Verificarea programelor prin model-checking

30 octombrie 2014

Scopul verificării

demonstrarea că programul e *corect*

(dacă metoda și programul permit)

găsirea de *erori*

metode dedicate doar găsirii de erori

sau încearcă demonstrarea corectitudinii și indică erori la eșec

Metode de verificare

statică = fără execuția codului

  detecție de tipare

  analiză de flux de date

  verificare formală

dinamică = cu execuția codului

  instrumentare / rulare pe mașină virtuală

  execuție simbolică

Ce încredere am în rezultat ?

Metoda e:
consistentă ? (engl. *sound*) = orice răspuns e bun ?
completă ? = găsește toate răspunsurile ?

Verificare:
consistentă: un sistem raportat corect e corect
completă: poate determina corectitudinea oricărui sistem

Detecție de erori:
consistentă: o eroare raportată e reală
completă: găsește toate erorile

Verificarea formală

Sistemul e modelat matematic
$\Rightarrow$ sunt posibile rezultate *garantate* (certificate)
în limitele posibilităților/presupunerilor de modelare

Demonstrare de teoreme
condiții de verificare (din reguli Hoare)
demonstrații (sau algoritmi de realizabilitate a formulelor) (SAT-solvers)
probleme: necesar de anotații pentru formule complexe
interacțiune intensă cu expertul uman

Model checking
sistem = automat cu stări finite
algoritm = explorarea spațiului stărilor (traversare de grafuri)
automat; dă contraexemplu în caz de eroare
problema: explozia spațiului stărilor

Model checking pe scurt

dezvoltat din 1981 (Clarke & Emerson; Sifakis – premiul Turing 2007)
inițial aplicat la hardware și mici programe concurente

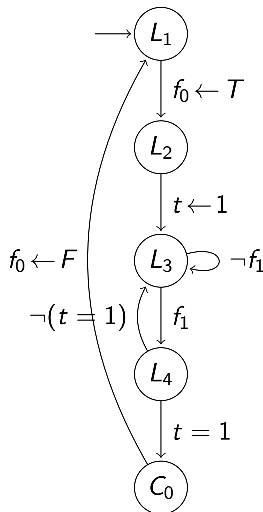
Exemplu: algoritmul lui Peterson de excluziune mutuală

```
while (1) {
  L1: flag[0] = true; // try
  L2: turn = 1; // other's turn
  L3: while (flag[1] && turn==1)
    ; // wait
  C0: flag[0] = false;
}
```

```
while (1) {
  R1: flag[1] = true; //try
  R2: turn = 0; // other's turn
  R3: while (flag[0] && turn==0)
    ; // wait
  C1: flag[1] = false;
}
```

Se poate ca programele să ajungă simultan în secțiunea critică ?
la etichetele C0 și C1, *înainte* de setarea pe *false* (eliberare)

Model checking: reprezentarea ca automat



Spațiul stărilor:
  variabilele: 3 biți: f_0, f_1, t, inițial $(?, ?, ?)$
  registrul de program pc (aici 2x fire)
  $\Rightarrow$ produs cartezian: perechi (pc_0, pc_1)
Reprezentând explicit: $2^3 \cdot 5 \cdot 5$ stări

Nu toate sunt *stări fezabile*.

Sunt fezabile stări cu $pc_0 = C_0$, $pc_1 = C_1$?

Răspuns: explorând spațiul stărilor
  înainte, din stările inițiale $(L_1, L_1, ?, ?, ?)$
    se ajunge într-o stare nedorită? sau
  înapoi, din stările nedorite $(C_0, C_1, ?, ?, ?)$
    se ajunge într-o stare inițială?

Un *model checker* implementează algoritmi de parcurgere; răspunde și la întrebări mai complexe, în *logică temporală*

Model checking software in practică

Proiectul SLAM [Microsoft Research] (începând din 2000)
(Software (Specifications), Languages, Analysis and Model checking)
ulterior, multe altele: BLAST (UC Berkeley), CBMC (Oxford), ...

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

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

Caracteristici:
– nu necesită anotarea programului de către utilizator
(doar specificarea unor reguli sub formă de automat - monitor)
– verificarea e făcută automat, pentru *toate* execuțiile posibile;
– se generează un *contraexemplu* (execuție concretă) în caz de eroare

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);
```

Doar codul colorat e de fapt relevant pentru alternanța corectă!

Specificarea proprietăților

Un lacăt poate fi reprezentat printr-un bit; `acquire` și `release` schimbă valoarea bitului sau dau eroare.

```
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)

Abstracția e esențială în verificare

Programele pot fi complexe.
Multe instrucțiuni pot fi irelevante pentru proprietatea studiată.

$\Rightarrow$ dorim să ne concentrăm pe fragmentul relevant din program

Tehnică: *program slicing*
determină fragmentul (*slice*) de program care afectează o anumită proprietate (*slicing criterion*)
(ex. valoarea unei variabile într-un punct)

Noțiune mai generală: *abstracția*
generarea unui model (program) simplificat, din a cărui analiză deducem proprietăți ale programului original

predicat = condiție booleană (expresie din variabilele programului)

Generarea programului boolean

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

Inițial, păstrăm doar *structura de control*, fără alte date:

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

Analiza programului boolean (model checking)

Se calculează mulțimea stărilor atinse:

stare = atribuire pentru variabilele din domeniul de vizibilitate
mulțime de stări = reprezentată eficient ca funcție booleană
(diagrame de decizie binare, BDD)
calculul cu mulțimi de stări: capturează corelările între variabile
relație de tranziție: tot o formulă booleană

$$state = 0 \wedge state' = 1$$

Pentru exemplul dat: se semnalează eroare, se poate parcurge
`A: KeAcquireSpinLock()` de două ori succesiv
dacă nu se intră în `if` care contine `B: Release...`

Eroarea descoperită e reală?

Execuția eronată (contraexemplul) în programul abstract (model)
e fezabilă în programul original (concret)?

Se încearcă execuția programului original pe calea de eroare găsită
= găsirea de valori de intrare, ex. prin *execuție simbolică*
(rezolvarea constrângerilor între valori date de calea aleasă)

Dacă contraexemplul e fezabil, reprezintă o eroare reală.

Dacă contraexemplul nu e fezabil, abstracția a fost prea grosieră
modelul trebuie rafinat și reanalizat
Tehnica: *counterexample-guided abstraction refinement*

Counterexample-guided abstraction refinement

În exemplul dat, reproducerea contraexemplului eșuează:
programul iese din ciclul `while` după prima iterație
$\Rightarrow$ condiția respectivă e *relevantă* pentru proprietatea analizată

Se introduce un nou *predicat* (variabilă booleană) reprezentând condiția:

$$\texttt{b} \stackrel{\text{def}}{:=} \texttt{nPackets != nPacketsOld}$$

Generăm un nou program boolean $\Rightarrow$ găsim instrucțiunile legate de `b`.
Atribuirile `nPacketsOld = nPackets` și `nPackets++` *afectează* `b`

Determinăm în ce condiții, după o atribuire știm sigur valoarea lui `b` (adevărat sau fals)
în funcție de toți biții de stare (2^n posibilități pentru n predicate, aici 1)

Abstractizarea instrucțiunilor

Găsim cea mai slabă precondiție pentru `b`, resp. `!b` după atribuirea dată.
Abreviem în continuare `nP` și `nP0`.

Aflăm *wp* pentru `b`: $wp_T = wp(\mathtt{nP} \leftarrow \mathtt{nP+1},\ \mathtt{nP{=}nP0}) = \mathtt{nP+1{=}nP0}$
Verificăm dacă $\mathtt{b} \rightarrow wp_T$ și dacă $\mathtt{!b} \rightarrow wp_T$
$\mathtt{nP{=}nP0} \not\rightarrow \mathtt{nP+1{=}nP0}$ și $\mathtt{nP{\neq}nP0} \not\rightarrow \mathtt{nP+1{=}nP0}$
Deci oricum ar fi `b` nu putem fi siguri că după `nP++`, `b` va fi adevărat.

Repetăm cu $wp_F = wp(\mathtt{nP} \leftarrow \mathtt{nP+1},\ \mathtt{nP{\neq}nP0}) = \mathtt{nP+1{\neq}nP0}$
Avem $\mathtt{nP{=}nP0} \rightarrow \mathtt{nP+1{\neq}nP0}$ și $\mathtt{nP{\neq}nP0} \not\rightarrow \mathtt{nP+1{\neq}nP0}$

Deci dacă `b` știm sigur că după `nP++` avem `!b`, altfel nu știm nimic.
$\Rightarrow$ putem abstractiza `nP++` cu `b = b ? F : nondet`

La fel, putem abstractiza `nP0 = nP` cu `b = T`

Se regenerează programul boolean cu noile predicate și se reia verificarea.

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();
```

În concluzie ...

În acest caz, noua abstracție obținută e suficientă.
Explorând toate stările pentru programul boolean dat, *model-checker*ul nu mai găsește o cale spre starea de eroare.

dacă se trece prin `B:Release`, `b` devine `F`, se rămâne în ciclu, nu se poate executa succesiv `C:Release` (se face `A:Acquire`)

dacă nu se trece prin `B:Release`, `b` rămâne `T`, se iese din ciclu, nu se poate repeta `A:Acquire` (se face `C:Release`)

Pot fi necesari mai mulți pași de abstracție, nu e garantată terminarea.

În practică, *model checking* e fezabil pentru programe în care predomină *controlul*; găsește curent erori în drivere, nucleu Linux, etc.