Metode formale. Introducere

5 oct. 2004

- Funcţionarea corectă a sistemelor. Importanţă şi dificultate.
- Ce sunt metodele formale ?
- Domenii şi exemple de aplicaţie
- Modelare şi specificare

Obiectivele cursului

– urmărirea *corectitudinii* sistemelor proiectate
– cunoașterea principalelor tipuri și surse de *erori*
– cunoașterea *metodelor formale* ca alternativă la simulare și testare
– obișnuința cu *rigoarea* în descrierea sistemelor
– construirea de *modele* corespunzătoare pentru sistemele proiectate
– exprimarea *neambiguă* a specificațiilor (proprietăților dorite)
– *evaluarea aplicabilității* metodelor formale (manuale sau automate)
– cunoașterea unor *utilitare* de verificare

Erori grave: Therac-25

- aparat medical pentru terapie cu radiaţie
- 6 accidente cu morţi şi răni grave (1985-87, SUA, Canada)
- cauza directă: erori in programul de control

Analiză retrospectivă [Leveson 1995]:

- încredere excesivă în software (în analiza produsului)
- fiabilitate $\neq$ siguranţă
- lipsa măsurilor de siguranţă hardware
- lipsa practicilor de ingineria programării (proiectare defensivă, specificare, documentaţie, simplitate, analiză formală, testare)
- corectarea unei erori nu face sistemul mai sigur !!

Erori: racheta Ariane 5

- Autodistrugere după o defecţiune la 40 s de la lansare (1996)
- Cauza: conversia 64-bit float $\rightarrow$ 16-bit int generează o excepţie de depăşire netratată în programul ADA
- Cost: 500 milioane dolari (racheta), 7 miliarde dolari (proiectul)

Analiză retrospectivă

- principala cauză: reutilizarea nejudicioasă de software
- cod preluat de la Ariane 4, fără reanalizare corespunzătoare:
  - execuţia nu mai era necesară in timpul erorii
  - analiza absenţei depăşirii pentru variabilele neprotejate

  $\Rightarrow$ necesitatea specificării şi respectării unei interfeţe
- proiectarea greşită a redundanţei: sistemul de referinţă inerţial şi cel de rezervă scoase din funcţiune de aceeaşi eroare

Eroarea din procesorul Pentium

Eroare în unitatea de împăţire cu virgulă mobilă, 1994

- algoritm de împărţire cu refacere, în baza 4
- determină următoarea cifră din cât dintr-un tabel
- câteva intrări marcate greşit ca "don't care"
- cost: cca. 500 milioane dolari

  Analiză ulterioară
- Circuitul putea fi verificat formal

  - demonstrare automată de teoreme [Clarke, German & Zhao]

  - cu structuri speciale de date pentru reprezentarea înmulţirii/împărţirii [Bryant & Chen]
- dar accentul a fost pus pe componente mai complexe (unitatea de execuţie, coerenţa memoriei cache)

Erori: probele trimise pe Marte

Mars Pathfinder, 1997

- ajunsă pe Marte, proba spațială se reseta frecvent
- cauza: inversiune de prioritate între procese cu resurse comune
- fenomenul și soluția: cunoscute în literatura de specialitate !

  [Sha, Rajkumar, Lehoczky. Priority Inheritance Protocols, 1990]

1. procesul A de prioritate mică obține resursa R
2. A întrerupt de C (prioritate mare)
3. C așteaptă eliberarea lui R; A revine în execuție
4. A întrerupt de B (prioritate medie, A $<$ B $<$ C)

$\Rightarrow$ C așteaptă după B, fără a fi direct condiționat de B !

Soluția: ridicarea priorității unui proces care obține o resursă (A) la nivelul celui mai prioritar proces care poate solicita resursa (C)

Erori: probele trimise pe Marte

Mars Climate Orbiter, 1998

- dezintegrare la intrarea în atmosferă
- eroarea tehnică: discrepanța între unități de măsură în sistemele anglo-american și metric
- erori multiple de proces: lipsa unor interfețe formale

Mars Polar Lander, 1998

- trenul de aterizare e activat prematur la intrarea în atmosferă
- șocul e interpretat ca aterizare, motoarele sunt oprite
- eroarea: lipsa testării de integrare

Cum se pot detecta erorile ?

Testare

\+ direct pe produs $\Rightarrow$ teste concludente

– erorile detectate târziu sunt costisitoare

– diagnosticul necesită observabilitate completă

Simulare

\+ efectuată deja în faza de proiectare

– simulatorul poate fi mai lent decat sistemul real

– Testarea sau simularea exhaustivă e adeseori imposibilă

Testarea poate demonstra prezența erorilor, dar nu absența lor.
(E. W. Dijkstra, 1979)

Ce sunt metodele formale?

“ ... limbaje, tehnici și unelte matematice pentru specificarea și verificarea sistemelor” [Clarke & Wing, 1996]

Sau, mai in detaliu: “un set de unelte și notații,
– cu o semantică formală,
– folosite pentru a specifica neambiguu cerințele unui sistem,
– care admite demonstrarea de proprietăți ale acelei specificații
– și demonstrarea corectitudinii unei implementări în raport cu acea specificație”
[Hinchey & Bowen, *Applications of Formal Methods*, 1995]

Ce pot garanta metodele formale?

- Nu există garanţii absolute
- O metodă formală nu poate fi mai bună decât modelul şi specificaţiile care sunt folosite
  – *modelul şi specificaţiile* trebuiesc validate

  Dar pot oferi:
- o procedură logic consistentă de raţionament
- o acoperire exhaustivă, adeseori imposibilă altfel
- mecanizare şi automatizare $\Rightarrow$ performanţă şi corectitudine
  Pot complementa cu succes simularea, testarea, etc.

Metode formale: Necesitate și dificultăți

Utilitate îndeosebi pentru:

– *complexitate*: tehnici de abstracție / aproximare

– *concurență*: foarte greu de reprodus și analizat altfel

– *criticalitate*: (avionică, bancar, medical, securitate)

Dinamica erorilor în software [după John Rushby, SRI]

- 20-50 erori/kloc înainte de testare, 2-4 după
- examinarea formală a codului poate reduce erorile înainte de testare de cca 10 ori

Studiu de caz pe 10kloc timp real, distribuit:

- verificare și validare: 52% cost (57% timp)
- din acesta, 27% cost în examinare, 73% în testare
- 21% pt. 4 defecte în testarea finala, din care 1 de proiectare
- eliminarea erorilor la examinări detaliate ale codului: de 160 de ori mai eficientă decât la testare

Cauzele și costul erorilor (cont.)

Erori în programe [după NASA JPL (sondele Voyager, Galileo)]

- majoritatea: deficienţe în specificarea cerinţelor și a interfeţelor
- 1 eroare la 3 pagini de cerinţe și 21 pagini de cod
- doar 3 din 197 erau erori de programare
- 2/3 din erorile funcţionale: omisiuni în specificarea cerinţelor
- majoritatea erorilor de interfaţă: datorate proastei comunicări

Erori în hardware [după Kurt Keutzer, UC Berkeley]

- $>$ 50% din proiecte au erori dupa prima fabricare
- erorile pe linie de cod hardware trebuie reduse în ritmul creșterii numărului de tranzistori pe chip
- practic zero erori la nivelele inferioare de implementare
- problemele sunt în proiectarea la nivel înalt
  (hazarduri în pipeline, execuţie nesecvenţială, procesoare superscalare, coerenţa memoriilor cache, protocoale complexe, etc.)

O privire de ansamblu

- Cauzele cele mai frecvente de erori: din concepţie, defecte simultane, interacţiuni neprevăzute
  – lipsurile principale: în aplicarea timpurie a metodelor formale
  – costul principal: eliminarea târzie a erorilor
- Potenţialul maxim al metodelor formale:
  – in modelarea şi verificarea la nivel înalt
  – pentru sisteme complexe, concurente, distribuite, reactive, în timp real, tolerante la eroare, etc.

Metodele formale în ciclul de producţie

- Analiza cerinţelor.
  – identifică contradicţii, ambiguităţi, omisiuni
- Proiectare
  – descompunerea în componente şi specificarea interfeţelor
  – proiectarea prin rafinare succesivă
- Verificare
- Testare şi depanare
  – generarea direcţionată de cazuri de test
- Analiză
  – model abstract, mai puţin complex decât sistemul real

Aplicaţii: Proiectare hardware

- Verificarea echivalenţei combinatoriale
  – este deja standard în utilitarele CAD
- Verificarea circuitelor secvenţiale
  – marile companii au colective speciale
  (IBM, Intel, Motorola, Fujitsu, Siemens, etc.)
  – folosesc verificatoarele disponibile public sau cele proprii
- protocoalele de coerenţă cache Gigamax şi Futurebus+
- Motorola 68020: modelat în demonstratorul Boyer-Moore,
  verificarea codul binar produs de compilatoare
- AAMP-5 (procesor pentru avionică): modelat în PVS,
  verificarea microcodului pentru instrucţiuni
- procesoare cu pipelining / superscalare tip DLX

Aplicaţii: Avionică

Lockheed C130J

– analiza codului ADA cu anotaţii în limbajul SPARK

– software "corect prin construcţie", cost redus

TCAS-II (Traffic Collision Avoidance System)

- instalat obligatoriu pe avioanele comerciale din S.U.A.
- alertă și deviere automată în cazul apropierii periculoase
- specificaţia redactată într-un limbaj formal (RSML)
- s-a verificat completitudinea și consistenţa [Heimdahl, Leveson '96]
- s-a abandonat încercarea de a specifica în engleză

  – Modele formale pentru sisteme complexe sunt fezabile

  – Pot fi analizate de experţii din domeniul de aplicaţie

Alte Aplicaţii

- Telefonie. Specificarea şi analiza interacţiilor între diversele caracteristici ale sistemului telefonic.
- Sisteme electronice de consum. Verificarea manuală, apoi automată a protocolului de control din componentele audio Philips,
- Sisteme de control în electronica auto.
- Protocoale de comunicaţie.
- Protocoale de securitate. Analiză folosind logici speciale pentru a raţiona despre mesaje criptate, intruşi, etc.
- Software de sistem. Verificarea driverelor de periferice.

Metode formale: Specificare

- Specificarea: necesară în orice metodă formală (poate fi unicul aspect)
- necesită limbaj cu sintaxă și semantică definită formal (matematic)
  Limbajul de specificare definește:
  – un domeniu sintactic (notația)
  – un domeniu semantic (universul de obiecte considerat)
  – o definiţie precisă a obiectelor care satisfac o specificaţie
  [M. Chechik, *Automated Verification*, curs, U. Toronto]

Sintaxă și semantică

Sintaxa

– un alfabet de simboluri (ex. propoziţii, operatori logici)
– reguli gramaticale pentru formarea unor formule bine definite

Semantica

Domeniul semantic variază de la limbaj al limbaj:
– secvenţe de stări, secvenţe de evenimente, structuri de sincronizare (în limbajele de specificare a sistemelor concurente)
– funcţii intrare $\rightarrow$ ieșire, relaţii, computaţii, transformatoare de predicate (în cazul limbajelor de programare)

Tipuri de specificaţii

- declarative (nu trebuie să reprezinte o funcţie calculabilă)
- executabile (ex. limbajele de programare)

- comportamentale (orientate pe proprietăţi) (ex. funcţionalitate, reactivitate)
  – descriu comportamentul sistemelor în raport cu proprietăţile ce trebuie satisfăcute
- structurale (orientate pe modele) (ex. diagrame, conexiuni, ierarhie)
  – construiesc un model al sistemului folosind noţiuni matematice precise
  (mulţimi, funcţii, logica predicatelor)

  Uneori: același limbaj pentru specificaţie și model (sau implementare)
  $\Rightarrow$ e posibilă rafinarea pe nivele succesive de abstracţie

Proprietăţi ale specificaţiilor

- neambiguă: un înţeles bine definit (NU: limbaj fără semantică formală, limbaj natural, scheme grafice cu mai multe interpretări)
- consistentă (necontradictorie)
  – există cel puţin un obiect care o satisface
- poate fi incompletă
  – comportament la latitudinea implementării sau nondeterminism

Dacă limbajul are un sistem de *inferenţă logică* se pot *demonstra* proprietăţi pornind de la o specificaţie.

Specificare: Limbajul Z

– bazat pe logica de ordinul I și teoria mulțimilor
– descriere funcțională, declarativă
– folosit extensiv la proiecte industriale în Marea Britanie

$PhoneDB$
$members : \mathbf{P}\, Person$
$telephones : Person \leftrightarrow Phone$

---

dom $phones \subseteq members$

$FindPhones$
$\Xi PhoneDB$
$name? : Person$
$numbers! : \mathbf{P}\, Phone$

$name? \in$ dom $phones$
$numbers = phones(|\{name?\}|)$

– o *schemă* (PhoneDB) (stări + evtl. tranziții), și un *invariant*
– operații care schimbă starea (Δ) sau nu (Ξ)

Specificare: Limbajul Larch

[Guttag, Hornig, Garlan, MIT/DEC SRC]

1. abstracţie (specificare) independentă de limbaj
2. specificare de interfaţă pentru module într-un anumit limbaj

```
Table: trait
  includes Integer
  introduces
    new: -> Tab
    add: Tab, Ind, Val -> Tab
    lookup: Tab, Ind -> Val
  asserts \forall i, i1: Ind, v: Val, t: Tab
    \not (i \in new);
    i \in add (t, i1, v) == i = i1 \/ i \in t
    lookup(add(t, i, v), i1) ==
        if i = i1 then v else lookup(t, i1)
```

Limbajul Larch (cont.)

Specificare de interfață pentru limbajul C:

```
mutable type table
uses Table(table for Tab, char for Ind,
           char for Val, int for Int);
constant int maxTabsize;
table table_create(void) {
  ensures result' = new /\ fresh(result);
}
char table_read(table t, char i)
  requires i \in t^;
  ensures result = lookup(t^, i);
}
```

– definește precondiții și postcondiții

– interfața rămâne la nivelul abstract (fără algoritmi)

Specificare: alte limbaje

VDM (Vienna Development Method

– originează din eforturile grupului IBM Viena în anii '70
– similară și înrudită cu Z

B

– dezvoltată de Jean-Raymond Abrial
– spre deosebire de Z, are și suport automatizat puternic
– precondiții / postcondiții, invarianți, rafinament
– suport pentru generarea automată de cod
– utilizare industrială (metroul din Paris, Alsthom, $n \cdot 10$kloc)

Noțiuni de specificare de interfață încorporate direct în limbaje, de ex. Eiffel (design by contract)

Modelarea sistemelor concurente

Două direcţii de abordare:
– programare imperativă + adăugiri
(semafoare, monitoare, rendezvous, etc.)
– model de calcul concurent, bazat pe interacţiunea proceselor
(“*interacţiune indivizibilă*”)

Comunicarea şi concurenţa sunt noţiuni complementare [Milner]

- Communicating Sequential Processes [Hoare]
- Calculus of Communicating Systems [Milner]

Modelare: Communicating Sequential Processes (CSP)

Exemplu [Hoare]: automat pentru ciocolată, cu monede

Alfabetul: $\alpha_V = \{in1p, in2p, small, large, out1p\}$
Comportamentul:
$V = (in2p \to (large \to V | small \to out1p \to V)$
$\quad | in1p \to small \to V)$
sau, formal:
$V = \mu X.(in2p \to (large \to X | small \to out1p \to X)$
$\quad | in1p \to small \to X)$
(unica soluție a ecuației de mai sus)

CSP: formalism (algebră a proceselor) axat pe acțiuni,
cu nondeterminism, compoziție sincronă, etc.

Modelare: Automate cu stări finite

- Variante:
  - etichete pe stări sau pe tranziţii
  - tranziţii specificate ca funcţii sau relaţii
  - augmentat sau nu cu variabile (date)
- Structură Kripke:
  = automat etichetat cu *propoziţii atomice* dintr-o mulţime AP:

$$M = (S, S_0, R, L)$$

  - S: mulţime finită de stări
  - S_0: mulţimea stărilor iniţiale
  - $R \subseteq S \times S$: relaţie de tranziţie totală
  - $L : S \to 2^{AP}$: funcţie de etichetare a stărilor

Noţiunea de corectitudine

- În general: sistemul satisface o proprietate (specificaţie)
- Comportament funcţional corect.
  - sistemul privit ca realizând o funcţie intrare $\longrightarrow$ ieşire
  - exemplu de formalism: tripletele lui Hoare

$$\{P\} \quad S \quad \{Q\}$$

{ precondiţie } program(sistem) { postcondiţie }

Exemplu de raţionament:

$$\frac{\{P\}\ S_1\ \{Q_1\} \qquad Q_1 \Rightarrow Q_2 \qquad \{Q_2\}\ S_2\ \{R\}}{\{P\} \quad S_1; S_2 \quad \{R\}}$$

Noţiunea de corectitudine (cont.)

Comportament temporal corect

- pentru sisteme *reactive*: execuţie (conceptual) infinită
- comportamentul definit ca reacţie la o secvenţa de intrare
- specificare: de ex. logică temporală
- proprietăţi: absenţa blocajului, reacţie în timp limitat, etc.

Exemple:

– orice cerere este urmată de un răspuns în cel mult 5 secunde

– orice proces obţine resursa de un număr infinit de ori

– pe orice traiectorie, se ajunge la un moment dat în starea stabilă

Procedee de verificare

Două mari categorii:

Explorarea spaţiului stărilor (model checking)

– specificarea de regulă în logică temporală

– algoritmi de explorare exhaustivă verifică valoarea de adevăr

sau produc o secvenţă de execuţie ca și contraexemplu

– verificarea echivalenţei: specificarea e la rândul ei un model

Demonstrarea de teoreme

– reprezentare într-un sistem logic cu axiome și reguli de deducţie

– domeniul analizat reprezentat și el printr-un grup de axiome și reguli (o teorie)

– demonstrare mecanizată: ghidată manual sau automată