indipendenti.** Questo determina k posizioni distinte nella tabella per ogni stato, riducendo la probabilità di collisione.

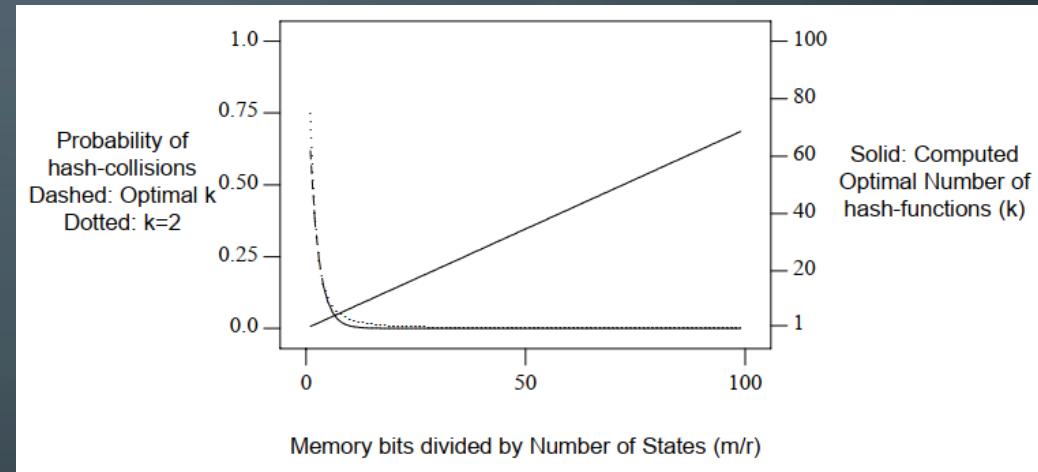
Ne segue che un valore alto di k porta a meno collisioni, ma aumenta notevolmente i costi.

Nel primo esempio vi sono due inserimenti con  $k=3$ .  
Nel secondo esempio, il verde porta ad un falso positivo.

# DOUBLE BITSTATE HASHING

I requisiti di runtime per la nested DFS sono linearmente dipendenti da  $k$ ; questo perché il calcolo delle funzioni hash è l'operazione più costosa che il model checker deve fare. Un'esecuzione con  $k=90$  costa approssimativamente 45 volte di più rispetto ad una run con  $k=2$ .

**SPIN utilizza  $k=2$ .**



$m$  = spazio tabella

$r$  = numero stati

$k$  = funzioni hash

Man mano che aumentiamo  $m/r$ ,  $k$  diventa meno influente e utilizzare  $k=2$  è molto vantaggioso.

# HASH-COMPACT

L'hash-compact è una variante del bitstate hashing, in cui **calcoliamo un hash da 64 bit dello stato e lo inseriamo all'interno di una normale tabella hash**, invece dello stato  $s$ .

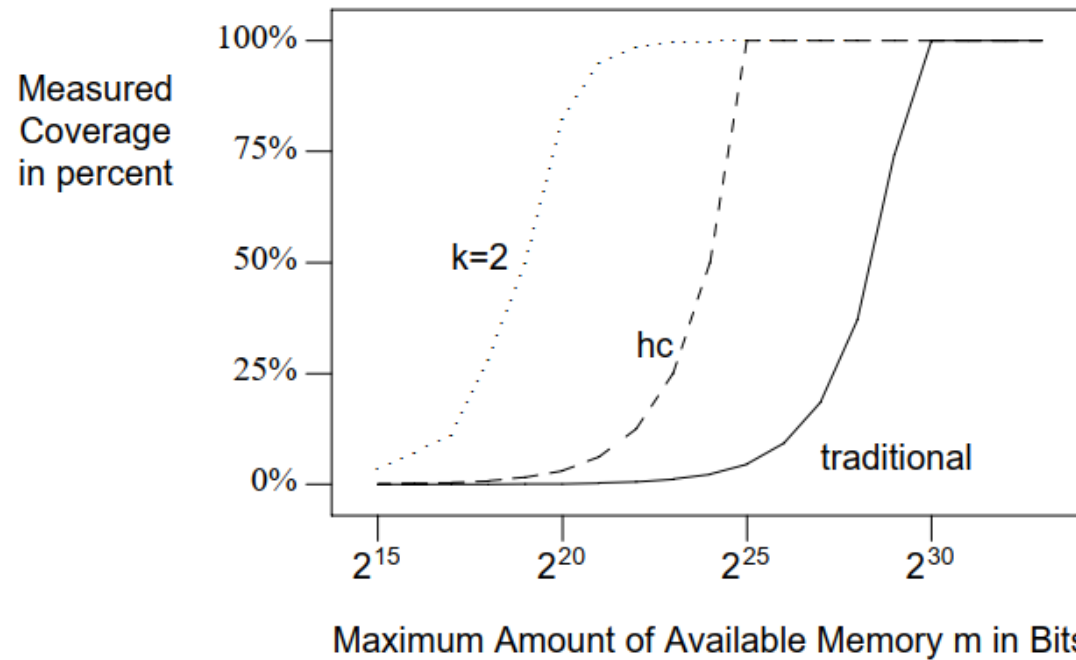
In questo caso  $m$  è da considerarsi  $2^{64}$ .

Se assumiamo  $r = 10^7$  allora lo spazio necessario è:

$$10^7 * 64bits = 640Mbits \approx 80MB$$

SPIN utilizza anche il metodo di hash-compact.

Caratteristica	Bitstate Hashing (k=2)	Hash-Compact
Memoria per stato	2 bit	64 bit
Probabilità di collisione (Upper Bound)	$4 \times 10^{-4}$ (circa 99% di copertura)	$10^{-37}$ (praticamente 100% di copertura)
Complessità di lookup	$O(1)$ (semplice accesso a un bit)	$O(1)$ (lookup in tabella hash)



**Fig. 15** — *Measured Coverage of Hash-Compact ( $hc$ ) and Double Bitstate Hashing ( $k=2$ ), for varying  $m$ , and fixed  $r=427567$  states and  $S=1376$  bits.*

## BITSTATE HASHING VS HASH-COMPACT VS FULLSTATE

In sintesi, se abbiamo poca memoria o molti stati, il bitstate hashing è da preferire, seguito da hash-compact e infine il metodo tradizionale di mantenere l'intero stato. Da notare come il double bitstate hashing raggiunge un'accuratezza del 50% con solo lo 0.1% delle risorse necessarie per una visita tradizionale.

**GRAZIE PER L'ATTENZIONE!**



# RIFERIMENTI

- A Primer on Model Checking:  
<https://spinroot.com/spin/Doc/p40-ben-ari.pdf>
- SPIN Model Checker Primer and Reference Manual:  
<https://www.cin.ufpe.br/~acm/esd/intranet/spinPrimer.pdf>
- Software Model Checking:  
<https://spinroot.com/gerard/pdf/marktoberdorf.pdf>
- An improvement in formal verification  
<https://spinroot.com/spin/Doc/forte94a.pdf>
- An analysis of Bitstate Hashing:  
<https://spinroot.com/spin/Doc/fmsd98.pdf>
- Bloom Filter: <https://systemdesign.one/bloom-filters-explained/#what-is-a-bloomfilter>