Model checking simbolic. Diagrame de decizie binare

22 octombrie 2004

Model checking cu reprezentare explicită a spațiului stărilor

Algoritmii de până acum: consideră individual fiecare stare
⇒ dimensiunea spațiului stărilor limitează aplicabilitatea (dacă o stare are n biți, rezultă direct câte putem reprezenta în memorie)
– tipic, limitat la câteva milioane de stări

Dacă spațiul stărilor atinse e mic față de spațiul potențial complet, se poate încerca o codificare a stărilor pe număr mai mic de biți (*bitstate hashing*, metodă folosită în SPIN). Metoda e însă doar o *aproximație*: dacă se ajunge la o stare cu cod deja întâlnit, explorarea se oprește (deși poate starea e diferită).
⇒ o parte din spațiul stărilor poate rămâne neexplorată

Explorarea cu stări și mulțimi

Calculul stărilor care pot fi atinse din stările inițiale (**EF** *true*)
– prin traversare a grafului pornind de la stările inițiale
– R: mulțimea stărilor explorate; F: *frontiera* stărilor atinse

Cu *stări individuale*	Cu *mulțimi de stări*
$R = \emptyset$; $F = S_0$	$R = \emptyset$; $F = S_0$
while $(F \neq \emptyset)$	**while** $(F \not\subseteq R)$
choose $s \in F$;	$R \leftarrow R \cup F$
$F \leftarrow F \setminus \{s\}$; $R \leftarrow R \cup \{s\}$	$F = \{s' \in S \mid \exists s \in F \,.\, s \to s'\}$
forall s' with $s \to s'$	/* eventual $F = F \setminus R$
if $s' \notin F \cup R$	* cu test $F \neq \emptyset$ */
$F \leftarrow F \cup \{s'\}$	

⇒Algoritmul se exprimă mult mai simplu dacă se poate calcula *într-o singură operație* mulțimea *succesorilor* unei mulțimi de stări
⇒mulțimea R a stărilor explorate crește la fiecare iterație, dar e finită

Model checking simbolic

- O nouă abordare: explorarea mulțimilor de stări
  – ideea: o mulțime poate fi uneori reprezentată (printr-o formulă) într-un spațiu mai mic decât explicit pentru fiecare stare în parte
  – reprezentare eficientă pt. mulțimi de stări, relație de tranziție [McMillan'92]
  – cu ajutorul diagramelor de decizie binare (BDD) [Bryant'86]
- idee cheie: Operarea cu *mulțimi* de stări
  – folosită și în cazul în care mulțimea stărilor este infinită (sisteme în timp continuu, sisteme hibride)
- idee cheie: prelucrarea iterativă până când nu se mai produc modificări
  ⇒noțiunea de *punct fix*

Reprezentări de punct fix

Def: $x \in D$ este *punct fix* pentru $f : D \to D$ dacă $f(x) = x$. Def: O *latice* e o mulțime parțial ordonată în care orice submulțime finită are un cel mai mic majorant și un cel mai mare minorant
Ex: mulțimea părților $\mathcal{P}(S)$ a lui S, cu relația de incluziune $\subseteq$

– Lucrăm cu funcții $\tau : \mathcal{P}(S) \to \mathcal{P}(S)$ peste *laticea* $\mathcal{P}(S)$

– Privim $S' \subseteq S$ ca un *predicat* peste S: $S'(s) =$ *true* $\Leftrightarrow s \in S'$
în particular: $\emptyset =$ *false*, $S =$ *true*
⇒ $\tau : \mathcal{P}(S) \to \mathcal{P}(S)$ este o *transformare de predicate*
Def:

- τ este *monoton* dacă $P \subseteq Q \Rightarrow \tau(P) \subseteq \tau(Q)$
- τ este *continuu la uniune* dacă pentru orice șir $P_1 \subseteq P_2 \subseteq \ldots$ avem $\tau(\cup_i P_i) = \cup_i \tau(P_i)$
- τ este *continuu la intersecție* dacă pentru orice șir $P_1 \supseteq P_2 \supseteq \ldots$ avem $\tau(\cap_i P_i) = \cap_i \tau(P_i)$

Teoreme de punct fix

O transformare de predicate monotonă pe $\mathcal{P}(S)$ are întotdeauna
– un punct fix minimal, notat $\mu Z.\tau(Z)$
– și un punct fix maximal, notat $\nu Z.\tau(Z)$ [Tarski]

Dacă S e finită și τ e monotonă, atunci τ e continuă la uniune și la intersecție.

τ monotonă ⇒ $\tau^i(\mathit{False}) \subseteq \tau^{i+1}(\mathit{False})$ și $\tau^i(\mathit{True}) \supseteq \tau^{i+1}(\mathit{True})$

Dacă τ e monotonă și S e finită, există $i, j \geq 0$ astfel ca
$\forall k \geq i$, $\tau^k(\mathit{False}) = \tau^i(\mathit{False})$ și $\forall k \geq j$, $\tau^k(\mathit{True}) = \tau^j(\mathit{True})$

Dacă τ e monotonă și S e finită, există $i, j \geq 0$ astfel ca
$\mu Z.\tau(Z) = \tau^i(\mathit{False})$ și $\nu Z.\tau(Z) = \tau^j(\mathit{True})$

Calculul punctului fix minimal/maximal

function *Lfp*(τ : *Trans*) : *Pred*
  Q := *False*;
  Q' := $\tau(Q)$;
  while ($Q' \neq Q$) **do**
    Q := Q';
    Q' := $\tau(Q)$;
  return Q;

function *Gfp*(τ : *Trans*) : *Pred*
  Q := *True*;
  Q' := $\tau(Q)$;
  while ($Q' \neq Q$) **do**
    Q := Q';
    Q' := $\tau(Q)$;
  return Q;

Relații de punct fix pentru CTL

Identificăm formula CTL f cu mulțimea de stări $\{s \mid M, s \models f\}$

- **AF** $f = \mu Z \,.\, f \vee$ **AX** Z **EF** $f = \mu Z \,.\, f \vee$ **EX** Z
- **AG** $f = \nu Z \,.\, f \wedge$ **AX** Z **EG** $f = \nu Z \,.\, f \wedge$ **EX** Z
- **A** $[f_1$ **U** $f_2] = \mu Z \,.\, f_2 \vee (f_1 \wedge$ **AX** $Z)$
- **E** $[f_1$ **U** $f_2] = \mu Z \,.\, f_2 \vee (f_1 \wedge$ **EX** $Z)$
- **A** $[f_1$ **R** $f_2] = \nu Z \,.\, f_2 \wedge (f_1 \vee$ **AX** $Z)$
- **E** $[f_1$ **R** $f_2] = \nu Z \,.\, f_2 \wedge (f_1 \vee$ **EX** $Z)$

punct fix minimal: proprietăți de evoluție: **F**
punct fix maximal: proprietăți de siguranță (invarianți): **G**

Algoritmul de model checking simbolic

Prin descompunere după structura formulei.
Check(f) returnează $\{s \in S \mid M, s \models f\}$

Check(p) $= \{s \in S \mid p \in L(s)\}$ propoziții atomice
Check($\neg f$) $= S \setminus$ *Check*(f)   complement
Check($f \wedge g$) = *Check*(f) $\cap$ *Check*(g)   intersecție
Check(**EX** f) = *CheckEX*(*Check*(f))
  CheckEX($f(\bar{v})$) $= \exists \bar{v}' \,.\, [f(\bar{v}') \wedge R(\bar{v}, \bar{v}')]$ produs relațional
Check(**E** $[f$ **U** $g]$) = *CheckEU*(*Check*(f), *Check*(g))
  folosind **E** $[f_1$ **U** $f_2] = \mu Z \,.\, f_2 \vee (f_1 \wedge$ **EX** $Z)$ și funcția *Lfp*
Check(**EG** f) = *CheckEG*(*Check*(f))
  folosind **EG** $f = \nu Z \,.\, f \wedge$ **EX** Z și funcția *Gfp*

Aceste operații de bază se pot implementa cu BDD-uri

Diagrame de decizie binare

Binary Decision Diagrams (BDDs)

- o reprezentare compactă și canonică a funcțiilor boolene
- cu algoritmi eficienți de manipulare
  [R. Bryant, "Graph-based algorithms for boolean function manipulation", *IEEE Transactions on Computers*, 1986]
- impact deosebit asupra verificării formale:
  ACM Kanellakis Award for Theory & Practice, 1998
  – Randal E. Bryant: BDDs ('86)
  – Edmund M. Clarke, E. Allen Emerson: model checking ('81)
  – Ken McMillan: symbolic model checking ('92)

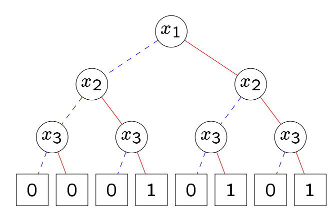
Funcții boolene. Reprezentări

Obiectul de lucru: funcții Boolene $f : B^n \to B$

- reprezentări uzuale: tabele de adevăr, diagrame Karnaugh, sumă canonică de mintermi – dimensiune exponentială
- sumă redusă de produse, factorizări, etc.
  – exponențiale pentru anumite funcții comune (ex. paritate)
- operații elementare pot rezulta în creștere exponențială (ex. complementarea)
- reprezentări necanonice ⇒ e dificil de testat:
  – echivalența (după transformări în proiectarea circuitelor)
  – satisfiabilitatea: $\exists x_1, \cdots x_n \,.\, f(x_1, \cdots, x_n) = 1$?
  $\forall x \,.\, f_1(x) = f_2(x) \equiv \neg \exists x \,.\, f_1(x) \oplus f_2(x) = 1$

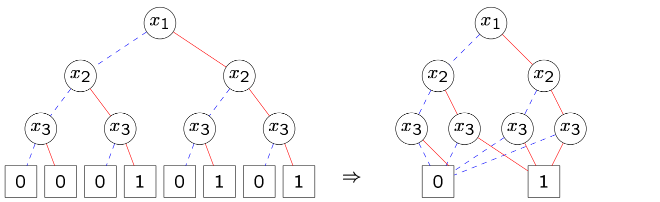
Arbori de decizie binari

- noduri terminale: valoarea funcției (0 sau 1)
- noduri neterminale: variabile
- ramuri: $low(v)$ (stânga) / $high(v)$ (dreapta): atribuire 0/1 pt. variabila din nod

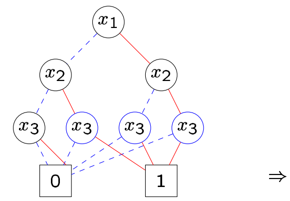


BDD-uri: obținute prin 3 reguli de reducere

Reducerea nr. 1: Comasarea nodurilor terminale

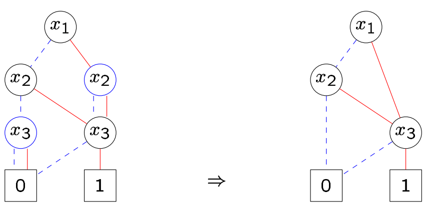


Reducerea nr. 2: Comasarea nodurilor izomorfe



$f(n_1) = f(n_2) \Rightarrow$ comasează n_1 și n_2

Reducerea nr. 3: Eliminarea testelor inutile



$low(n) = high(n) \Rightarrow$ elimină testarea lui n

Proprietăți esențiale

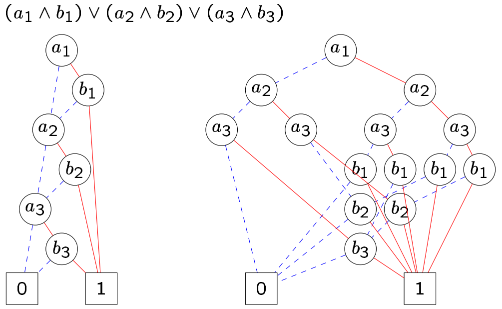
Cele 3 reguli se pot aplica indiferent de ordonarea variabilelor.
Pentru a defini o BDD *ordonată* (Ordered BDD = OBDD)
se impune o condiție esențială:
Pe toate căile de la vârf la nodurile terminale, variabilele apar
în aceeași ordine (= există o ordonare globală a variabilelor).

Teoremă: Pentru orice funcție, reprezentarea ca BDD *ordonată*,
redusă cf. regulilor 1-3 e unică pâna la un izomorfism.
$\Rightarrow$ reprezentare *canonică*
$\Rightarrow$ testare de echivalență sau satisfiabilitate în $O(1)$

Obs:
Un subgraf cu rădăcina într-un nod de BDD este tot o BDD.

Efectul ordonării variabilelor

Funcția: $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$



Creștere liniară: $2(n+1)$ Creștere exponențială: 2^{n+1}

Algoritmi cu BDD-uri: Apply

function *Apply*$(f, g : OBDD, op : Operator) : OBDD$
if *is_leaf*$(f) \wedge$ *is_leaf*(g) **return** $op(f, g)$;
elsif (f, g, op, h) in apply_cache **return** h;
else
  $x :=$ *topvar*(f) /* variabila din vârful lui f */
  $y :=$ *topvar*(g)
  if (*ord*$(x) =$ *ord*(y)) /* x = y = aceeași variabilă */
    $h :=$ *find_bdd*$(x, Apply(f|_{x=0}, g|_{x=0}, op), Apply(f|_{x=1}, g|_{x=1}, op))$
    /* *find_bdd* creează un nou BDD dacă nu există deja */
  elsif (*ord*$(x) <$ *ord*(y)) /* x inaintea lui y în ordine */
    $h :=$ *find_bdd*$(x, Apply(f|_{x=0}, g, op), Apply(f|_{x=1}, g, op))$
  else $h :=$ *find_bdd*$(y, Apply(f, g|_{y=0}, op), Apply(f, g|_{y=1}, op))$
  insert (f, g, op, h) in apply_cache
  return h

Algoritmi cu BDD-uri: Produs relational

function *Relprod*(f, g : *OBDD*, E : *varset*) : *OBDD*
if $f =$ *false* $\lor\ g =$ *false* **return** *false*
elsif $f =$ *true* $\land\ g =$ *true* **return** *true*
elsif (f, g, E, h) in relprod_cache **return** h
else
  $x :=$ *topvar*(f) /* variabila din vârful lui f */
  $y :=$ *topvar*(g)
  $z :=$ *topmost*(x, y) /* prima în ordinea variabilelor */
  $h_0 :=$ *RelProd*$(f\mid_{z=0}, g\mid_{z=0}, E)$
  $h_1 :=$ *RelProd*$(f\mid_{z=1}, g\mid_{z=1}, E)$
  if $z \in E$ $h :=$ *bdd_or*(h_0, h_1) /* $\exists z\ .\ h = h_0 \lor h_1$ */
  else $h :=$ *bdd_if_then_else*(z, h_1, h_0)
  insert (f, g, E, h) in relprod_cache
  return h

Complexitatea algoritmilor

• Reducere (la forma canonică)	$O(\lvert G\rvert \cdot \log \lvert G\rvert)$
• Apply ($f_1 \langle op \rangle f_2$)	$O(\lvert G_1\rvert \cdot \lvert G_2\rvert)$
• Restrict ($f\mid_{x_i=b}$)	$O(\lvert G\rvert \cdot \log \lvert G\rvert)$
• Compose ($f_1\mid_{x_i=f_2}$)	$O(\lvert G_1\rvert^2 \cdot \lvert G_2\rvert)$
• Satisfy-one (un $\bar{x}$ cu $f(\bar{x}) = 1$)	$O(n)$
• Satisfy-count ($\lvert\{\bar{x} \mid f(\bar{x}) = 1\}\rvert$)	$O(\lvert G\rvert)$

Factorii logaritmici pot fi eliminați
(prin algoritmi mai sofisticați sau hashing).

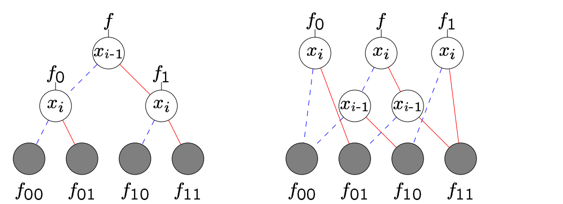
Produsul relațional poate avea complexitate exponențială.

Implementare

- Există implementări mature de biblioteci (pachete) de BDD-uri
- Într-o aplicație tipică, multe BDD-uri au subgrafuri comune
  ⇒ pointeri într-un unic graf multi-rădăcină
- Gestiunea memoriei: cu contor de referință și garbage-collection
- Un vast număr de optimizări și euristici:
  – metode de dispunere și traversare pt. caching avantajos
  – calcul paralel și distribuit, etc.

Reordonarea dinamică a variabilelor

- Ordonarea variabilelor - efect critic asupra dimensiunii
- Există funcții cu reprezentări exponențiale indiferent de ordonare
  (ex. bitul mijlociul al unui multiplicator [Bryant'91])
- BDD-urile evoluează în timpul execuției aplicației
  ⇒ f. importantă reordonarea *dinamică*
  – efectuată transparent pentru algoritmii de verificare
  – reordonarea nivelelor adiacente nu modifică pointerii externi



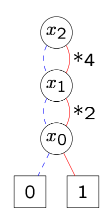
Variante de BDD-uri

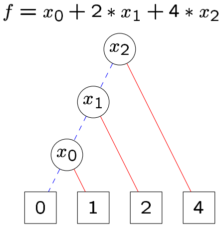
- Relaxarea condiției de ordonare: FreeBDDs
  – variabilele apar în orice ordine, fiecare doar o dată
  – $\forall$ atribuire la $\bar{v}$, aceeași ordine în toate funcțiile
  – reprezentări bune pt. unele funcții cu OBDD-uri exponențiale
- Alegerea relației de descompoziție funcțională
  pentru OBDD: descompunerea Boole-Shannon:
  $$f = \bar{x} \land f_{\bar{x}} \lor x \land f_x$$
  $f_{\bar{x}} = f\mid_{x=0}$: cofactorul negativ
  $f_x = f\mid_{x=1}$: cofactorul pozitiv

  Alte relații de descompoziție:
  – $f = f_{\bar{x}} \oplus x \land f_{\delta x}$ descompunere Reed-Muller
  – $f = f_x \oplus \bar{x} \land f_{\delta x}$ descompunere Davio pozitivă
  unde $f_{\delta x} = f_x \oplus f_{\bar{x}}$
- Multiterminal BDDs: extensie cu noduri terminale arbitrare (întregi, etc.)

Diagrame pentru operații aritmetice

Exemplu: codificare numerică pe 3 biți: $f = x_0 + 2 * x_1 + 4 * x_2$





Edge-Valued BDD (EVBDD)
ponderi multiplicative pe muchii

Binary Moment Diagram (BMD)
$f = f_{\bar{x}} + x \cdot (f_x - f_{\bar{x}})$

Hybrid Decision Diagrams = BDD + BMD + MTBDD

Word-level Verification

Cazul tipic: demonstrarea corespondenței dintre:
– o descriere matematică (numerică) F_{num} (ex. operatia de înmulțire)
– un circuit f_{bit} cu operanzi în reprezentare binară

Fie *Num* : $B^n \rightarrow$ **Z** (sau **R**) funcția care descrie semnificația numerică a unui număr reprezentar binar.

Trebuie verificat:

$$Num(f_{bit}(\vec{x_1}, \vec{x_2})) = F_{num}(Num(\vec{x_1}), Num(\vec{x_2}))$$

⇒ reprezentare eficientă pentru funcții pe biți cât și numerice
⇒ se utilizează EVBDD, BMD, HDD, etc.

Aplicații

Principala aplicație: in CAD și verificare formală.

În general: pentru reprezentarea compactă a datelor cu o anumită regularitate/structură comună, dificil de exprimat analitic.

- teoria codurilor
- structuri mari de date, indexare
- biologie computațională

Exemplu: sistem de filtrare publish-subscribe în timp real
– flux mare de mesaje; sute de mii de criterii de filtrare
– criteriile de filtrare reprezentate ca structură de BDD-uri (propoziții atomice: atribut ⟨*relație*⟩ valoare)
– mesajele noi sunt filtrate prin algoritmi de evaluare a BDD-urilor
– subcriteriile comune din BDD: evaluate o singură dată

Model checking simbolic: Istoric

Ken McMillan (CMU, 1987): ideea de a reprezenta sisteme în mod simbolic cu BDD-uri.
⇒ [Burch, Clarke, Dill, McMillan, Hwang: *Symbolic model checking: 10^{20} states and beyond*, 1990]
⇒ verificatorul SMV; teza de doctorat: *Symbolic Model Checking: An Approach to the State Explosion Problem* (CMU, 1992)
Independent:
– Coudert, Berthet, Madre [1989, 1990]
– Pixley [Motorola, 1990]

Model checking simbolic cu BDD-uri

Reprezentare: codificare binară pentru stări și propozitii atomice
⇒ BDD-uri pentru mulțimi de stări, relație de tranziție

$Check(p) = \{s \in S \mid p \in L(s)\}$	$bdd\_if\_then\_else(p, 1, 0)$
$Check(\neg f) = S \setminus Check(f)$	bdd_not
$Check(f \wedge g) = Check(f) \cap Check(g)$	bdd_and
$Check(\mathbf{EX}\, f) = CheckEX(Check(f))$	
$\quad CheckEX(f(\bar{v})) = \exists \bar{v}' \,.\, [f(\bar{v}') \wedge R(\bar{v}, \bar{v}')]$	$RelProd(f, R, \bar{v}')$
$Check(\mathbf{E}\,[f\, \mathbf{U}\, g]) = CheckEU(Check(f), Check(g))$	
$\quad \mathbf{E}\,[f_1\, \mathbf{U}\, f_2] = \mu Z \,.\, f_2 \vee (f_1 \wedge \mathbf{EX}\, Z)$	algoritmul *Lfp*
$Check(\mathbf{EG}\, f) = CheckEG(Check(f))$	
$\quad \mathbf{EG}\, f = \nu Z \,.\, f \wedge \mathbf{EX}\, Z$	algoritmul *Gfp*

Partiționarea relației de tranziție

O relație de tranziție monolitică poate deveni foarte mare
principala dificultate: produsul relațional

- partiționare *disjunctivă* (sisteme asincrone)

$$R(\bar{v}, \bar{v}') = R_1(\bar{v}, \bar{v}') \vee \cdots \vee R_n(\bar{v}, \bar{v}')$$

prin distributivitate: $\exists \bar{v}'[f(\bar{v}') \wedge R(\bar{v}, \bar{v}')] =$

$$= \exists \bar{v}'[f(\bar{v}') \wedge R_1(\bar{v}, \bar{v}')] \vee \cdots \vee \exists \bar{v}'[f(\bar{v}') \wedge R_n(\bar{v}, \bar{v}')]$$

- partiționare conjunctivă (sisteme sincrone)

$\exists$ nu distribuie față de $\wedge$, dar se poate exploata localitatea (dacă R_i nu depind de toate variabilele din $\bar{v}'$):

$$R(\bar{v}, \bar{v}') = R_1(\bar{v}, v_1') \wedge \cdots \wedge R_n(\bar{v}, v_n')$$

$\exists \bar{v}'[f(\bar{v}') \wedge R(\bar{v}, \bar{v}')] =$

$$= \exists v_n'[\cdots \exists v_1'[f(\bar{v}') \wedge R_0(\bar{v}, v_1'] \wedge R_1(\bar{v}, v_1')] \cdots \wedge R_n(\bar{v}, v_n')]$$

(conjuncție și cuantificare succesivă pentru fiecare componentă)

Model checking simbolic cu fairness

Restricția de fairness: $F = \{P_1, P_2, \cdots, P_n\}$, cu $P_i \subseteq S$

EG f e adevărată în mulțimea maximală Z cu proprietățile:
– toate stările din Z satisfac f
– $\forall P_k \in F, s \in Z$ există un drum din s într-o stare din $Z \cap P_k$ (trecând doar prin stări care satisfac f)

⇒ exprimare ca punct fix, poate fi calculată simbolic:

$$\mathbf{EG}_{\,fair} f = \nu Z \,.\, f \wedge \bigwedge_{i=1}^{n} \mathbf{EX}\, \mathbf{E}\,[f\, \mathbf{U}\, (Z \wedge P_k)]$$

La fel cu cele definite pentru reprezentarea explicită:

$$\mathbf{EX}_{\,fair}\, f = \mathbf{EX}\,(f \wedge fair)$$

$$\mathbf{EU}_{\,fair}\,(f, g) = \mathbf{EU}\,(f, g \wedge fair)$$

Generarea de contraexemple

Principalele avantaje ale tehnicii de model checking:
– metodă complet automată
– generează contraexemple care identifică erorile

- formule existențiale (**E**) : produce o traiectorie "martor" (*witness*) pentru care formula este adevărată
- formule universale (**A**): produce un contraexemplu
- contraexemplul pentru o formulă universală este traiectoria martor pentru negația ei (formula existențială duală)

Traiectorie pentru **EF** f

– punct fix minimal: $\mathbf{EF}\, f = \mu Z \,.\, f \vee \mathbf{EX}\, Z$
– se obțin și se rețin aproximări succesive $f = Q_0 \subseteq Q_1 \subseteq \ldots \subseteq Q_k$
– Q_k: mulțimea stărilor din care se poate atinge f în cel mult k pași
– se găsește o intersecție $Q_k \cap S_0 \neq \emptyset$
(o primă secvență de traversare: înapoi, simbolică)
– se alege $s_k \in S_0 \cap Q_k$
– se calculează mulțimea *Succ*(s_k) a succesorilor lui s_k
– ea trebuie să aibă o intersecție cu Q_{k-1} (din s_k se atinge f în cel mult k pași, deci există un succesor din care se ajunge în $k-1$ pași)
– se alege $s_{k-1} \in$ *Succ*$(s_k) \cap Q_{k-1}$, etc. până la $Q_0 = f$
(a doua traversare, înainte, prin stări individuale)
– s-a găsit o traiectorie $s_k \to \ldots \to s_0$ care atinge f

Traiectorie pentru **EG** f cu fairness

– calcul prin iterație (*Gfp*): $\mathbf{EG}\, f = \nu Z. f \wedge \bigwedge_{i=1}^{n} \mathbf{EX}\, \mathbf{E}\, [f\, \mathbf{U}\, (Z \wedge P_k)]$ (*)
– iterații interioare (*Lfp*) pentru $\mathbf{E}\, [f\, \mathbf{U}\, (Z \wedge P_k)]$, $\forall P_k$
– în ultima iterație exterioară se reține șirul de aproximări
$Q_0^{P_k} \subseteq Q_1^{P_k} \subseteq \ldots \subseteq Q_i^{P_k} \subseteq \ldots$ pentru care $\mathbf{EG}\, f \wedge P_k$ e atins în i pași
– se alege o stare inițială $s_0 \models \mathbf{EG}\, f$
– din mulțimea de succesori *Succ*(s_0) (vezi **EX** în (*)) se caută s_1 care atinge un P_k în număr minim de pași: $\min i \,.\, Succ(s_0) \cap Q_i^{P_k} \neq \emptyset$
(euristică *greedy* pentru găsirea unei traiectorii scurte)
– se găsește traiectoria: i pași din $s_1 \Rightarrow i-1$ pași din *Succ*(s_1)
$\Rightarrow$ alegem $s_2 \in$ *Succ*$(s_1) \cap Q_{i-1} \ldots \Rightarrow$ se atinge $s_{i+1} \in \mathbf{EG}\,_f \cap P_k$
– eliminăm P_k, continuăm din s_{i+1} până vizităm și ultimul P_j (în s')
– dacă $s' \models \mathbf{EX}\, \mathbf{E}\, [f\, \mathbf{U}\, s_1]$, închidem ciclul spre $s_1 \Rightarrow$ avem martorul
– dacă nu, s' e în altă SCC decât s_1. Repetăm procedeul din s'.
– graful SCC-urilor e aciclic $\Rightarrow$ procedeul e finit.