

Verificarea programelor în practică

18 ianuarie 2005

- Verificarea programelor Java (Pathfinder, ESC/Java, Bandera)
- Proof Carrying Code
- Combinații cu analiză statică

Java PathFinder (NASA Ames)

Unul din primele eforturi de a verifica cod în limbaje de programare uzuale

Inițial, (Java PathFinder 1.0): traducere din Java în PROMELA (Spin)

– similarități de limbaj: tratarea de creare dinamică de obiecte, thread-uri

– dar: lipsesc alte facilități (numere reale); necesită sursa completă

Java PathFinder 2.0: verificator de sine stătător, scris în Java

[G. Brat, K. Havelund, S. Park, W. Visser '00]

Arhitectura generală

Model checking cu reprezentare explicită a stărilor

– tehnică uzuală pentru reprezentarea stărilor de dim. mari (structuri): fiecare val. structurală stocată doar o dată; înlocuită cu un cod întreg

Mașină virtuală Java proprie pentru model checking

– + algoritmi de explorare (comandă pași înainte/înapoi în JVM proprie)

– mediu nedeterminist: prin metode speciale captate de JVM

Java PathFinder (cont.)

Tehnici folosite (cont.)

Analiză statică

- pentru construirea abstracțiilor din program (prin *slicing*)
- pentru identificarea condițiilor de reducere cu ordonare parțială folosind un demonstrator de teoreme, SVC (Stanford Validity Checker)

Analiză la rulare, pentru detectarea potențialelor condiții de eroare

- accesul concurent neprotejat la variabile comune
- accesarea în ordine diferită a semafoarelor

Performanță; sisteme verificate

- scris în Java, de 10x mai lent ca Spin; viteză: $n \cdot 1000$ stări/secundă
- un agent de control pentru operarea în spațiu
- un fragment de sistem de operare distribuit (14 clase, 1 kloc)

Abstracția în Java Pathfinder

Prin transformare la nivelul sursă, rezultă alt program Java care operează cu predicatele abstracte.

```
Abstracția e specificată ca și anotație specială: class Abstract;  
Abstract.remove(x) // abstractizează x  
Abstract.addBoolean("x0", x == 0) // adaugă predicatul x == 0
```

Se pot abstractiza și predicate peste mai multe clase:

```
Abstract.addBoolean("xGTy", A.x > B.y);
```

- generează un predicat pentru fiecare pereche de obiecte instanțiate din clasele A și B
- posibilă explozie a numărului de predicate

ESC/Java (DEC/Compaq SRC)

ESC = Extended Static Checking; inițial pentru Modula 3, apoi Java

- nu este un model checker, ci folosește verificări prin *analiză statică*
- poate detecta erori de genul: referințe nule, depășiri de indici
- pentru proprietăți mai complexe, se folosesc *adnotări*
(invarianți, precondiții, postcondiții, condiții de nul/nenul, etc.)
- pentru verificarea se folosește un demonstrator de teoreme (Simplify)
- permite verificarea modulară, separat pentru fiecare metodă
- modulele cu sursă indisponibilă: suplinite prin fișiere de specificații

Analizoare statice similare există și pentru C (fost `lint`, acum `splint`)

Bandera [Kansas State U.]

Un verificator modular pentru programe Java

- un front-end pentru simplificarea programului (program slicing)
- o bibliotecă de abstracții frecvent folosite
- eventual restricția modelului la număr mic de instanțe de module (utilizatorul specifică pt. fiecare variabilă abstracția dorită)
- un generator pentru un model finit într-un format generic (limbaj cu guarded commands, tradus ușor în alte limbaje de specificare)
- interfețe cu verificatoare uzuale (SMV, Spin, demonstratorul PVS)
- un limbaj de specificare, și suport pt. formule pe bază de tipare

Proof Carrying Code

[Necula & Lee '96, Necula '97]

- O metodă pentru execuția sigură a codului lipsit de nivel cert de încredere (untrusted code). Exemplu: aplicație via internet
- Consumatorul de cod definește niște reguli de siguranță
- Furnizorul livrează codul însoțit de o demonstrație formală că satisface regulile de siguranță
- Consumatorul folosește un verificator simplu de demonstrații pentru a stabili validitatea codului și demonstrației primite

PCC: Structura sistemului

Producătorul

- Compilator cu certificarea codului (certifying compiler) generează cod nativ cu anotații (ex. invarianti)
 - Generator de condiții de verificare (VCGen)
- VC = predicat a cărui validitate garantează execuția sigură
- Generator de demonstrații (pornind de la VC)

Consumatorul

- Un verficator de demonstrații: validează corespondența între codul și demonstrația primită
(eventual: asistat de un VCGen identic cu cel anterior)

PCC: Generarea condițiilor de verificare

- Noțiunea de VC: legată de regulile pentru corectitudinea programelor, bazate pe precondiții și postcondiții [Floyd '67]
- VC nu este neapărat weakest precondition (poate fi mai simplă, mai ușor de exprimat și demonstrat)

```
for (i = 0; i < length(a); i++) s += a[i];
```

VC construită din predicate de tipul:

$s : int \wedge l \geq 0$	premisă
$i \geq l \rightarrow s : int$	postcondiție
$i < l \rightarrow \text{saferd}(mem, a + i) \wedge i + 1 \leq l \wedge s + rd(mem, a + i) : int$	invariant

PCC: Avantajele metodei

- Validarea demonstrației e mult mai simplă decât găsirea
⇒: simplu pt. consumator; încredere doar în verificator
- Procedeu rezilient la intervenții: orice modificare a codului sau a demonstrației e detectată (dacă nu, programul, chiar modificat, satisface condițiile de siguranță)
- Verificarea se face static, o singură dată: permite execuția eficientă fără introducerea de teste la rulare
- Permite siguranța absolută în compilator, și chiar depanarea acestuia în procesul de dezvoltare

CCured: Cod C puternic tipizat

- [Necula et al '01]: programe C corecte dpdv al tipurilor
- combinație de inferență a tipurilor și verificare la rulare
 - inferența tipurilor pt. a demonstra că o porțiune cât mai mare din cod este type-safe
 - inserarea de teste la execuție în restul programului pentru a asigura corectitudinea accesului la memorie
 - ideea de bază: extinderea sistemului de tipuri cu pointeri SAFE, SEQ (pt. tablouri) și DYNAMIC
 - introducerea de câmpuri suplimentare (bază și lungime) pentru pointerii care nu sunt SAFE
 - factor de încetinire: 1 - 2 (față de 10-100 pt. Purify)
 - analiza permite și descoperirea de erori

Extensii pentru metacompilare

Contextul: *lightweight and semi-formal methods*: sacrifică parte din garanții pentru a crește abordabilitatea și aplicabilitatea practică

Metacompilation [Engler et al '00]: pt. erori în sisteme de operare

– reguli la nivel înalt bine stabilite (ex. “variabila x protejată cu semaforul s ”, “întreruperile trebuie reactivate”)

⇒ definirea unei meta-semantici accesibile compilatorului

– regulile specificate ca automate care tranziționează la analiza sintactică a unui model (pattern) relevant din sursă

– un compilator C augmentat aplică aceste extensii la analiza semantică (propagare prin graful de control, local sau global)

– rezultate: sute de erori în Linux, OpenBSD, Exokernel, etc.

Proiect complementar: extragerea automată de modele din sursă, prin slicing (tot pentru sisteme de operare)

– modelele sunt analizate apoi cu un model checker