

Verificarea programelor

30 noiembrie 2004

- Axiomele lui Hoare și raționamente Floyd-Hoare
- Operatorul lui Dijkstra (weakest precondition)
 - Verificarea prin abstracție cu predicate

Începuturile verificării programelor

- verificarea formală a avut primele succese practice pentru hardware
- dar începuturile: din formalizarea semanticii limbajelor de programare

Robert W. Floyd. *Assigning Meanings to Programs* (1967)

"an adequate basis for formal definitions of the meanings of programs [...] in such a way that a rigorous standard is established for proofs"

"If the initial values of the program variables satisfy the relation R_1 , the final values on completion will satisfy the relation R_2 ."

- metoda: anotarea unui program (schemă logică) cu aserțiuni
- introduce noțiunile de *verification condition*: o formuiă $V_c(P; Q)$ a.î. dacă P e adevărat înainte de a executa c , atunci Q e adevărat la ieșire. și *strongest verifiable consequent* pt. un program + condiție inițială
- foarte general: aserții exprimate în logica predicatelor de ordinul I
- dezvoltă reguli generale pt. combinarea condițiilor de verificare și reguli specifice pentru diferitele tipuri de instrucțiuni
- introduce explicit *invariantii* pentru raționamentele despre cicluri
- tratează *terminarea* cu ajutorul unei măsuri pozitive descrescătoare

Lucrările lui Hoare

C.A.R. Hoare. *An Axiomatic Basis for Computer Programming* (1969)

– ca și Floyd, tratează precondiții și postcondiții pentru execuția unei instrucțiuni, dar notația de *triplet Hoare* pune mai clar în evidență relația dintre instrucțiune și cele două aserțiuni

– lucrează cu programe textuale, nu scheme logice

– Notăție: *corectitudine parțială* $\{P\} S \{Q\}$

Dacă S e executat într-o stare care satisface P , și execuția lui S se termină, starea rezultantă satisface Q

– Ulterior: raționamente similare pt. *corectitudine totală* $[P] S [Q]$

Dacă S e executat într-o stare care satisface P , atunci execuția lui S se termină, și starea rezultantă satisface Q

Aplicație riguroasă: C.A.R. Hoare. *Proof of a Program: FIND* (1971)

Axiomele (regulile) lui Hoare

– sunt definite pentru fiecare tip de instrucțiune în parte;
prin combinația lor, se poate raționa despre programe întregi

Atribuire:
$$\frac{}{\{Q[x/E]\} x := E \{Q\}}$$
 unde $Q[x/E]$ e substituția lui x cu E în Q

Ex: $\{x = y - 2\} x := x + 2 \{x = y\}$ (în rezultat, $x = y$, substituim x cu expresia atribuită, $x + 2$ și obținem $x + 2 = y$, deci $x = y - 2$)

Obs: scrierea “înapoi” a regulii (P în funcție de Q) simplifică exprimarea

Secvențiere:
$$\frac{\{P\} S_1 \{Q\} \quad \{Q\} S_2 \{R\}}{\{P\} S_1; S_2 \{R\}}$$

Decizie:
$$\frac{\{P \wedge E\} S_1 \{Q\} \quad \{P \wedge \neg E\} S_2 \{Q\}}{\{P\} \text{if } E \text{ then } S_1 \text{ else } S_2 \{Q\}}$$

Regulile lui Hoare (cont.)

Ciclul cu test (inițial): cheia în raționamentul despre programe

- trebuie găsit un *invariant* I = o proprietate menținută adevărată de fiecare execuție a ciclului (exprimată în punctul dintre iterații)
- dacă intrăm în ciclu (E), invariantul e menținut după o iterație S
- dacă nu mai intrăm ($\neg E$), invariantul implică concluzia Q

$$\frac{\{I \wedge E\} S \{I\} \quad I \wedge \neg E \Rightarrow Q}{\{I\} \text{ while } E \text{ do } S \{Q\}}$$

```
while (lo < hi) { /* căutare binară; I: lo <= n && n <= hi */
  m = (lo + hi) / 2;
  if (n > m) /* ambele cazuri mențin lo<=n && n<=hi */
    lo = m+1; /* n > m => n >= m+1 => n >= lo */
  else hi = m; /* !(n < m) => n <= m => n <= hi */
} /* I rămâne adevărat */
n = lo; /* lo<=n && n<=hi && !(lo<hi) => lo==n && n==hi */
```

Regulile lui Hoare în prezența pointerilor

Să stabilim $\{P\} *x = 2 \{v + *x = 4\}$

Care este precondiția P ? Răspunsul corect: $v = 2 \vee x = \&v$.

Dar aplicarea regulii $(v + *x = 4)[*x/2]$ pierde cazul al doilea.

Trebuie să modelăm memoria. m = memorie, a = adresă, d = dată.

Fie funcțiile $rd(m, a)$ return d și $wr(m, a, d)$ return m'

Avem regula: $rd(wr(m, a_1, d), a_2) = \begin{cases} d & \text{dacă } a_2 = a_1 \\ rd(m, a_2) & \text{dacă } a_2 \neq a_1 \end{cases}$

Trebuie să deducem o proprietate a memoriei m din relația:

$$rd(wr(m, x, 2), \&v) + rd(wr(m, x, 2), x) = 4$$

$$rd(wr(m, x, 2), \&v) + 2 = 4$$

$$rd(wr(m, x, 2), \&v) = 2$$

$$x = \&v \wedge 2 = 2 \vee x \neq \&v \wedge rd(m, \&v) = 2$$

$$x = \&v \vee v = 2$$

Operatorul *weakest precondition* al lui Dijkstra

E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs (1975)

- pentru o instrucțiune S și postcondiție dată Q pot exista mai multe precondiții P astfel încât $\{P\} S \{Q\}$ sau $[P] S [Q]$.
- Dijkstra calculează o precondiție *necesară și suficientă* $wp(S, Q)$ pentru terminarea cu succes a lui S cu postcondiția Q .
- necesară (*weakest*): dacă $[P] S [Q]$ atunci $P \Rightarrow wp(S, Q)$
- wp e un *transformator de predicate* (transformă post- în precondiție)
- permite definirea unui *calcul* cu astfel de transformări

Precondițiile lui Dijkstra (cont.)

Atribuire: $wp(x := E, Q) = Q[x/E]$ (v. regula lui Hoare)

Secvențiere: $wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$

Condițional:

$wp(\text{if } E \text{ then } S_1 \text{ else } S_2, Q) = (E \Rightarrow wp(S_1, Q)) \wedge (\neg E \Rightarrow wp(S_2, Q))$

Pentru iterație e nevoie de un calcul recursiv.

Definim wp_k , în ipoteza că bucla se termină în cel mult k iterații:

$wp_0(\text{while } E \text{ do } S, Q) = \neg E \Rightarrow Q$ (nu se intră în ciclu)

$wp_{k+1}(\text{while } E \text{ do } S, Q) = (E \Rightarrow wp(S, wp_k(\text{while } E \text{ do } S, Q))) \wedge (\neg E \Rightarrow Q)$

($\leq k + 1$ iterații \Leftrightarrow o iterație urmată de $\leq k$, sau 0 iterații; echivalent cu descompunerea primului while în if)

\Rightarrow se poate scrie ca formulă de punct fix

O problemă: invarianții pentru iterații

Cunoaștem P înainte de o buclă, dorim Q după execuție.
Cum stabilim un invariant I al buclei pentru a demonstra Q ?

I trebuie să satisfacă următoarele condiții:

- $P \Rightarrow I$ (I suficient de slab pt. a fi stabilit)
- $\{I \wedge E\} S \{I\}$ (I este un invariant)
- $I \wedge \neg E \Rightarrow Q$ (I suficient de puternic pt. a fi util)

Stabilirea invarianților e o problemă dificilă

Exemplu trivial:

$\{x \leq 0\}$ while $x < 9$ do $x \leftarrow x + 1$ $\{x < 10\}$

$x < 10$ e singurul invariant care poate fi stabilit cu succes
(sau $x < n$ cu $n \geq 10$, dar nu așa de util)

De regulă: calcul iterativ, ca punct fix.

necesită uneori întărirea invariantului (strengthening)

Abstracție cu predicate

Formalismele de tip Floyd-Hoare permit exprimarea de proprietăți ca *predicate* peste variabilele de stare ale programului.

- astfel am specificat de ex. proprietățile în verificatorul Spin.
- exemple de predicate: $x > 0$, $lock = 1$, $x + 1 < y$

Predicate abstraction [Graf & Saïdi '97]: metodă de construcție a unui *spațiu abstract* al stărilor date doar de valorile predicatelor definite.

Dorim: predicate suficient de bine alese pentru ca specificația să fie verificabilă peste spațiul abstract, fără explorarea spațiului concret.

Operații necesare pentru explorarea spațiului *abstract* al programului:

- *concretizare*: calculul stărilor concrete (din modelul inițial) reprezentate de predicatele din starea abstractă
 - calculul succesoriilor / predecesoriilor acestor stări (utilizând semantica *concretă* a instrucțiunilor programului)
 - *abstractizarea* stărilor concrete prin reintroducerea de predicate
- ⇒ E necesară o metodă (eventual aproximativă) pentru explorarea înainte / înapoi în spațiul abstract.

Interpretare abstractă

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

Constă în:

– un domeniu concret D și un domeniu abstract A , legate printr-o *conexiune Galois*:

– o funcție de abstracție $\alpha : D \rightarrow A$

– o funcție de concretizare $\gamma : A \rightarrow \mathcal{P}(D)$

(asociază fiecărei valori abstracte o mulțime de valori concrete)

– a.î. $\forall x \in \mathcal{P}(D) . x \subseteq \gamma(\alpha(x))$ și $\forall a \in A . a = \alpha(\gamma(a))$

(abstractizarea urmată de concretizare introduce aproximare;
concretizarea urmată de abstractizare e exactă)

Majoritatea abstracțiilor pot fi reformulate în acest cadru general.

Explorarea spațiului abstract: cadru general

- lucrăm *simbolic*, cu *mulțimi de stări* reprezentate prin formule
 $post(r, t) = \{s' \mid \exists s \in r . s \xrightarrow{t} s'\}$: *succesorul regiunii* (mulțimii de stări) r
- căutăm operatorul în spațiul abstract: $post^a(r, t) = \alpha(post^c(\gamma(r), t))$
- în general, acest calcul nu poate fi / e costisitor de efectuat precis (în speță operația de abstractizare α)
- ⇒ variante de abstracție cu predicate, în funcție de aproximația aleasă

Varianta 1: aproximare cu monoame [Graf-Saïdi]

- fiecare predicat e reprezentat în formă canonică disjunctivă (ca și disjuncție de *monoame* ϕ)
monom = conjuncție (produs) de predicate p_i sau negația lor $\neg p_i$
- succesorul $post^a(\psi, t)$ pt. tranziția (instrucțiunea) t e aproximat tot printr-un monom.
- \Rightarrow determinăm pentru fiecare predicat dacă monomul conține p_i sau $\neg p_i$ sau nicicare
- \Rightarrow determinăm pentru fiecare predicat p_i dacă $post^a(\psi, t)$ implică **sigur** p_i sau $\neg p_i$, adică, dacă $\psi \Rightarrow wp(p_i, t)$ sau $\psi \Rightarrow wp(\neg p_i, t)$

Varianta 2: descompunere completă, cu BDD-uri [Das-Dill-Park]

- Limitarea la aproximația succesorilor cu monoame e restrictivă
⇒ poate conduce la aproximări grosiere
- Pe de altă parte, un calcul mai precis ar putea duce la un număr exponențial de apeluri la proceduri de decizie ⇒ nefezabil
- regiunea ϕ e despărțită recursiv în fragmentele care pot conduce la stări ce satisfac p_i , respectiv $\neg p_i$.

$post^a(\phi, t) = post_1(\phi, t)$, unde

$post_k(\phi, t) = p_k \wedge post_{k+1}(\phi \wedge pre^c(\gamma(p_k), t), t) \vee \neg p_k \wedge post_{k+1}(\phi \wedge \neg pre^c(\gamma(p_k), t), t)$,

pentru $1 \leq k \leq n$, unde $pre^c(r, t) = \{s \mid \exists s' \in r . s \xrightarrow{t} s'\}$

și $post_{n+1}(\phi, t)$ a adevărat dacă ϕ e realizabil, și fals altfel.

Varianta 3: construirea unui program abstract [Ball-Rajamani]

- Variantele anterioare implică calculul *dinamic* al lui $post^a$ pentru *orice* combinație de predicate care apare în explorare
- în cazul cel mai defavorabil, acest număr este *exponențial*
- Soluție: separarea efectelor programului asupra fiecărui predicat
- ⇒ calculează *o singură dată* pentru fiecare instrucțiune care e efectul ei posibil asupra predicatului p_i
- ⇒ produce un program *boolean abstract* în care fiecare instrucțiune are ca efect atribuirea cu noi valori a fiecărui predicat
- adevărat, dacă starea dinainte implică $wp(\gamma(p_i), t)$
- fals, dacă starea dinainte implică $wp(\gamma(\neg p_i), t)$
- necunoscut, în caz contrar

Abstracția cu predicate în practica verificării software

Exemplu: Proiectul SLAM [Microsoft Research]
(Software (Specifications), Languages, Analysis and Model checking)

Scopul: verificarea unor proprietăți de siguranță (invarianti)
concret: un program respectă regulile de utilizare API
(ex: apelurile la `lock()` și `unlock()` alternează

- concentrat mai ales pe descoperirea erorilor de *interfață*
- utilizat practic pentru device drivers în Windows NT/XP

Caracteristici:

- nu necesită anotarea programului de către utilizator
(doar specificarea unor reguli sub formă de automat - monitor)
- abstracția e rafinată automat, ghidată de contraexemplele găsite

Abstracția cu rafinare ghidată contraexemple

- modelul abstract e graful de flux de control al programului augmentat cu un set ales de predicate boolene peste variabilele de program (la limită, setul inițial de predicate poate fi vid)
 - acest model finit e analizat prin algoritmi de model checking pentru a găsi o eventuală violare a specificației
 - dacă modelul e corect, programul e corect (abstracție conservatoare)
 - dacă se găsește un contraexemplu, acesta e explorat (parcurs) simbolic în programul concret, reținând corelațiile între variabile
 - dacă contraexemplul e fezabil, s-a găsit o eroare în program
 - contraexemplul poate fi și nefezabil, dacă conjuncția condițiilor necesare pentru a parcurge ramurile alese e nerealizabilă (falsă)
 - ⇒ contraexemplul e datorat abstracției prea grosiere
 - în acest caz, procedura de decizie care a demonstrat nerealizabilitatea sugerează și ce predicate sunt necesare pentru rafinarea abstracției
 - acestea sunt augmentate la modelul anterior și se repetă procedeul
- Aceste e doar un *semialgorithm* și succesul/terminarea nu e garantată

Exemplu de program

```
do {      /* Fragment de device driver, [Ball & Rajamani '01] */
    KeAcquireSpinLock(&devExt->writeListLock);
    nPacketsOld = nPackets;
    request = devExt->WriteListHeadVa;
    if(request && request->status) {
        devExt->WriteListHeadVa = request->Next;
        KeReleaseSpinLock(&devExt->writeListLock);
        irp = request->irp;
        if (request->status > 0) {
            irp->IoStatus.Status = STATUS_SUCCESS;
            irp->IoStatus.Information = request->Status;
        } else {
            irp->IoStatus.Status = STATUS_UNSUCCESSFUL;
            irp->IoStatus.Information = request->Status;
        }
        SmartDevFreeBlock(request);
        IoCompleteRequest(irp, IO_NO_INCREMENT);
        nPackets++;
    }
} while (nPackets != nPacketsOld);
KeReleaseSpinLock(&devExt->writeListLock);
```

Specificarea proprietăților

```
state {  
    enum { Unlocked=0, Locked=1 }  
    state = Unlocked;  
}
```

```
KeAcquireSpinLock.return {  
    if (state == Locked) abort;  
    else state = Locked;  
}
```

```
KeReleaseSpinLock.return {  
    if (state == Unlocked) abort;  
    else state = Unlocked;  
}
```

Specificarea este tradusă în C și programul original este instrumentat (programul original corect \Leftrightarrow programul instrumentat nu atinge eroare)

Generarea programului boolean

Pornește de la predicatele din specificație
nondeterminism în control; skip pentru instrucțiuni irelevante;

```
do {  
A: KeAcquireSpinLock_return();  
  skip;  
  if(*) {  
B:  KeReleaseSpinLock_return();  
    if (*) {  
      skip;  
    } else {  
      skip;  
    }  
  }  
} while (*);  
C: KeReleaseSpinLock_return();
```

Analiza programului boolean (model checking)

Bebop: calculează mulțimea stărilor atinse pentru fiecare instrucțiune dintr-un program boolean, folosind un algoritm de analiză a fluxului de date (dataflow analysis) interprocedural.

stare = atribuire pentru variabilele din domeniul de vizibilitate

mulțime de stări = funcție booleană, reprezentată prin BDD

calculul cu mulțimi de stări: capturează corelările între variabile

– nu expandează procedurile, exploatează localitatea variabilelor

– folosește un graf de control explicit

– complexitate: liniară în graful de control, exponențială în nr. de variabile vizibile într-un punct de program

Pentru exemplul dat: se semnalează eroare, se poate parcurge

A: KeAcquireSpinLock() de două ori succesiv

Contraexemplul și generarea noilor predicate

Se folosește un demonstrator de teoreme și un generator de condiții de verificare pentru a stabili dacă contraexemplul în programul abstract reprezintă un contraexemplu în programul concret.

Evaluează instrucțiunile programului folosind constante simbolice, până când demonstratorul de teoreme determină fie că atribuirea la sfârșitul traiectoriei este fezabilă, fie găsește o inconsistență pe parcurs.

Dacă se găsește o inconsistență, se caută una minimală și se generează predicatele corespunzătoare.

In exemplu: `nPacketsOld = nPackets` și `nPacketsOld != nPackets`
procedurile de decizie sunt incomplete \Rightarrow poate returna “nu știu”

(Re)generarea programului boolean

pt. fiecare instrucțiune, se determină dacă poate afecta vreun predicat
analiză modulară pt. proceduri, analiză de aliasuri pt. pointeri, etc.

Al doilea program boolean

```
do {
A: KeAcquireSpinLock_return();
  b = T;      /* b == (nPackets == nPacketsOld) */
  if(*) {
B:  KeReleaseSpinLock_return();
    if (*) {
      skip;
    } else {
      skip;
    }
    b := choose(F, b);      /* choose(p1, p2) == p1 ? T : p2 ? F : nondet */
  }
} while (!b);
C: KeReleaseSpinLock_return();
```

Abstracția obținută este suficientă pentru a demonstra corectitudinea.

Aplicații practice și optimizări

- În prezent: se pot analiza programe de cca. 10kloc și câteva sute de variabile boolene în câteva (zeci de) minute
- se estimează că prin optimizări se poate ajunge la cca 100kloc

Verificatoare pentru C, disponibile: BLAST (UC Berkeley), MAGIC (CMU)

Optimizare: *lazy abstraction* [Henzinger, Jhala, Majumdar, Sutre, Berkeley'02]

- nu se recrează abstracția la fiecare iterație
- abstracția curentă este rafinată cu predicatele noi
- ⇒ rafinată doar în fragmentele unde este necesar (on-the-fly)
- ⇒ se păstrează localitatea (ex. abstracții diferite pentru diferitele ramuri de control)