Verifica con modelli dotati di stack (stati infiniti)

Modellare il control-flow di programmi

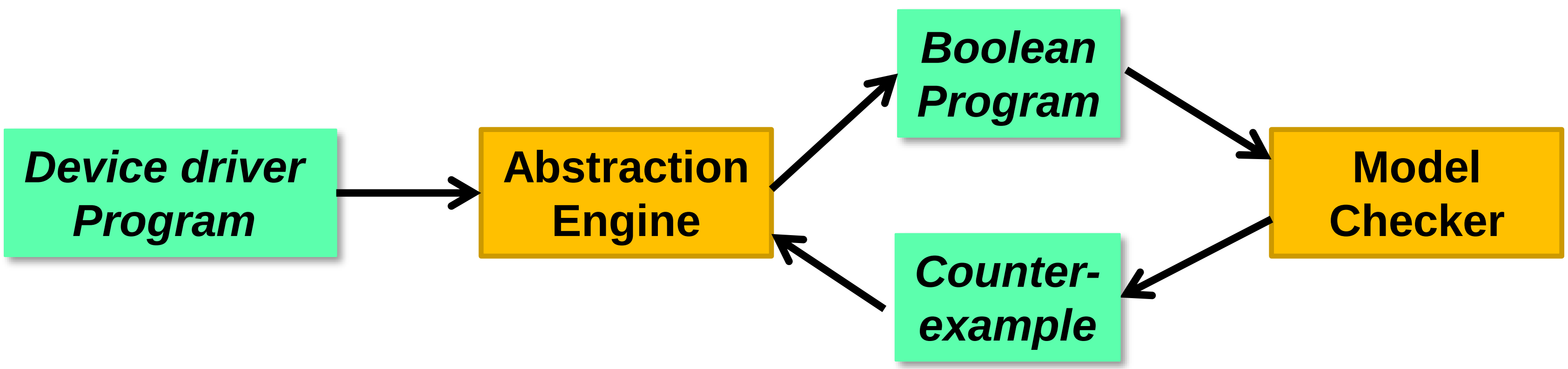
- le chiamate a procedura ricorsive sono normalmente utilizzate nella pratica della programmazione
- per modellare il flusso di controllo con gli automi finiti occorre limitare la profondità del call-stack
- in alternativa si possono usare modelli dotati di stack

Modelli di programmi (Dominio variabili finito)

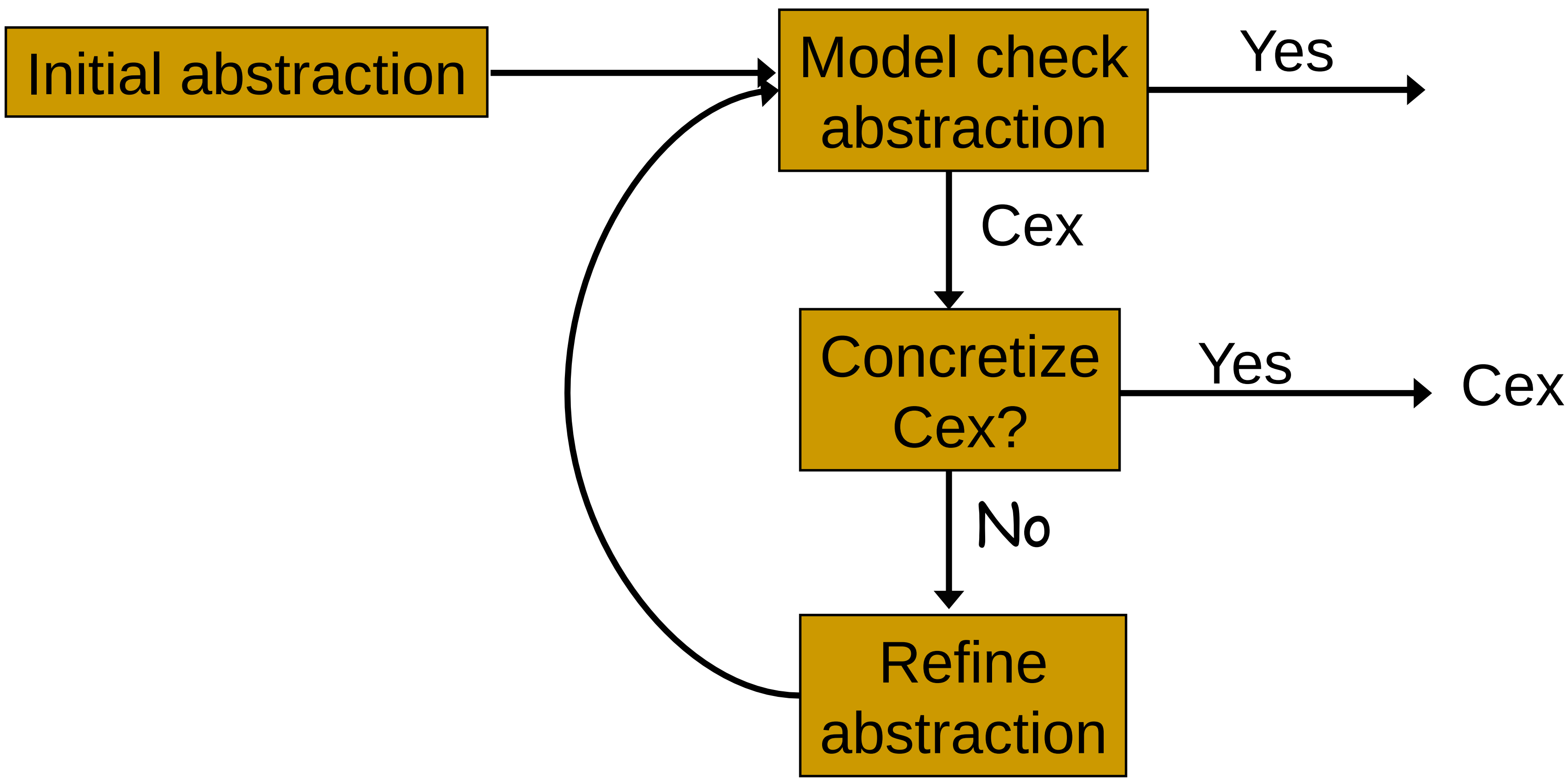
- Programmi booleani
  - Programmazione strutturata, unico tipo boolean
- Macchine ricorsive a stati finiti
  - automi a stati finiti che possono chiamare (ricorsivamente) altri automi a stati finiti
- Automi pushdown
  - Macchine a stati finiti con stack non limitato

Interesse per i programmi booleani

- catturano fedelmente programmi con variabili su domini finiti
- forniscono astrazioni di programmi:
  - Microsoft SLAM/SDV li usa per verificare device drivers
  - sono il risultato di una *predicate abstraction*
    - mantengono predicati sui domini dei dati



CounterExample-Guided Abstraction Refinement (CEGAR loop)



Esempio: partizione di una lista

```
typedef struct cell {
    int val;
    struct cell* next;
} *list;


list partition(list *l, int v) {
    list curr, prev, newl,
        nextCurr;
    curr = *l;
    prev = NULL;
    newl = NULL;
```

```
    while (curr != NULL) {
      nextCurr = curr->next;
      if (curr->val > v) {
        if (prev != NULL)
          prev->next = nextCurr;
        if (curr == *l)
          *l = nextCurr;
        curr->next = newl;
L:      newl = curr;
      }
      else  prev = curr;
      curr = nextCurr;
    }
    return newl;
}
```

Astrazione di partition (1)

- Predicati usati per funzione di astrazione:
  curr == NULL, prev == NULL, curr->val > v, prev->val > v

```
bool unknown() begin if (*) then return true; else return false;  fi end

void partition()
begin
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};

    {curr==NULL} = unknown();          // curr = *l;
    {curr->val>v} = unknown();
    {prev==NULL} = true;               // prev = NULL;
    {prev->val>v} = unknown();
    skip;                              // newl = NULL;
```

Astrazione di partition (2)

```
  while(*)  do                               // while(curr!=NULL) {
        assume(!{curr==NULL});               //
        skip;                                //      nextCurr = curr->next
        if (*) then                          //      if (curr->val > v) {
          assume({curr->val>v});             //
          if (*) then                        //        if (prev != NULL)
            assume(!{prev==NULL});           //
            skip;                            //          prev->next = nextCurr;
          fi                                 //
          if (*)  then                       //        if (curr == *l)
            skip;                            //            *l = nextCurr;
          fi                                 //
          skip;                              //        curr->next = newl;
L:        skip;                              //        newl = curr
                                              //     }
```

Astrazione di partition (3)

```
        else                                 //     else {
          assume(!{curr->val>v});            //
          {prev==NULL} = {curr==NULL};       //         prev = curr;
          {prev->val>v} = {curr->val>v};     //
        fi                                   //     }
        {curr==NULL} = unknown();            // curr = nextCurr;
        {curr->val>v} = unknown();
    od
    assume({curr==NULL});
end
```

Programmi Booleani: Sintassi

<*pgm*>::= <*gvar-decl*> <*proc-list*>

<*gvar-decl*>::= **decl** x**;** | <*gvar-decl*> <*gvar-decl*>

<*proc-list*> ::= <*proc*> <*proc-list*> | <*proc*>

<*proc*> ::= $f^{h,k}$ (x_1, . . . , x_h) **begin** <*lvar-decl*> <*stmt*> **end**

<*lvar-decl*> ::= **decl** x **;** | <*lvar-decl*> <*lvar-decl*>

<*stmt*> ::= <*stmt*> **; <***stmt*> **|** **skip** **|** <*assign*> **|**

call $f^{h,0}$(<*expr*$_1$>,. . ., <*expr*$_h$>) **|**

return <*expr*$_1$>,. . ., <*expr*$_k$> **|**

if (<*expr*>) **then** **<***stmt*> **else** **<***stmt*> **fi** **|**

while (<*expr*>) **do** **<***stmt*> **od** **|**

assume*(*<*expr*> *)* **|** **assert***(*<*expr*> *)* **|** **enforce***(*<*expr*> *)*

<*assign*> ::= x_1, . . . , x_m **=** <*expr*$_1$>,. . ., <*expr*$_m$>**|**

x_1, . . . , x_m **=** <*expr*$_1$>,. . ., <*expr*$_m$> **constrain** (<*expr*>) **|**

x_1, . . . , x_k **=** $f^{h,k}$(<*expr*$_1$>,. . ., <*expr*$_h$>)

<*expr*> ::= *T* | *F* | $*$ | x | ¬ <*expr*> | <*expr*>∨ <*expr*> |<*expr*>∧ <*expr*>

Esempio programma booleano

```
bool locked;
     /* global variable */

void lock() begin
    assert(!locked);
    locked := 1;
    . . . /* acquire a lock */
end

void unlock() begin
    assert(locked);
    . . . /* release the lock */
    locked := 0;
end
```

```
bool g (bool x) begin
    return !x;
end

void main() begin
    bool a,b;
    locked,a := 0,0;
    . . .
    lock();
    b := g(a);
    unlock();
    . . .
end
```

Esempio

Predicati: {b1 ≡ x < y,
b2 ≡ y < z,
b3 ≡ x < z}

```
void main() {

    int x = *;
    int y = *;
    int z = *;

    if (x<y) {
          if (y<z) {
                    if (!(x<z)) {
                              error();
                    }
          }
    }
}
```

```
void main()  begin
    bool b1, b2, b3;
    b1,b3 = *,*;
    b1,b2 = *,*;
    b2,b3 = *,*;

    if (b1) then
          if (b2) then
              if (!b3) then
                    error();
              fi
          fi
    fi
end
```

Esempio

Predicati: {b1 ≡ x < y,
b2 ≡ y < z,
b3 ≡ x < z}

```
void main() {

    int x = *;
    int y = *;
    int z = *;

    if (x<y) {
        if (y<z) {
            if (!(x<z)) {
                error();
            }
        }
    }
}
```

```
void main()  begin
    bool b1, b2, b3;
    b1,b3 = *,*;
    b1,b2 = *,*;
    b2,b3 = *,* constrain (!(b1 &&
                    b2’ && !b3’));
    if (b1) then
        if (b2) then
            if (!b3) then
                error();
            fi
        fi
    fi
end
```

Model checker per Boolean programs

- Bebop: MC di Static Driver Verifier (SDV) – Microsoft
  - Permette di usare una serie di euristiche
  - Usato prevalentemente nello sviluppo di device drivers
- Moped: MC basato su automi pushdown
- Getafix: MC simbolico con algoritmi basati su fixed-point calculus
  - progetto congiunto di Università di Salerno e University of Illinois (UC)

Recursive State Machine (RSM)

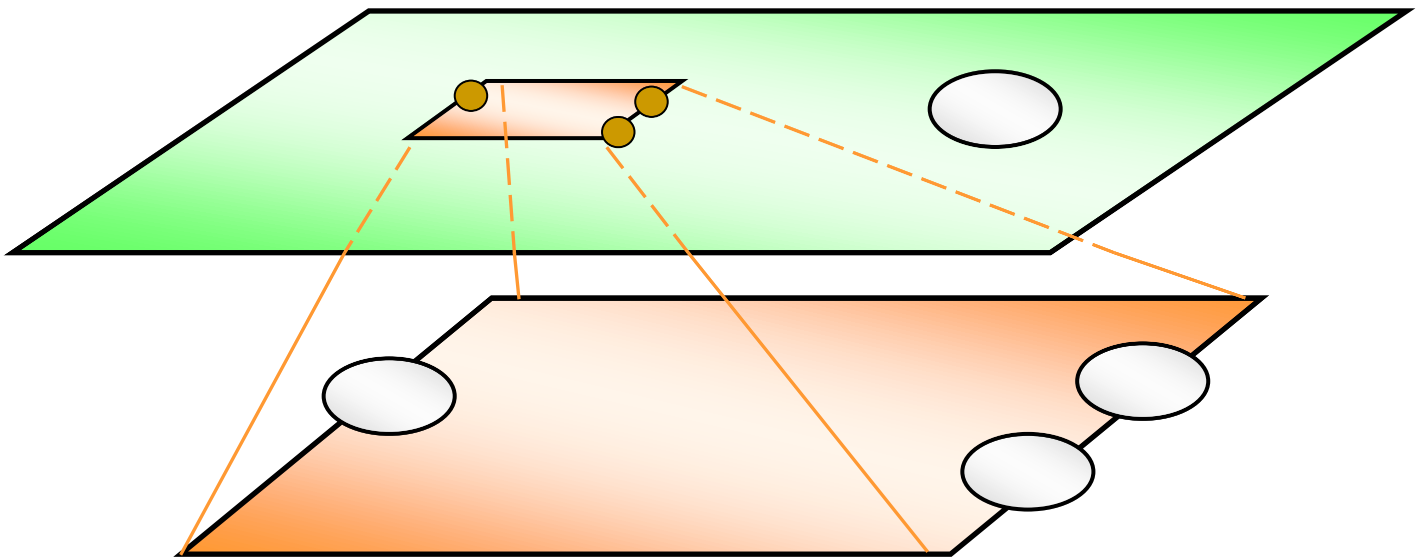
- Modello a stati corrispondente a programma con variabili su domini finiti
  - valore variabili memorizzato negli stati

- Una RSM $\mathcal{M} = (M_1,\ldots, M_k)$ si compone di
  - k macchine che modellano k procedure
  - ogni macchina può invocare ogni altra macchina in maniera ricorsiva
  - ogni macchina corrisponde a un grafo finito

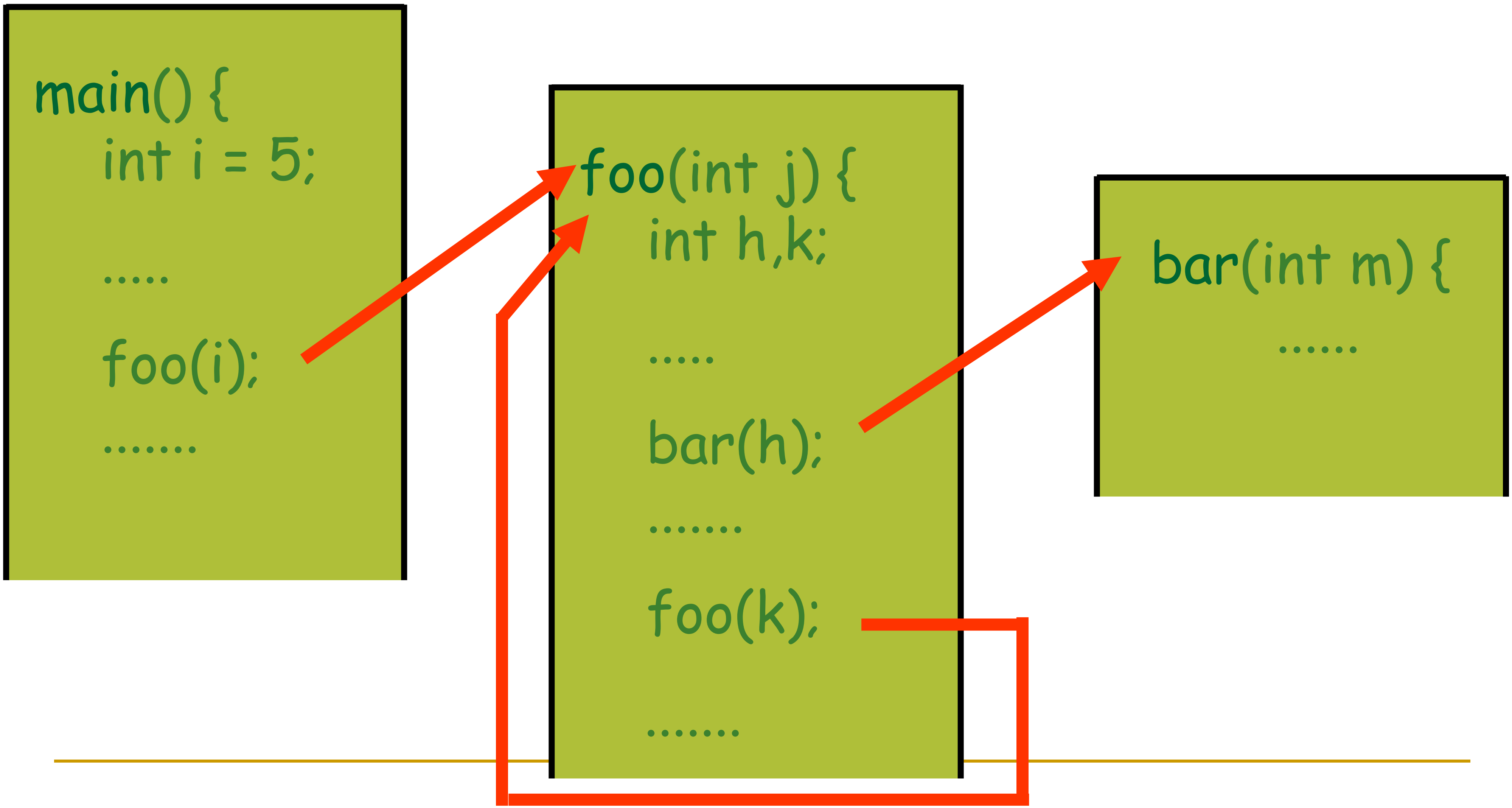
Vertici

Ogni macchina ha due tipi di vertici:

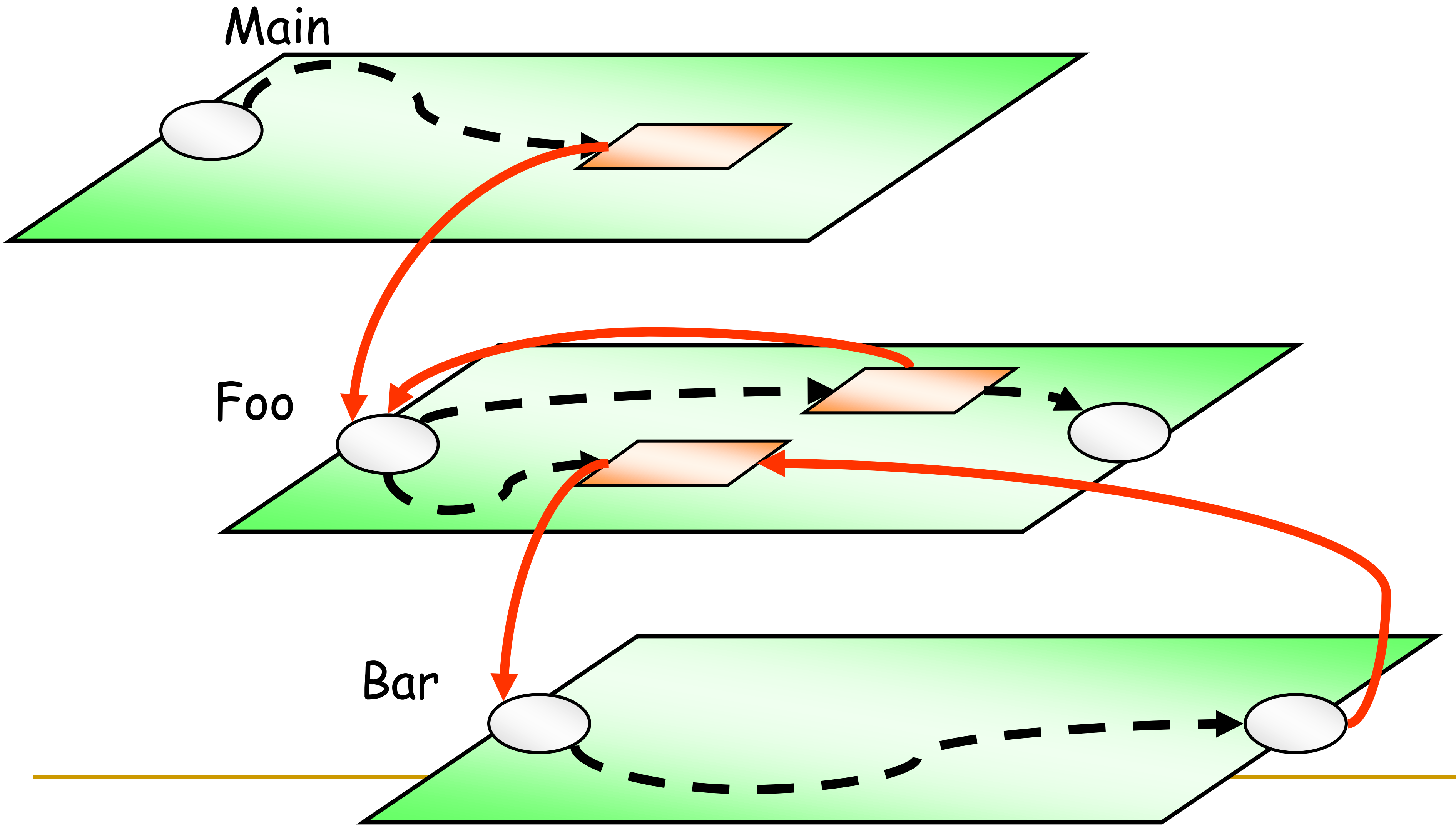
- Nodi (internal state)
- Box (procedure call/return)



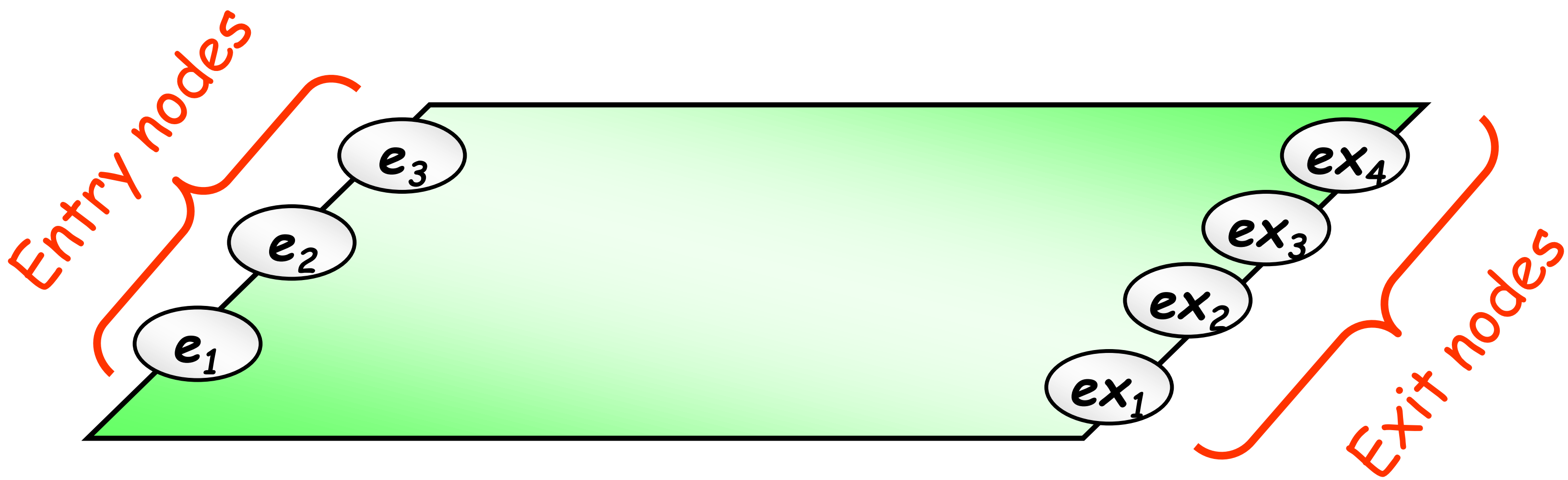
Esempio: semplice programma



Modello corrispondente



Nodi Entry e Exit

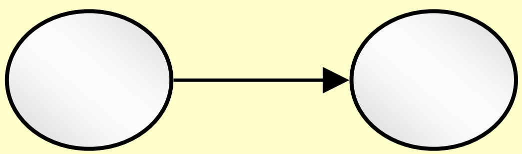


parametri procedura

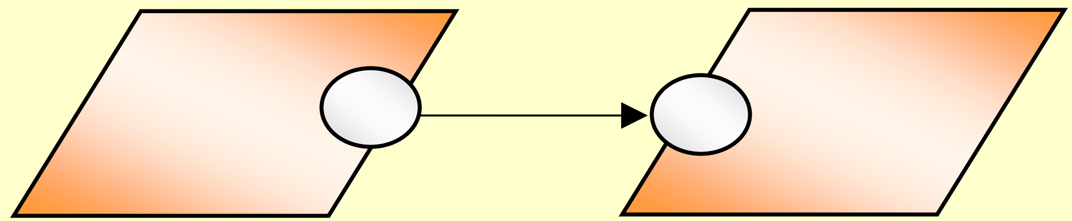
valori restituiti (return)

Archi

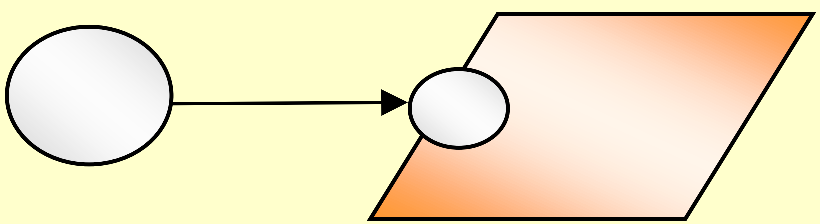
Node-to-Node



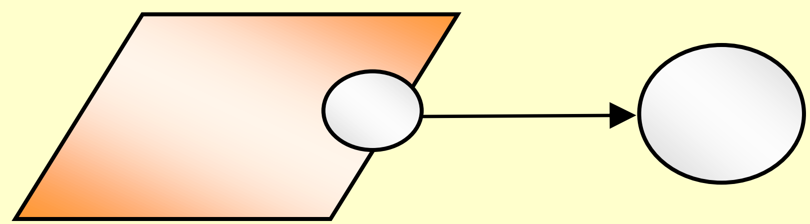
Box-to-Box



Node-to-Box



Box-to-Node



Chiamata (call) e fine chiamata (return)

- una call è una coppia (u,e) dove
  - u è un box
  - e è un nodo entry di un modulo corrispondente al box u
- un return è una coppia (u,x) dove
  - u è un box
  - x è un nodo exit di un modulo corrispondente al box u

Semantica

- modello appiattito corrispondente a RSM (sistema di transizione a stati infiniti)
- ogni stato è $(b_1 \ldots b_m\ u)$ where:
  - $b_1 \ldots b_m$ è il contenuto del call-stack (sequenza di box visitate nella sequenza di chiamate)
  - u è un nodo della macchina corrispondente alla procedura attualmente in esecuzione (contenente l'istruzione correntemente puntata dal program counter)
- le proposizioni atomiche vere a $(b_1 \ldots b_m\ u)$ sono esattamente quelle che etichettano u

Semantica

- le transizioni riflettono la semantica data per gli stati:
  - entrare in un box b corrisponde a fare il push di b nel call-stack e quindi chiamare il modulo corrispondente

    $$(b_1 \ldots b_m\ u) \longrightarrow (b_1 \ldots b_m\ b\ e)$$

  - uscire da un box corrisponde a un pop dal call-stack e fare il return dal modulo corrispondente al box b

    $$(b_1 \ldots b_m\ b\ x) \longrightarrow (b_1 \ldots b_m\ u)$$

Reachability

- u,v vertici di $\mathcal{M}$:
  v è raggiungibile da u iff
  $(b_1 \ldots b_m b_{m+1} \ldots b_n v)$ è raggiungibile da $(b_1 \ldots b_m\, u)$ nell'appiattimento di $\mathcal{M}$

- l'algoritmo per la raggiungibilità in RSM calcola delle summary relation per ogni modulo
  - relazioni che riassumono la relazione tra entry e exit di ogni modulo

Summary relations (symbolic algorithm)

- per ogni M_i, $R_i \subseteq V_i x V_i$ indichiamo con $R_i(x,y)$ che
  "y è raggiungibile da x"
- computazione forward (per ogni entry x e ogni vertice y):
  - $R_i(x,x)$ vale
  - $R_i(x,y) := \vee_z (Edge(z,y) \wedge R_i(x,z))$ (y non è un return)
  - $R_i(x,y) := \vee_e (R_j(e,t) \wedge R_i(x,(z,e)))$

  (y è un return (z,t) da M_j e (z,e) è una sua matching call)

- computazione backward (per ogni vertice x e ogni exit y):
  - $R_i(y,y)$ vale
  - $R_i(x,y) := \vee_z (Edge(x,z) \wedge R_i(z,y))$ (x non è una call)
  - $R_i(x,y) := \vee_t (R_j(e,t) \wedge R_i((z,t),y))$

  (x è una call (z,e) a M_j e (z,t) è un suo matching return)

Mixing backward/forward

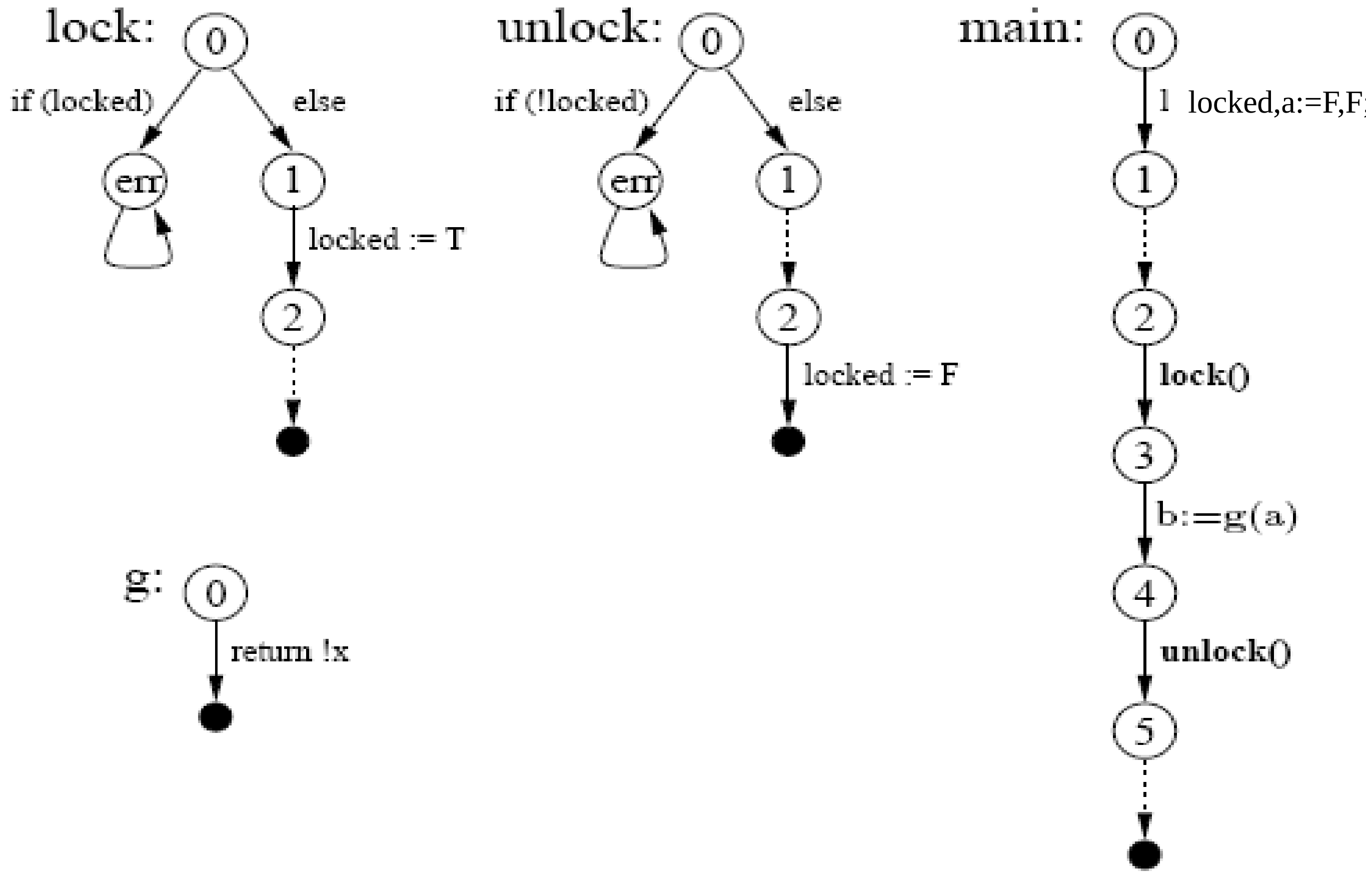
- possiamo pre-computare $R_i(x,y)$
  - se #exits < #entries di M_i
    - calcola $R_i(x,y)$ soltanto per ogni vertice x e ogni exit y di M_i (backward)
  - altrimenti
    - calcola $R_i(x,y)$ solo per ogni entry x e ogni vertice y di M_i (forward)
- l'algoritmo risultante richiede tempo $O(|M| \cdot t^2)$ e spazio $O(|M| \cdot t)$ [AEY'01]

  ($t = \max_i \min \{\#exits_i, \ \#entries_i\}$)

Automi pushdown

- Automa pushdown (PDA)
  - Stato locale di una procedura (variabili locali + program counter) codificato con simboli stack
  - Stati del grafo di controllo corrispondono a variabili globali e valori di return
- Chiamata procedura corrisponde a operazione di push
- Return da una procedura corrisponde a operazione di pop

Flow-graph



lock:
0
if (locked)
else
err
1
locked := T
2
unlock:
0
if (!locked)
else
err
1
2
locked := F
main:
0
1 locked,a:=F,F;
1
2
lock()
3
b:=g(a)
4
unlock()
5
g:
0
return !x

PDA corrispondente

(1)	$\langle (\mathtt{T}), (lock_0) \rangle \hookrightarrow \langle (\mathtt{T}), (err) \rangle$	(l è locked)
(2)	$\langle (\mathtt{F}), (lock_0) \rangle \hookrightarrow \langle (\mathtt{F}), (lock_1) \rangle$	
(3)	$\langle (l), (lock_1) \rangle \hookrightarrow \langle (\mathtt{T}), (lock_2) \rangle$	$l \in \{\mathtt{T}, \mathtt{F}\}$
(4)	$\langle (l), (lock_2) \rangle \hookrightarrow \langle (l), \varepsilon \rangle$	$l \in \{\mathtt{T}, \mathtt{F}\}$
(5)	$\langle (l), (err) \rangle \hookrightarrow \langle (l), (err) \rangle$	$l \in \{\mathtt{T}, \mathtt{F}\}$
(6)	$\langle (\mathtt{F}), (unlock_0) \rangle \hookrightarrow \langle (\mathtt{F}), (err) \rangle$	
(7)	$\langle (\mathtt{T}), (unlock_0) \rangle \hookrightarrow \langle (\mathtt{T}), (unlock_1) \rangle$	
(8)	$\langle (l), (unlock_1) \rangle \hookrightarrow \langle (l), (unlock_2) \rangle$	$l \in \{\mathtt{T}, \mathtt{F}\}$
(9)	$\langle (l), (unlock_2) \rangle \hookrightarrow \langle (\mathtt{F}), \varepsilon \rangle$	$l \in \{\mathtt{T}, \mathtt{F}\}$
(10)	$\langle (l), (g_0, x) \rangle \hookrightarrow \langle (l, \neg x), \varepsilon \rangle$	$l, x \in \{\mathtt{T}, \mathtt{F}\}$
(11)	$\langle (l), (main_0, a, b) \rangle \hookrightarrow \langle (\mathtt{F}), (main_1, \mathtt{F}, b) \rangle$	$a, b, l \in \{\mathtt{T}, \mathtt{F}\}$
(12)	$\langle (l), (main_1, a, b) \rangle \hookrightarrow \langle (l), (main_2, a, b) \rangle$	$a, b, l \in \{\mathtt{T}, \mathtt{F}\}$
(13)	$\langle (l), (main_2, a, b) \rangle \hookrightarrow \langle (l), (lock_0)\,(main_3, a, b) \rangle$	$a, b, l \in \{\mathtt{T}, \mathtt{F}\}$
(14)	$\langle (l), (main_3, a, b) \rangle \hookrightarrow \langle (l), (g_0, a)\,(main_{3'}, a, b) \rangle$	$a, b, l \in \{\mathtt{T}, \mathtt{F}\}$
(15)	$\langle (l, r), (main_{3'}, a, b) \rangle \hookrightarrow \langle (l), (main_4, a, r) \rangle$	$a, b, l, r \in \{\mathtt{T}, \mathtt{F}\}$
(16)	$\langle (l), (main_4, a, b) \rangle \hookrightarrow \langle (l), (unlock_0)\,(main_5, a, b) \rangle$	$a, b, l \in \{\mathtt{T}, \mathtt{F}\}$
(17)	$\langle (l), (main_5, a, b) \rangle \hookrightarrow \langle (l), \varepsilon \rangle$	$a, b, l \in \{\mathtt{T}, \mathtt{F}\}$

Model-checking con PDA

- Automi pushdown permettono
  - modellazione accurata flusso di controllo
  - variabili modellate su domini finiti
- Problema di decisione di base:
  - raggiungibilità
- Algoritmi di soluzione
  - forward/backward
  - esplicito/simbolico

Algoritmo backward (pre*) -- esplicito

- Siano
  - $P=(Q,Q_0,\Gamma,\Gamma_0, \Delta)$ automa pushdown
  - $F \subseteq Q$ insieme target

- Algoritmo di saturazione che
  - costruisce automa finito $A^{pre}=(Q \cup \{q_{fin}\},Q,\Gamma,\Delta^{pre},\{q_{fin}\})$
  - mantenendo la proprietà

    A^{pre} accetta w a partire da q sse

    in P da (q,w) possiamo raggiungere (q',w') con $q' \in F$

Algoritmo backward (pre*) -- esplicito

- Proprietà: A^{pre} accetta w a partire da q sse
  in P da (q,w) possiamo raggiungere (q',w') con q'$\in$F

- Passi:
  - inizialmente si inseriscono transizioni che portano da ogni stato in F a q_{fin}
    (quindi, l'automa A^{pre} corrente accetta Γ* a partire da ogni stato in F)
  - ad ogni iterazione, transizioni aggiunte in base a regola:

$$\boxed{\begin{array}{l}\text{If } \langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle \text{ and } p' \xrightarrow{w}^{*} q \text{ in the current automaton,}\\ \text{add a transition } (p, \gamma, q).\end{array}}$$

- Costruito l'automa, si ha che
  il target F è raggiungibile in P sse A^{pre} accetta $\gamma \in \Gamma_0$ a partire da un $q \in Q_0$

Algoritmo forward (post*) --esplicito

- Algoritmo di saturazione che
  - costruisce automa finito A^{post} con insieme di stati $Q \cup Q' \cup \{q_f\}$ dove
    - q_f unico stato finale
    - Q' contiene un nuovo stato $q_{p',\gamma'}$ per ogni coppia (p',γ') tale che $(p, \gamma, p', \gamma'\gamma'')$ è un push dell'automa pushdown
  - mantenendo la proprietà

  A^{post} accetta w a partire da q sse
  (q,w) è raggiungibile a partire da qualche $(q_0,\gamma_0) \in Q_0 x \Gamma_0$

Algoritmo forward (post*) --esplicito

- Proprietà: A^{post} accetta w a partire da q sse
  (q,w) è raggiungibile a partire da qualche $(q_0,\gamma_0) \in Q_0 x \Gamma_0$

- Passi:
  - inizialmente si inseriscono transizioni che portano da Q_0 a q_{fin} su Γ_0
    (A^{post} accetta il linguaggio Γ_0 da ogni stato in Q_0)
  - Ad ogni iterazione, transizioni aggiunte in base alle regole:

(i) If $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$ and $p \xrightarrow{\gamma}{}^* q$ in the current automaton, add a transition (p', ε, q).

(ii) If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and $p \xrightarrow{\gamma}{}^* q$ in the current automaton, add a transition (p', γ', q).

(iii) If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and $p \xrightarrow{\gamma}{}^* q$ in the current automaton, first add $(p', \gamma', q_{p',\gamma'})$ and then $(q_{p',\gamma'}, \gamma'', q)$.

- Target F è raggiungibile in P sse
  A^{post} accetta un linguaggio non vuoto a partire da uno stato in F

Tool: Moped

http://www.fmi.uni-stuttgart.de/szs/tools/moped/

- Accetta in input programmi booleani, programmi remopla (C-like) oppure automi pushdown
- Verifica:
  - Raggiungibilità (pre* e post*)
  - Proprietà LTL
- Implementa tecniche di model-checking simbolico