Reducerea spaţiului stărilor

Metode cu ordonare parţială. Abstracţie

23 noiembrie 2004

Abstracţia: Introducere

Abstracţia: esenţială pentru verificarea sistemelor realiste.
- implică construirea unui sistem *abstract* (cu mai puţine detalii)
- şi stabilirea unei *corespondenţe* între proprietăţile sistemului abstract şi cele ale sistemului original
  - – abstracţii exacte: păstrează valoarea de adevăr
  - – abstracţii conservatoare (aproximative): corectitudinea sistemului abstract implică cea a sistemului real, dar nu invers (contraexemplu în sistemul abstract poate să nu existe în cel real)

Modelul abstract: obţinut fără a-l construi pe cel concret (construirea modelului concret e adesea imposibilă practic, fiind prea mare)
– tehnici de abstracţie *sintactice*
– sau *semantice* (ex. domeniu redus pentru variabile)

Exemple de abstracţii întâlnite

Automate temporizate: graful regiunilor, automatul zonelor
– sunt abstracţii *finite* ale unui sistem infinit
– la mai multe stări din sistemul concret (locaţie, atribuire de ceasuri) corespunde o stare în sistemul abstract

O *specificaţie* e de regulă o *abstracţie* a implementării
– tabloul pentru o formulă LTL e o abstracţie pt. sistemul care o verifică

Relaţiile de *rafinare* (incluziune a limbajelor, simulare) între un sistem concret şi unul mai abstract.

Folosirea pachetelor de 1 bit în modelarea protocolului de la proiectul 1

Reducerea bazată pe conul de influenţă

Abstracţie prin eliminarea variabilelor care nu afectează specificaţia.

Fie un sistem M cu mulţimea de variabile $V = \{v_1, v_2, \cdots, v_n\}$ descris prin ecuaţiile $v_i' = f_i(V)$.
Fie V' mulţimea variabilelor referite în specificaţie.
Conul de influenţă al lui V' = mulţimea minimală $C \subseteq V$ a.î.
– $V' \subseteq C$
– dacă $v_i \in C$, şi f_i depinde de v_j, atunci $v_j \in C$ (închidere tranzitivă)

Construim un nou sistem M' eliminând toate variabilele (şi ecuaţiile lor funcţionale) care nu apar în C.

Invarianţa specificaţiilor CTL*

Demonstrăm că metoda conului de influenţă păstrează valoarea de adevăr a specificaţiilor CTL* (definite peste variabilele din C).

Concret, fie $V = \{v_1, v_2, \cdots v_n\}$ o mulţime de variabile *boolene* şi $M = (S, S_0, R, L)$, cu:
– $S = \{0,1\}^n$ = mulţimea atribuirilor la V; $S_0 \subseteq S$
– $R = \bigwedge_{i=1}^{n}(v_i' = f_i(V))$
– $L(s) = \{v_i | s(v_i) = 1\}$ (variabilele egale cu 1 în s)

Fie V numerotată a.î. $C = \{v_1, \cdots, v_k\}$. Definim $M' = (S', S_0', R', L')$:
– $S' = \{0,1\}^k$ = mulţimea atribuirilor la C
– $S_0 = \{(d_1', \cdots, d_k') | \exists (d_1, \cdots d_n) \in S_0 \text{ cu } d_1' = d_1 \wedge \ldots \wedge d_k' = d_k\}$
– $R' = \bigwedge_{i=1}^{k}(v_i' = f_i(C))$
– $L'(s) = \{v_i | s'(v_i) = 1\}$

Se arată că modelul concret M şi cel abstract M' sunt *bisimilare*.

Program slicing

Noţiune similară dar mai generală, pt. programe [Weiser'79]
– inspirată din operaţiunile mentale făcute la depanare
= calculul fragmentului de program care poate afecta valorile calculate într-un anumit punct de interes (ex. nume variabilă + linie sursă)
– de regulă: fragment executabil, în limbajul sursă
– metodă bazată pe noţiunile de dependenţă de control şi de date

Diverse tipuri de slicing:
– static sau dinamic
– criterii sintactice sau semantice
– traversare înainte sau înapoi a grafului de control
– tipul de dependenţe considerat: date/control; directe/tranzitive
– pe *toate* sau *unele din* drumurile prin graful de control

Abstracţia datelor

– folosită pentru a raţiona despre circuite cu număr mare de biţi sau pentru programe cu structuri de date complexe
– utilă în cazul în care operaţiile de prelucrare a datelor în sistem sunt relativ simple (transfer, nr. mic de operaţii logice/aritmetice)

Ideea: stabilirea unei corespondenţe între domeniul original al datelor şi un domeniu de dimensiune mai mică (ex. cu doar câteva valori)

Exemplu: abstracţia semn

$$h(x) = \begin{cases} - & \text{dacă } x < 0 \\ 0 & \text{dacă } x = 0 \\ + & \text{dacă } x > 0 \end{cases}$$

$\cdot$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

$+$	$-$	0	$+$
$-$	$-$	$-$	$\top$
0	$-$	0	$+$
$+$	$\top$	$+$	$+$

unde $\top = \{-, 0, +\}$

$\Rightarrow$ nu întotdeauna putem avea abstracţie precisă
$\Rightarrow$ domeniul şi funcţia de abstracţie: alegere dificilă, judicioasă

Principiul de generare a sistemului abstract

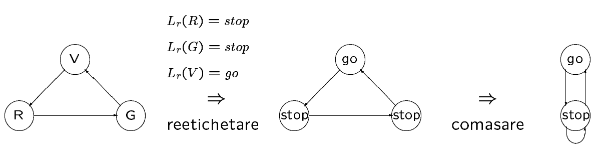
– pt. orice variabilă x, definim o variabilă abstractă $\hat{x}$
– etichetarea stărilor cu propoziţii atomice indicând valoarea abstractă (pt. abstracţia semn: 3 propoziţii p_x^-, p_x^0, p_x^+ pt. fiecare variabilă x, indicând $\hat{x} = "-"$, $\hat{x} = 0$, $\hat{x} = "+"$)
– comasarea tuturor stărilor cu aceleaşi etichete abstracte
$\Rightarrow$ spaţiul abstract al stărilor: 2^{AP}, AP = propoziţiile abstracte

Pentru un model M reprezentat *explicit*, definim modelul abstract (redus) $M_r = (S_r, S_r^0, R_r, L_r)$:
– $S_r = \{L_r(s) \mid s \in S\}$ = etichetările (abstracte) ale stărilor din S.
– $S_r^0 = \{s_r^0 \in S_r \mid \exists s_0 \in S^0 . L_r(s_0) = s_r^0\}$ (etichetările stărilor iniţiale)
– $R_r(s_r, t_r) \Leftrightarrow \exists s, t \in S . R(s,t) \wedge L_r(s) = s_r \wedge L_r(t) = t_r$ (tranziţie între două stări abstracte dacă $\exists$ tranziţie între doi reprezentanţi concreţi)

Se demonstrează: Modelul abstract M_r îl simulează pe cel original M.

Exemplu de abstracţie

Semafor de circulaţie cu 3 stări – redus la două



Obs.: Sistemul abstract poate introduce comportamente noi (ex. sistemul ramâne întotdeauna în "stop")

Abstractii exacte şi aproximări

Fie un sistem reprezentat *implicit*, prin predicate pentru relaţia de tranziţie $\mathcal{R}$ şi stările iniţiale $\mathcal{S}_0$.
Presupunem aceeaşi funcţie de abstracţie pentru toate variabilele, $h : D \to A$ (D = domeniu concret, A = domeniu abstract)

Trebuie să definim $\hat{\mathcal{S}}_0$ şi $\hat{\mathcal{R}}$ pentru sistemul abstract:
$\hat{\mathcal{S}}_0 = \exists x_1 \ldots \exists x_n . \mathcal{S}_0(x_1, \cdots, x_n) \wedge h(x_1) = \hat{x_1} \wedge \cdots \wedge h(x_n) = \hat{x_n}$
Similar definim $\hat{\mathcal{R}}(\hat{x_1}, \cdots \hat{x_n}, \hat{x_1}', \cdots \hat{x_n}')$.
$\Rightarrow$ din $\phi(x_1, \cdots, x_n)$ obţinem $\hat{\phi}(\hat{x_1}, \cdots, \hat{x_n})$ în variabile abstracte

Transformarea $\phi \to \hat{\phi}$: operaţie complexă $\Rightarrow$ o aplicăm (ca şi negaţia) doar la relaţii elementare între variabile (ex. $=, <, >$, etc.).
Definim prin inducţie structurală o abstracţie cu aproximare $\mathcal{A}$:
– $\mathcal{A}(P(x_1, \ldots, x_n)) = \hat{P}(\hat{x_1}, \cdots, \hat{x_n})$, dacă P este relaţie elementară.
– $\mathcal{A}(\neg P(x_1, \ldots, x_n)) = \neg \hat{P}(\hat{x_1}, \cdots, \hat{x_n})$
– $\mathcal{A}(\phi_1 \wedge \phi_2) = \mathcal{A}(\phi_1) \wedge \mathcal{A}(\phi_2)$ – $\mathcal{A}(\phi_1 \vee \phi_2) = \mathcal{A}(\phi_1) \vee \mathcal{A}(\phi_2)$
– $\mathcal{A}(\exists x\, \phi) = \exists \hat{x}\, \mathcal{A}(\phi)$ – $\mathcal{A}(\forall x\, \phi) = \forall \hat{x}\, \mathcal{A}(\phi)$

Abstractii exacte şi aproximări (cont.)

Cu definiţiile anterioare, se demonstrează că $\forall \phi . \hat{\phi} \Rightarrow \mathcal{A}(\phi)$
În particular, $\hat{\mathcal{S}}_0 \Rightarrow \mathcal{A}(\mathcal{S}_0)$ şi $\hat{\mathcal{R}} \Rightarrow \mathcal{A}(\mathcal{R})$.
(aproximarea poate adăuga stări iniţiale şi/sau tranziţii)

Fie modelul abstract aproximat $M_a = (S_r, \mathcal{A}(\mathcal{S}_0), \mathcal{A}(\mathcal{R}), L_r)$. Atunci $M \preceq M_a$ (modelul abstract aproximat îl simulează pe cel original)

Dacă funcţia de abstracţie păstrează relaţiile care corespund operaţiilor primitive din program, atunci $\mathcal{A}$ este o aproximaţie exactă. Formal:

O funcţie de abstracţie h_x defineşte o relaţie de echivalenţă între valorile concrete pentru x care corespund aceleiaşi valori abstracte:

$$d_1 \sim_x d_2 \Leftrightarrow h_x(d_1) = h_x(d_2)$$

Dacă valoarea oricărei relaţii primitive P din program este aceeaşi pentru orice perechi de valori concrete echivalente:

$$\forall d_1, \cdots d_n, d'_1, \cdots d'_n . \bigwedge_{i=1}^n d_i \sim_{x_i} d'_i \Rightarrow P(d_1, \cdots, d_n) = P(d'_1, \cdots, d'_n)$$

atunci $M \simeq M_a$ (modelul abstract e bisimilar celui original)

Interpretare abstractă

O metodă pentru definirea unei *semantici abstracte* a unui program, care poate fi utilizată pentru a analiza programul şi a produce informatii despre comportamentul său în execuţie. [Cousot & Cousot '77]

Constă în:
– un domeniu concret D şi un domeniu abstract A, legate printr-o *conexiune Galois*:
– o funcţie de abstracţie $\alpha : D \to A$
– o funcţie de concretizare $\gamma : A \to \mathcal{P}(D)$
(asociază fiecărei valori abstracte o mulţime de valori concrete)
– a.î. $\forall x \in \mathcal{P}(D) . x \subseteq \gamma(\alpha(x))$ şi $\forall a \in A . a = \alpha(\gamma(a))$
(abstractizarea urmată de concretizare introduce aproximare; concretizarea urmată de abstractizare e exactă)

Majoritatea abstracţiilor pot fi reformulate în acest cadru general.

Exemplu: Abstracții modulo un întreg

Pentru circuite/programe aritmetice: abstracția definită de

$$h(x) = x \text{ mod } n,\ n \in \mathbf{Z}$$

Păstrează relațiile primitive aritmetice, pentru că

$$((x \text{ mod } n) + (y \text{ mod } n)) \text{ mod } n = (x + y) \text{ mod } n, \text{ etc.}$$

În plus (lema chineză a resturilor): dacă $n_1, \cdots n_k$ relativ prime, și $n = n_1 \cdot n_2 \cdot \ldots \cdot n_k$, atunci

$$x \equiv y \ (\text{mod } n) \Leftrightarrow \bigwedge_{i=1}^{k} x \equiv y \ (\text{mod } n_i)$$

⇒ pentru a verifica un sistem cu aritmetică pe 16 biți, e suficientă verificarea pentru întregi modulo 5, 7, 9, 11, 32 (produs $> 2^{16}$)

Abstracții simbolice

Pentru verificarea căilor de date ale unui sistem
(funcția principală: calculul și transmiterea unor valori)

Exemplu: transmiterea corectă din a în b. Inițial: pt. valoare fixă:

$$\mathbf{AG}(a = 17 \rightarrow \mathbf{AX}\, b = 17)$$

Funcția de abstracție: $h(x) = \begin{cases} 1 & \text{dacă } x = 17 \\ 0 & \text{în caz contrar} \end{cases}$

Mai general: introducem parametrul simbolic c:

$h(x) = \begin{cases} 1 & \text{dacă } x = c \\ 0 & \text{în caz contrar} \end{cases}$

⇒ relație de tranziție abstractă $\hat{R}(\hat{a}, \hat{a}', \hat{b}, \hat{b}', c)$

Într-o reprezentare cu BDD-uri, c practic nu crește complexitatea dacă comportamentul sistemului e independent de c.

Exemplu: sumator cu pipeline pe două cicluri:

$$\mathbf{AG}(reg1 = a \wedge reg2 = b \rightarrow \mathbf{AX}\,\mathbf{AX}\, sum = a + b)$$

Model checking "on-the-fly"

Spațiul stărilor unui sistem = produsul componentelor: $S = S_1 \times \ldots \times S_n$
⇒ exponențial în numărul componentelor, adesea imposibil de construit

Dacă specificatiile sunt automate: pot "ghida" algoritmul de verificare, construind doar porțiunile necesare ale spațiului stărilor.

Se construiește doar automatul $\mathcal{S}$ din negația specificației

Din starea $s = (r, q)$ cu $r \in \mathcal{A}$ și $q \in \mathcal{S}$:
– se consideră doar acei succesori ai lui r cu tranziții etichetate la fel ca tranzițiile din q (din specificație)
– la găsirea unui (contra)exemplu, explorarea se încheie înainte de a fi explorat întreg spațiul stărilor

Metode cu ordonare parțială

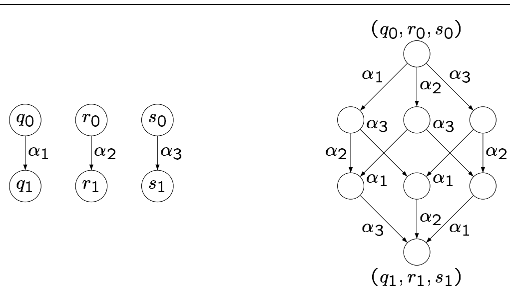
Ideea de bază: construirea unui model *redus*
= spațiul de stări și traiectoriile sunt submulțime ale celor originale

Justificarea: traiectoriile excluse nu aduc un plus de informație
– relație de *echivalență* între traiectorii
– specificația nu poate distinge între traiectorii echivalente
– modelul redus conține un reprezentant din fiecare clasă

Denumirea: inițial, bazată pe *ordonarea parțială* a tranzițiilor

Mai generic: *model checking cu reprezentanți*

O privire intuitivă



Compoziția asincronă ⇒ ordonare arbitrară a evenimentelor concurente
⇒ n tranziții generează $n!$ ordonări și 2^n stări
⇒ "explozie" combinatorială (exponențială) a spațiului stărilor

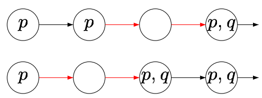
Tranziții. Dependență și independență

Modelul: sistem de stări-tranziții (S, T, S_0, L)
O tranziție $\alpha \in T$ e o *submulțime* $\alpha \subseteq S \times S$
(privită ca o familie de tranziții elementare etichetate la fel)
Tranziție *activată* în s: $\alpha \in enabled(s) \Leftrightarrow \exists s' \in S \,.\, \alpha(s, s')$
Considerăm doar tranziții *deterministe*: $\forall \alpha, s\ \exists! s' \,.\, \alpha(s, s')$
– sistemul însă poate fi nedeterminist, dacă $|enabled(s)| > 1$

Independență: două condiții, $\forall s \in S$:
Activare: $\alpha, \beta \in enabled(s) \Rightarrow \alpha \in enabled(\beta(s)) \wedge \beta \in enabled(\alpha(s))$
– două tranziții independente nu se pot *dezactiva* reciproc
– dar una o poate activa pe cealaltă
Comutativitate: $\alpha, \beta \in enabled(s) \Rightarrow \alpha(\beta(s)) = \beta(\alpha(s))$
– efectul execuției e același independent de ordine

Tranziții vizibile

Vizibilitate (în raport cu $AP' \subseteq AP$)
$\alpha \in T$ invizibilă $\Leftrightarrow \forall s, s' \in S, s' = \alpha(s) \Rightarrow L(s) \cap AP' = L(s') \cap AP'$
(nu schimbă etichetarea cu propoziții din AP')
în practică: AP' = propozițiile atomice din specificație



Invarianța la repetiție

În compoziția asincronă, operatorul **X** nu este relevant:
– două tranziții în componente diferite pot fi ordonate arbitrar
– două tranziții în aceeași componentă pot fi separate de tranziții în alte componente $\Rightarrow$ starea *locală* în componentă rămâne aceeași

Două traiectorii infinite $\pi = s_0 s_1 \ldots$ și $\pi' = r_0 r_1 \ldots$ sunt *echivalente la repetiție* (*stuttering equivalent*), $\pi \sim_{st} \pi'$ dacă pot fi divizate in blocuri finite de stări identic etichetate:
$\exists$ șirurile infinite $0 = i_0 < i_1 < \ldots$ și $0 = j_0 < j_1 < \ldots$, a.î. $\forall k \geq 0$
$L(s_{i_k}) = L(s_{i_k+1}) = \ldots L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = \ldots L(r_{j_{k+1}-1})$

O formulă LTL **A**f este *invariantă la repetiție* (*stuttering invariant*) dacă $\forall \pi, \pi'$ cu $\pi \sim_{st} \pi'$, $\pi \models f \Leftrightarrow \pi' \models f$

Teoremă: Orice formulă în LTL$_{-X}$ (fără operatorul **X**) este o proprietate invariantă la repetiție, si reciproc.

Principiul reducerii

Modelul redus e construit selectând din fiecare stare doar o *submulțime* de tranziții din cele activate în acea stare.

Selecția se face păstrând pentru fiecare cale din modelul original M o cale repetitiv echivalentă în modelul redus M'
$\Rightarrow \forall \mathbf{A}f \in LTL_{-X} \quad M \models \mathbf{A}f \Leftrightarrow M' \models \mathbf{A}f$

Diverse criterii de selecție și denumiri: *stubborn sets* [Valmari], *persistent sets* [Godefroid]; utilizăm *ample sets* [Peled].

Selecția tranzițiilor: descrisă printr-un set de condiții:

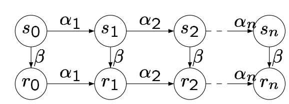
C0: $ample(s) = \emptyset \Leftrightarrow enabled(s) = \emptyset$
Succesor în modelul original $\Rightarrow$ există un succesor în modelul redus.

Condiții de reducere

C1 O traiectorie din s nu poate executa o tranziție dependentă de o tranziție din $ample(s)$ înainte de a fi executat o tranziție din $ample(s)$.

Proprietate: Tranzițiile din $ample(s)$ sunt independente de cele din $enabled(s) \setminus ample(s)$

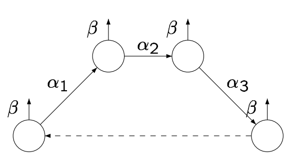
$\Rightarrow$ orice traiectorie dintr-o stare s are una din formele:
– un prefix $\alpha_1 \alpha_2 \ldots \alpha_n \beta$, unde $\beta \in ample(s)$, și α_i independente de β
– un șir infinit $\alpha_0 \alpha_1 \ldots$, cu α_i independente de orice $\beta \in ample(s)$



C2 (Invizibilitate) $ample(s) \neq enabled(s) \Rightarrow ample(s) \subseteq invisible(s)$
Dacă s nu e explorat complet, tranzițiile din $ample(s)$ sunt invizibile.

Condiții de reducere (cont.)

C3 O tranziție activată în toate stările unui ciclu trebuie inclusă în $ample(s)$ pentru o stare s din ciclu.



– garantează ca nu sunt ignorate porțiuni din spațiul stărilor datorită ignorării persistente a unei tranziții
– implementare: în orice ciclu, o stare e explorată complet

Construirea unei traiectorii echivalente

Pt. traiectoria π din s, construim una echivalentă π' în modelul redus:
a) dacă următoarea tranziție e în $ample(s)$, o adăugăm la π'
b) următoarea tranziție din π nu e în $ample(s)$
$\Rightarrow$ cf. **C2** tranzițiile din $ample(s)$ sunt invizibile ($\exists$ tranziții $\notin ample(s)$)
b1) dacă în π mai urmează o tranziție $\beta \in ample(s)$, se adaugă la π'
– cf. **C1**, β e independentă de tranzițiile precedente
– e invizibilă, deci comutareea nu afectează specificația
b2) nu există tranziții din $ample(s)$ în π
$\Rightarrow$ se adaugă tranziție arbitrară $\beta \in ample(s)$ la π'
– cf. **C1** nu dezactivează tranzițiile următoare
– e invizibilă $\Rightarrow$ nu afectează specificația
– cf. **C3** acest caz apare în număr finit

Selecția tranzițiilor în practică

Condițiile nu se pot verifica direct ⇒ euristici conservatoare

– Tranziții care citesc și scriu o variabilă comună sunt dependente
– Alegeri condiționale în același proces sunt dependente
– Tranzițiile de comunicație intră în dependențele ambelor procese
– Operațiuni de transmitere pe același buffer sunt dependente.
La fel, pt. operațiuni de recepție.

Tranziții cu mulțimi disjuncte de procese sunt independente.

⇒ selecteză o mulțime P de procese care în starea curentă nu au operații de comunicație cu procese din afara lui P.
⇒ $ample(s)$ = tranzițiile activate din P

Ideal: puține tranziții în $ample(s)$ (ex. tranziție locală într-un proces)

Reducerea *statică* cu ordonare parțială

Implementarea directă a condițiilor: reprezentare explicită a stărilor.
– **C3** e cel mai ușor de verificat într-o explorare DFS.

Obs: **C2** și **C3** limitează potențialul de reducere ⇒ pot fi combinate:
C2': Dacă $ample(s) \cap T^* \neq \emptyset$, s e explorat complet, unde T^*:
– conține toate tranzițiile vizibile
– orice ciclu din modelul redus conține o tranziție din T^*

Determinarea unei mulțimi T^* se poate face *static*
⇒ reducerea $enabled \to ample$ inclusă în model (precompilare)
⇒ nu necesită modificarea algoritmilor de verificare
⇒ combinație: reducere cu ordonare parțială + explorare simbolică
⇒ eficientă pentru sisteme mixte hardware/software