Úvod do paralelného programovania

- Damas Gruska, I20,
  gruska@fmph.uniba.sk

  www.ii.fmph.uniba.sk/~gruska

Organizácia kurzu

Dva testy počas semestra: prvý v polovici, druhý v poslednom týždni semestra.

Testy budem opravovať až na skúške a budú tvoriť jej jadro.

Kto nebude písať test cez semester bude ho musieť písať v čase skúšky.

Známku "vychádzajúcu" z testu si možno opraviť na skúške.

Top 10 – November 2013

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3120000	33862.7	54902.4	17808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	17173.2	20132.7	7890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786432	8586.6	10066.3	3945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115984	6271.0	7788.9	2325
7	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462462	5168.1	8520.1	4510
8	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458752	5008.9	5872.0	2301
9	DOE/NNSA/LLNL United States	**Vulcan** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393216	4293.3	5033.2	1972
10	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147456	2897.0	3185.1	3423

Distribuované výpočty

Pomocou internetu spojené osobné počítače

SETI@Home - 769 TFLOPS

BOINC - 5.428 PFLOPS (1petaflops=1000 TFLOPS)

Berkeley Open Infrastructure for Network Computing (rôzne aplikácie matematika, medicína, klimatológia, astofyzika, ...)

Folding@Home - 5,6 PFLOPS

(simulácie, molekulárna biológia)

Cena

Cena za jednu GFLOPS

1961 - 1 100 000 000 000 USD- 17 miliónov počítačov IBM 1620 po 64 000$

1984 - 15 000 000 USD Cray X-MP

1997 - 30 000 USD dva 16-procesorové Beowulf klustre s Pentium Pro

2000, apríl - 1 000 USD Bunyip Beowulf kluster

2003, august - 82 USD KASY0

2013, december - 0.12 USD Intel Pentium G550 2.6GHz Dual-Core Processor

Úvod do paralelného programovania

- **Cieľ kurzu:**

  naučiť systémový návrh riešenia úloh paralelného programovania pre rôzne typy architektúr a pomocou logického kalkulu naučiť dokázať správnosť týchto riešení

- **Plán kurzu:**

  syntax jazyka UNITY, špecifikačná logika, typy architektúr

  riešenie konkrétnych úloh paralelného programovania z rôznych oblastí (operačné systémy, fault tolerant systémy, protokoly ...)

  iné paralelné jazyky, iné špecifikačné logiky

- **Literatúra:**

  K. M. Chandy, J. Misra: *Parallel Program Design*. Addison-Wesley 1988

  C. Stirling: Modal and Temporal Properties of Processes, Springer 2001

Obsah

- Úvod
- UNITY – syntax, sémantika, logika
- Architektúry a jednoduché príklady
- Najkratšia cesta
- Readers–Writers problém
- Večerajúci filozofi
- Koordinácia schôdzí
- Pijúci filozofi
- Triedenie
- Faulty channels
- Global snapshots
- Detekovanie stabilných vlastností
- Byzantská dohoda
- Temporálne logiky

Úvod

- **História:**
  - 50-te roky: sekvenčné programy boli šité na mieru pre daný hardvér (inštrukcie procesora, veľkosť pamäte, spôsob adresovania atď.)
  - Cestou na prekonanie tejto nevýhody boli jazyky vyššej úrovne (Fortran, Algol, Pascal, C, C++, Java, ...)
  - paralelné programy dnes pripomínajú sekvenčné programy z 50-tych rokov – programátor musí vedieť typ architektúry (synchrónna, asynchrónna, distribuovaná), počet procesorov atď. Neexistuje „univerzálny" paralelný programovací jazyk typu C, Java.
- **Riešenie v dvoch krokoch:**
  - abstraktné riešenie (nezávislé na architektúre)
  - efektívne „zjemnenie" pre konkrétnu paralelnú architektúru

Základné črty teórie, ktorú budeme používať:

- **Nedeterminizmus** (je to jednoduchšie s ním a postupným zjemnením môže byt odstránený); niektoré systémy sú zo svojej povahy nedeterministické, napr. OS
- **absencia „control flow“**
- **synchrónnosť** a **asynchrónnosť**
- **stavy** a **priradenia** (prechodové systémy)
- **kód programu** nie je prepletený s dôkazom
- **správnosť** (závisí len od programu) a **zložitosť** (závisí od programu a jeho implementácie na konkrétnej architektúre) **sú oddelené**

UNITY

Unbounded **N**ondeterministic **I**terative **T**ransformations

- **Programy:**
  - deklarácie premenných
  - špecifikácia počiatočných hodnôt
  - množina priradení
- **Vykonanie:**
  - začína zo stavu vyhovujúceho vstupnej (počiatočnej) podmienke
  - program sa vykonáva donekonečna
  - v každom kroku sa nedeterministicky vyberie priraďovací príkaz a ten sa vykoná
  - každý príkaz sa vyberie nekonečne veľa krát
- **Nešpecifikuje sa:**
  - kedy sa príkaz vykoná
  - kde sa vykoná

UNITY

Programy sa ďalej alokujú na konkrétne architektúry, kde sa už špecifikuje viac

Stav sa volá *pevný bod*, ak vykonaním ľubovoľného príkazu program prejde do toho istého stavu

Predikát FP charakterizuje pevný bod

Stabilný predikát je taký, ktorý keď raz platí, platí potom už stále

(**Pozor:** angl. *eventually* znamená určite niekedy/raz iste)

(FP je stabilný)

Príklad: Určenie termínu schôdzky

- Úloha: určiť najbližší vyhovujúci čas (pre 3 osoby), kedy sa môžu stretnúť
- F má voľný len každý pondelok, tak vždy povie dátum najbližšieho pondelka; f, g, h sú funkcie zodpovedajúce osobám F, G, H („čo povedia“)
- $\text{com}(t) \equiv \{\ t = f(t) = g(t) = h(t)\ \}$ (boolovská funkcia)

- Špecifikácia: dané sú monotónne neklesajúce číselné fukcie f, g, h také, že pre každé t platí:
  - $f(t) \geq t \wedge g(t) \geq t \wedge h(t) \geq t$
  - $f(f(t)) = f(t) \wedge g(g(t)) = g(t) \wedge h(h(t)) = h(t)$
  - existuje z také, že $\text{com}(z)$ platí

- Treba nájsť program, ktorý má nasledujúci stabilný predikát:
    - $r = \min\{\, t \mid \mathrm{com}(t) \,\}$

- Rôzne stratégie:
    - F, G, H sedia za okrúhlym stolom a posielajú si lístočky s najbližším vyhovujúcim časom; keď lístok obehne dookola bez zmeny, tak je to dohodnuté
    - ústredný koordinátor: každý mu pošle svoj návrh, ten naspäť pošle všetkým maximum; opakuje sa to, až koordinátor nedostane naspäť to isté, čo poslal
    - aukcia (kto dá viac), prekrikovanie

UNITY riešenia 1

Program P1

  assign $r := \min\{ u \mid 0 \leq u \leq z \wedge \mathrm{com}(u)\}$

end{P1}

na sekvenčnej architektúre: zložitosť $O(z)$

keď má z procesorov: $O(\log z)$ krokov

nevýhoda: zbytočne testuje hodnoty, napríklad u také, že $t < u < f(t)$

UNITY riešenia 2

Program P2

initially $r = 0$

assign $r := f(r)$ ▯ $r := g(r)$ ▯ $r := h(r)$

end {P2}

Dôkaz správnosti:

- 1: invariant $(0 \le r) \wedge \forall u(0 \le u < r \Rightarrow \neg \mathrm{com}(u))$

  (z toho vieme, že $r \le z$)

  ukážeme, že na začiatku je true a že vykonanie hociktorého príkazu ho zachováva
- 2: FP $\equiv$ $r = f(r) \wedge r = g(r) \wedge r = h(r)$ t.j. FP $\equiv$ $\mathrm{com}(r)$

Pokračovanie dôkazu

- z 1 a 2 vyplýva, že ak program dosiahne pevný bod, tak tento je najskorším časom stretnutia
- Treba ešte ukázať, že každé vykonávanie programu vedie k pevnému bodu
- 3: Ukážeme, že ak $\neg$FP $\wedge$ r = K v nejakom bode výpočtu, tak neskôr bude platiť r > K
  - z 1 vieme, že r nemôže stúpať cez z, teda raz určite bude musieť FP platiť
  - z 2 máme $\neg$FP $\wedge$ r = K $\equiv$ K < f(K) $\vee$ K < g(K) $\vee$ K < h(K);
  - nech K < f(K); ale raz sa r := f(r) musí vykonať a tak sa r zvýši

UNITY riešenia 3

Riešenie s „centrálnym koordinátorom“:

Program P3
    initially r = 0
    assign r := max{ $f(r)$, $g(r)$, $h(r)$ }
end{P3}

Alokovanie na von Neumannovskom počítači

opakovať sekvenciu $r := f(r)$; $r := g(r)$; $r := h(r)$ až kým sa nedosiahne pevný bod, teda príkaz $r := h(g(f(r)))$

môže byť výhodnejšie častejšie aplikovať f než g, h

$r := f(r)$; $r := g(r)$; $r := f(r)$; $r := h(r)$

$r := h(f(g(f(r))))$

P2 možno alokovať na počítač s 3 procesormi, každý pre jednu osobu

správnosť týchto prístupov aj programu P3 plynie z dôkazu správnosti pre P2

UNITY Program Structure

Program program_name
  declare declare_section
  always always_section
  initially initially_section
  assign assign_section
end

program_name: string of text
  (ak je telo sekcie prázdne, môžeme zodpovedajúce kľúčové slovo vynechať)

declate_section: PASCAL like (int, boolean, array, set…)

always_section: definuje niektoré premenné ako funkcie iných; podmienky, ktoré vždy platia (invarianty)

initially_section: definujú počiatočné hodnoty premenných; neinicializované majú ľubovoľnú hodnotu

assign_section: obsahuje množinu priraďovacích príkazov

UNITY Program Structure 2

- Vykonávanie programu začína v stave, keď premenné majú hodnoty priradené v initially_section

- V každom kroku jeden príkaz je vykonaný v náhodnom poradí

- Počas nekonečného výpočtu každý príkaz je vykonaný nekonečne veľa krát

- Stav programu sa volá *FIXED POINT*, ak vykonanie ktoréhokoľvek príkazu tento stav nezmení

- Programy nemajú vstupno-výstupné príkazy

UNITY Program Structure 3

Assignment Statement

- $x, y, z := 0, 1, 2$
- $x, y := 0, 1 \; || \; z := 2$

□ $\langle || \; j\colon 0 \leq j \leq N :: A[j] := B[j] \rangle$
znamená $A[0] := B[0] \; || \; \ldots \; || \; A[N] := B[N]$

- $x := -1$ if $y < 0$ ~ 0 if $y = 0$ ~ 1 if $y > 0$

UNITY Program Structure 3

Structure of Assignment Statement (assign_stat)

assign_stat $\rightarrow$ assign_comp { || assign_comp }

Assignment component (assign_comp)

assign_comp $\rightarrow$ enum_assign | quantif_assign

premenná sa môže vyskytnúť aj viac ráz na ľavej strane, je však zodpovednosťou programátora, že všetky hodnoty, ktoré sú jej priradené, sú identické

každá assignment-component je vykonávaná nezávisle a simultánne

|| (dve čiary): súčasné (synchrónne) vykonanie

{...}: nula alebo viac krát

UNITY Program Structure 4

Enumerated assignment (enum_assign)
enum_assign → variable_list := expr_list

variable_list → variable { , variable }
expr_list → simple_expr_list | conditional_expr_list

simple_expr_list → expr { , expr }
conditional_expr_list → simple_expr_list if bool_expr
{ ~ simple_expr_list if bool_expr }

Expression (expr), Boolean expression (bool_expr): PASCAL like

- hodnoty všetkých expr na pravej strane a indexov na ľavej strane sú vyhodnotené a potom priradené premenným na ľavej strane

- Conditional_expr: ak ich je viac true, tak zodpovedajúce simple_expr_list musia mať rovnakú hodnotu – musí to byť DETERMINISTICKÉ

UNITY Program Structure 4

Príklady:

vymenenie obsahu x, y:

$$x, y := y, x$$

absolútna hodnota y je x:

$$x := y \text{ if } y \geq 0 \sim -y \text{ if } y \leq 0$$

$$sum, j := sum + A[j], j + 1 \text{ if } j < N$$

UNITY Program Structure 5

Quantified assignment (quant_assign)

    quant_assign $\rightarrow \langle$ || quant assign_stat $\rangle$

Quantification (quant)

    quant $\rightarrow$ variable_list: bool_expr ::

premenné z variable_list sa nazývajú *viazané*

rozsah quant je daný zátvorkami $\langle\ \rangle$

"prípad vyhovujúci quant" je množina hodnôt viazaných premenných pre ktoré platí bool_expr

quant_assign znamená nula alebo viac assign_comp získaných (z assign_stat) nahradením viazaných premenných ich „prípadmi", „prípadov" musí byť konečne veľa

UNITY Program Structure 5

Príklady

A[0..N], B[0..N] of int, treba priradiť max(A[j], B[j]) do A[j]

$\langle \; || \; j \colon 0 \le j \le N :: A[j] := \max(A[j], B[j]) \; \rangle$

Priradenie jednotkovej matice do U[0..N, 0..N]

$\langle \; || \; j, k \colon 0 \le j \le N \wedge 0 \le k \le N :: U[j, k] := 0 \text{ if } j \ne k \sim 1 \text{ if } j = k \; \rangle$

$\langle \; || \; j, k \colon 0 \le j \le N \wedge 0 \le k \le N \wedge j \ne k :: U[j, k] := 0 \; \rangle \; || \; \langle \; || \; j \colon 0 \le j \le N :: U[j, j] := 1 \; \rangle$

$\langle \; || \; j \colon 0 \le j \le N :: U[j, j] := 1 \; || \; \langle \; || \; k \colon 0 \le k \le N \wedge j \ne k :: U[j, k] := 0 \; \rangle\rangle$

UNITY Program Structure 6: Assign section

assign_section → statement_list

statement_list → statement {□ statement }

statement → assign_stat | quantified_statement_list

quantified_statement_list → ⟨ □ quant statement_list⟩

□(obdĺžniček): separátor medzi statements, ich počet musí byť konečný

Obmedzenie: bool_expr v quant nesmie obsahovať premenné, ktorých hodnota sa môže zmeniť počas behu programu

Toto obmedzenie zaručuje, že množina statements je pevná - statements sa netvoria počas výpočtu

UNITY Program Structure 6: Assign section

Príklady:

Priradenie jednotkovej matice do U[0..N, 0..N]

$(N + 1)^2$ statements:

$\langle \square\ j, k\!: 0 \le j \le N \wedge 0 \le k \le N :: U[j, k] := 0 \text{ if } j \ne k \sim 1 \text{ if } j = k \rangle$

2 statements:

$\langle\ ||\ j, k\!: 0 \le j \le N \wedge 0 \le k \le N \wedge j \ne k :: U[j, k] := 0\ \rangle$

$\square\ \langle\ ||\ j\!: 0 \le j \le N :: U[j, j] := 1\ \rangle$

N + 1 statement_lists, každý z nich má 2 statements

$\langle \square\ j\!: 0 \le j \le N :: U[j, j] := 1\ \square\ \langle\ ||\ k\!: 0 \le k \le N \wedge j \ne k :: U[j, k] := 0\ \rangle\rangle$

UNITY Program Structure 7: Initially section

syntax rovnaká ako assign_section, namiesto symbolu := sa používa =

definuje iniciálne hodnoty premenných

premenné sa vyskytujú na ľavej strane najviac raz

existuje usporiadanie rovností také, že každá *premenná v kvantifikácii* je buď viazaná alebo sa nachádza na ľavej strane nejakej predchádzajúcej rovnosti

existuje usporiadanie za quantified equations, že *každá premenná na pravej strane alebo v indexe* sa nachádza na ľavej strane nejakej predchádzajúcej rovnosti

tieto dve podmienky hovoria, že iniciálne hodnoty sú dobre definované

UNITY Program Structure 7: Initially section

initially_section definuje *inicial condition*; je to najsilnejší predikát, ktorý platí na začiatku

získa sa nahradením □ a || za $\wedge$ a podmienené výrazy tvaru

x = e_0 if b_0 ~ ... ~ e_n if b_n výrazom

$$(b_0 \Rightarrow (x = e_0)) \wedge \ldots \wedge (b_n \Rightarrow (x = e_n))$$

Čo *nie je* ekvivalentné s

$$((x = e_0) \wedge b_0) \vee \ldots \vee ((x = e_n) \wedge b_n)$$

napr. y = 2 if false

UNITY Program Structure 8: Initially section, príklady

- nemožno zameniť □ na ||, v takom prípade by neexistovalo usporiadanie príkazov také, že N je iniciované pred jeho použitím
  - initially N = 3 □ $\langle$ || k: $0 \le k < \mathrm{N}$ :: A[N – k] = k $\rangle$

- preloženie počiatočnej podmienky pre

  $\langle$ □ j, k: $0 \le j \le \mathrm{N} \wedge 0 \le k \le \mathrm{N}$ :: U[j, k] = 0 if $j \neq k$ ~ 1 if $j = k$ $\rangle$

  vyzerá nasledovne

$\langle\wedge$ j, k : $0 \le j \le \mathrm{N} \wedge 0 \le k \le \mathrm{N}$ :: ($j \neq k \Rightarrow$ U[j, k] = 0) $\wedge$ ($j = k \Rightarrow$ U[j, k] = 1) $\rangle$

Sorting 1

Program sort1

assign

$\langle \square\, j\!: 0 \leq j < N ::$

$\quad A[j], A[j + 1] := A[j + 1], A[j] \text{ if } A[j] > A[j + 1] \rangle$

end{sort1}

Program sort2

assign

$\langle \| \; j\!: 0 \leq j < N \wedge \text{even}(j) ::$

$\quad A[j], A[j + 1] := A[j + 1], A[j] \text{ if } A[j] > A[j + 1] \rangle$

$\square \langle \; \| \; j\!: 0 \leq j < N \wedge \text{odd}(j) ::$

$\quad A[j], A[j + 1] := A[j + 1], A[j] \text{ if } A[j] > A[j + 1] \rangle$

end{sort2}

Sorting 2

Program sort3

assign

$\langle \Box\ k\!: 0 \le k \le 1 ::$

$\quad \langle \,||\ j\!: 0 \le j \le N \wedge (k = j \bmod 2) ::$

$\quad A[j], A[j + 1] := A[j + 1], A[j] \text{ if } A[j] > A[j + 1] \rangle\rangle$

end{sort3}

Binomial Coefficients

C(n, k) značí „n nad k“

C(n, 0) = C(n, n) = 1, C(n, k) = C(n – 1, k – 1) + C(n – 1, k)

Program binomial

assign

$\langle$ □ n: $0 \leq n < N$ ::

$c[n, 0] := 1$ □ $c[n, n] := 1$

□ $\langle$ □ k: $0 < k < n$ ::

$c[n, k] := c[n-1, k-1] + c[n-1, k]$ $\rangle\rangle$

end{binomial}

poradie je ľubovoľné: $c[n, k]$ môže byť priradená hodnota, hoci $c[n-1, k-1]$ alebo $c[n-1, k]$ nie je ešte vypočítané

UNITY Program Structure 9: Always section

- syntax rovnaká ako v initially_section
- premenná na ľavej strane sa nazýva *transparentná*, ak je funkciou netransparentných a nie je na ľavej strane inicializácií alebo priradení
- rovnaké obmedzenia ako v initially_section
- Príklad:
  - *ne* – počet zamestnancov
  - *nm*, *nf* – počet mužov, žien
  - always *ne* = *nm* + *nf*
- always_section nie je nevyhnutná, ale
  - invarianty
  - transparentné premenné možno chápať ako „makro-inštrukcie"
  - efektívna implementácia – vyhodnotenie TP môže byť odložené kým treba alebo kým sa nezmení hodnota premennej ktorú definuje

UNITY Program Structure 10

x :=< min j: $0 \leq j \leq$ N :: A[j] >

Quantified Expression

expr $\rightarrow \langle$ op quant expr $\rangle$

op $\rightarrow$ min | max | + | $\times$ | $\wedge$ | $\vee$ | $\equiv$ | ...

ak neexistuje „prípad“, potom expr (naľavo) má hodnotu neutrálneho prvku operátora op

neutrálne prvky:

min	max	+	$\times$	$\wedge$	$\vee$	$\equiv$
∞	$-\infty$	0	1	true	false	true

Príklady

1. $\langle \vee\, j$: $0 \leq j \leq$ N :: $b[j]$ $\rangle$
   true, ak nejaké $b[j]$ je true
2. $\langle$ min j: $0 \leq j \leq$ N :: A[j] $\rangle$
   najmenší prvok poľa A[0..N]
3. $\langle + j$: $0 \leq j \leq$ N $\wedge$ A[j] < A[k] :: 1 $\rangle$
   počet prvkov menších ako A[k], ak A[k] je v A[0..N]

Programming Logic

- $\{p\}$ s $\{q\}$ ak platí p a s skončí tak bude platiť q
- p: precondition, q: postcondition, s: statement
- predikáty nie sú viazané s nejakými miestami v programe, ako u sekvenčných programov
- logika pre uvažovanie o nekonečných postupnostiach programových stavov
- tvrdenie $\{\text{true}\}$ t $\{p\}$ (pre príkaz t programu F) hovorí, že niekedy raz (*eventually*) p bude platiť
- vlastnosti programov:
  - Safety („nič zlé sa nestane")
  - Progress („niečo dobré sa stane")
- budeme predpokladať, že program má aspoň jeden príkaz
- <u>viacero výrokov v hypotéze znamená konjukciu</u>
- <u>viacero výrokov v závere znamená disjukciu</u>

Basic Concepts

hypotéza
 ----
  záver

$\{p\}\ s\ \{\text{true}\}$

$\{\text{false}\}\ s\ \{q\}$

$\{p\}\ s\ \{\text{false}\}$

$\neg p$

Basic Concepts

$$\frac{\{p\}\ s\ \{q\},\ \{p'\}\ s\ \{q'\}}{\{p \vee p'\}\ s\ \{q \vee q'\}}$$

$$\frac{p' \Rightarrow p,\ \{p\}\ s\ \{q\},\ q \Rightarrow q'}{\{p'\}\ s\ \{q'\}}$$

Dokazovanie tvrdení o priraďovacích príkazoch

$\{p\}\ x := \mathrm{E}\ \{q\}$

1. V q nahradíme za x výraz E (výsledok označíme $q^x{}_{\mathrm{E}}$)

Príklad: {x<2} x:= x+3 {x <10}; $q^x{}_{\mathrm{E}}$ = x+3 < 10

2. Ukážeme, že $p \Rightarrow q^x{}_{\mathrm{E}}$ x<2 ⇒ x+3 < 10

ak $\mathrm{E} = \mathrm{e}_0$ if $b_0 \sim \ldots \sim \mathrm{e}_n$ if b_n tak výraz $q^x{}_{\mathrm{E}}$ bude takýto:
$(b_0 \Rightarrow q^x{}_{\mathrm{e}0}) \wedge \ldots \wedge (b_n \Rightarrow q^x{}_{\mathrm{e}n})) \wedge ((\neg b_0 \wedge \ldots \wedge \neg b_n) \Rightarrow q)$

môže sa písať aj v tvare
$\{p \wedge b_0\}\ x := \mathrm{e}_0\ \{q\}$
…
$\{p \wedge b_n\}\ x := \mathrm{e}_n\ \{q\}$
$\{p \wedge (\neg b_0 \wedge \ldots \wedge \neg b_n) \Rightarrow q\}$

- Poznámka.
  - Znak ⇒ budeme používať ako označenie logickej implikácie.
  - Znak ≡ označuje jednak logickú ekvivalenciu, jednak syntaktickú ekvivalenciu. Význam je zrejmý z kontextu.

Kvantifikované tvrdenia

Majme program F (ktorý obsahuje aspoň jeden príkaz)

Budem uvažovať dva typy tvrdení:

1. $\langle\ \forall s$: s *in* F :: $\{p\}\ s\ \{q\}\ \rangle$
2. $\langle\ \exists s$: s *in* F :: $\{p\}\ s\ \{q\}\ \rangle$

Príklad: nech F $\equiv \langle\ \square\ j: b(j)\ ::\ t(j)\ \rangle$

pre 1. treba ukázať, že $\{p \wedge b(j)\}\ t(j)\ \{q\}$

pre 2. treba ukázať, že existuje j také, že $b(j)$ platí a súčasne $\{p \wedge b(j)\}\ t(j)\ \{q\}$

Kvantifikované tvrdenia - príklady

Príklady ($\forall s$, $\exists s$ znamená pre každý príkaz a existuje príkaz v programe *F*)

1. Hodnota *x* neklesá: (univerzálne kvantifikované cez všetky číselné hodnoty *k*)

$\langle\ \forall s :: \{x = k\}\ s\ \{x \geq k\}\ \rangle$

2. Správa ostáva v kanáli až kým nie je prijatá
   - inch = in channel (v kanáli)
   - rcvd = received (prijatá)

$\langle\ \forall s :: \{\text{inch}\}\ s\ \{\text{inch} \vee \text{rcvd}\}\ \rangle$

3. Ako 2. ale prijaté správy sú odstránené z kanálu

$\langle\ \forall s :: \{\text{inch} \wedge \neg\text{rcvd}\}\ s\ \{(\text{inch} \wedge \neg\text{rcvd}) \vee (\neg\text{inch} \wedge \text{rcvd})\}\ \rangle$

4. Hodnota *x* je neklesajúca a stúpne

$\langle\ \forall s :: \{x = k\}\ s\ \{x \geq k\}\ \rangle$

$\langle\ \exists s :: \{x = k\}\ s\ \{x > k\}\ \rangle$

Výpočtový model

- množina výpočtových postupností priradená ku každému programu
- nekonečné postupnosti, každá reprezentuje jeden možný beh (výpočet) programu
- R_j = j-ty prvok postupnosti R, $j \geq 0$
- $R_j = (R_j.\text{state}, R_j.\text{label})$
  state (stav) – hodnota všetkých premenných
  label – príkaz vykonaný v j-tom kroku

Výpočtový model

- R_0.state – iniciačný stav (ak nie sú dané iniciačné hodnoty pre všetky premenné, nemusia byť rovnaké pre rôzne R)
- R_{j+1}.state je jednoznačne určený R_j.state a R_j.label-om, teda R_0.state a $\{ R_k.\text{label} \mid 0 \leq k < j \}$ určujú R_j.state
- Spravodlivý výber príkazov: $\forall R \forall s$, R_j.label = s pre nekonečne veľa j
- $p[R_j] = p$ platí v stave R_j.state
- $\{p\}$ s $\{q\}$ znamená
  $\forall R \forall j$: $(p[R_j] \wedge R_j.\text{label} = s) \Rightarrow q[R_{j+1}]$

Základné pojmy

- unless (a špeciálne prípady stable a invariant)
- ensures
- leads-to (označenie $\rightarrow$)

- **Safety:** p unless q, p is stable, p is invariant
- **Progress:** p ensures q, $p \rightarrow q$

- používame univerzálnu kvantifikáciu

$$x = k \text{ unless } x > k$$

znamená $\langle\ \forall k :: x = k \text{ unless } x > k\ \rangle$

Unless

(daný je program F)

p unless $q \equiv \langle \forall s\text{: } s \text{ in } \mathrm{F} :: \{p \wedge \neg q\}\ S\ \{p \vee q\} \rangle$

ak p je true a q nie je true v nasledujúcom kroku, p ostáva true alebo q sa stane true

ak v nejakom mieste výpočtu F platí p, tak

- q nebude nikdy platiť a p bude stále platiť
- q bude určite raz (*eventually*) platiť a p platí aspoň pokiaľ q začne platiť

$(p \wedge \neg q)[R_j] \Rightarrow (p \vee q)[R_{j+1}]$

Z "p unless q" možno odvodiť: $p[R_j] \Rightarrow$

$\langle \forall k\text{: } k \geq j :: (p \wedge \neg q)[R_k] \rangle \vee$ ($p \wedge \neg q$ platí vždy)

$[\ \langle \exists m\text{: } m \geq j :: q[R_m] \rangle \wedge$ (nakoniec platí)

$\langle \forall k\text{: } j \leq k < m :: (p \wedge \neg q)[R_k] \rangle\ ]$ (dovtedy platí $p \wedge \neg q$)

Unless

Príklady:

1. x neklesne
    - $x = k$ unless $x > k$
    - $x \geq k$ unless $x > k$
    - $x \geq k$ unless false

2. Správa je v kanáli až kým nie je prijatá a potom je odstránená z kanálu
    - inch $\wedge$ $\neg$rcvd unless $\neg$inch $\wedge$ rcvd

Špeciálne prípady unless (stable, invariant)

- p is stable $\equiv$ p unless false
- ak p is stable, tak ak sa raz stane true, potom ostane vždy true (t.j. $\{p\}$ s $\{p\}$ pre každé s)
- q is invariant $\equiv$ (initial condition $\Rightarrow q$) $\wedge$ q is stable
- invariant je vždy true
- **Pozorovanie:** Ak I, J sú stable (pre program F), potom aj I $\wedge$ J, I $\vee$ J sú stable. Podobne pre invarianty.
- **Označenie:** constant p: ak p aj $\neg p$ sú stable

Špeciálne prípady unless (stable, invariant)

Substitučná axióma: Ak $x = y$ je invariant programu F, tak môžeme x zameniť za y vo všetkých vlastnostiach programu F.

Ak I je invariant, je zameniteľný s true a vice versa.

Dôsledok: p unless q, $\neg q$ is invariant $\Rightarrow$ p is stable.

Dôkaz:

$\neg q \equiv$ true	/* substitučná axióma */
p unless q	/* predpoklad */
p unless false	/* z predchádzajúcich */
p is stable	/* z definície */

Ak I je invariant, tak p môže byť zameniteľné s $I \wedge p$ alebo $\neg I \vee p$ a podobne.

Dokázať, že p je stabilné: stačí ukázať, že $I \wedge p$ je stabilné, čo môže byť ľahšie, ako pre samotné p.

Ensures

(daný je program F)

p ensures $q \equiv p$ unless $q \wedge \langle \exists s$: s *in* F :: $\{p \wedge \neg q\}$ S $\{q\}\rangle$

Ak p je true v nejakom bode výpočtu, p ostane true, pokiaľ q je false (p unless q) a určite raz (*eventually*) sa stane q true („raz naň dôjde") po vykonaní nejakého príkazu s

Z „ p ensures q" možno odvodiť:

$p[R_j] \Rightarrow$

$\langle \exists m$: $m \geq j$ :: $q[R_m] \rangle \wedge$ (niekedy platí q)

$\langle \forall k$: $j \leq k < m$ :: $(p \wedge \neg q)[R_k] \rangle$ (dovtedy platí $p \wedge \neg q$)

Ensures - príklady

1. x je neklesajúca a raz stúpne

   $x = k$ ensures $x > k$, teda

   $\langle\ \forall k :: x = k$ unless $x > k\ \rangle$ a

   $\langle\ \forall k :: \langle\ \exists s :: \{x = k\}\ s\ \{x > k\}\ \rangle\rangle$

2. Program P:

$x := 0$ if $x < 0 \quad \square \quad x := 0$ if $x > 0$

"$x \neq 0$ ensures $x = 0$"   Platí v P?

Nie je to vlastnosť tohto programu P.

Ensures - príklady

lebo neexistuje s také, že $\{x \neq 0\}$ s $\{x = 0\}$, pretože ak je to bol prvý príkaz (teda $x := 0$ if $x < 0$), potom

$$\{x \neq 0\}\ x := 0 \text{ if } x < 0\ \{x = 0\}$$

$$\{x \neq 0 \wedge x < 0\}\ x := 0\ \{x = 0\}$$

priradenie $x := 0$ sa ale pre $x > 0$ „nevykoná“ a tak výstupná podmienka $x = 0$ nebude platiť

rovnako nemôže vyhovovať ani druhé priradenie

Leads-to

program F má vlastnosť „ *p* leads-to *q*" ($p \to q$)

ak táto vlastnosť môže byť odvodená konečným počtom aplikácií nasledujúcich odvodzovacích pravidiel (tvar hypotéza / záver):

1. p ensures q / $p \to q$
2. $p \to q,\ q \to r$ / $p \to r$ /* tranzitivita */
3. $$\frac{\langle \forall m : m \in W :: p(m) \to q \rangle}{\langle \exists m : m \in W :: p(m) \rangle \to q}$$

pre nejakú indexovú množinu W /* disjunkcia */

Leads-to

- $p \rightarrow q$
- Ak p sa stane true, tak q je alebo bude true.

- Nemožno však tvrdiť, že p ostane true, až kým q nie je true.

- Z „ p leads-to q“ možno odvodiť:
$p[R_j] \Rightarrow \langle\ \exists m\colon m \geq j :: q[R_m]\ \rangle$ /* q niekedy platí */

Leads-to príklady

Z pravidla disjunkcie dostaneme, že platí :

$p_1 \rightarrow q, \;\; p_2 \rightarrow q \;/\; p_1 \vee p_2 \rightarrow q.$

$\langle \forall m\!: m \in \{1, 2\} :: p(m) \rightarrow q \,\rangle$

$\langle\, \exists m\!: m \in \{1, 2\} :: p(m) \,\rangle \rightarrow q.$

Pre program P z príkladov pre ensures dokážeme, že
“$x \neq 0 \rightarrow x = 0$”.

$x \neq 0$ ensures $x \geq 0$	/* z programu */
$x \neq 0 \rightarrow x \geq 0$	/* z predchádzajúceho */
$x \geq 0$ ensures $x = 0$	/* z programu */
$x \geq 0 \rightarrow x = 0$	/* z predchádzajúceho */
$x \neq 0 \rightarrow x = 0$	/* z tranzitivity */

Pevný bod

je to stav programu, ktorý sa ďalším vykonávaním programu už nemení

definujme predikát FP pre program G:

- FP $\equiv \langle \forall s$: s *in* G $\wedge$ s je "x := E" :: x = E $\rangle$
- $FP[R_j] \equiv \langle \forall k: k \geq j :: R_k.\text{state} = R_j.\text{state} \rangle$

Príklady

1. $k := k + 1$

$FP \equiv k = k + 1 \equiv$ false

2. $k := k + 1$ if $k < N$

$FP \equiv [k < N \Rightarrow k = k + 1] \equiv k \geq N$

3. $\langle \square\ j: 0 \leq j < N :: m = \max(m, A[j]) \rangle$,

$FP \equiv \langle \wedge j: 0 \leq j < N :: m = \max(m, A[j]) \rangle \equiv$

$\langle \wedge j: 0 \leq j < N :: m \geq A[j] \rangle \equiv$

$m \geq \langle \max j: 0 \leq j < N :: A[j] \rangle$

Vlastnosti unless 1

- Reflexívnosť a antireflexívnosť:

  p unless p

  p unless $\neg p$

  Dôkaz: {false} s $\{p\}$, $\{p\}$ s {true} pre $\forall s$

- Zoslabenie:

  p unless q, $q \Rightarrow r$ / p unless r

  Dôkaz:

  $\{p \wedge \neg q\}$ s $\{p \vee q\}$. Z $q \Rightarrow r$, $\neg r \Rightarrow \neg q$ a teda

  $p \wedge \neg r \Rightarrow p \wedge \neg q$

  $p \vee q \Rightarrow p \vee r$ a teda

  $\{p \wedge \neg r\}$ S $\{p \vee r\}$, t.j. „p unless r".

Vlastnosti unless 2

- Disjukcia

  p unless q, p' unless q' /

  / $(p \vee p')$ unless $(\neg p \wedge q') \vee (\neg p' \wedge q) \vee (q \wedge q')$

- Konjunkcia

  p unless q, p' unless q' /

  / $(p \wedge p')$ unless $(p \wedge q') \vee (p' \wedge q) \vee (q \wedge q')$

Dôkaz: $\wedge$ a $\vee$ môžu byť aplikované na post- a pre-conditions, teda

$\{(p \wedge \neg q) \wedge (p' \wedge \neg q')\}\ s\ \{(p \vee q) \wedge (p' \vee q')\}$,

potom použiť pravidlo

$p' \Rightarrow p,\ q \Rightarrow q',\ \{p\}\ S\ \{q\}\ /\ \{p'\}\ S\ \{q'\}$

Vlastnosti unless 3

Jednoduchá disjukcia

p unless q, p' unless q' / $p \vee p'$ unless $q \vee q'$

Jednoduchá konjunkcia

p unless q, p' unless q' / $p \wedge p'$ unless $q \vee q'$

Dôkaz: z predošlých viet a zoslabenia

"Tranzitivita"

p unless q, q unless r / $p \vee q$ unless r

Dôkaz: disjunkcia a zoslabenie

Vlastnosti unless 4

- Dôsledok 1.
  $p \Rightarrow q$ / p unless q
  Dôkaz: zoslabením. Podľa reflexívnosti p unless p,
  z predpokladu $p \Rightarrow q$ dostaneme p unless q

- Dôsledok 2.
  $\neg p \Rightarrow q$ / p unless q
  Dôkaz: zoslabením. Podľa antireflexívnosti p unless $\neg p$,
  z predpokladu $\neg p \Rightarrow q$ dostaneme p unless q

Vlastnosti unless 5

- Dôsledok 3.

  $[p \text{ unless } q \vee r] \equiv [p \wedge \neg q \text{ unless } q \vee r]$

  Dôkaz:

  Predpokladajme „p unless $q \vee r$".
  - antireflexivita: $\neg q$ unless q
  - jednoduchá konjukcia: $p \wedge \neg q$ unless $q \vee r$

  Predpokladajme "$p \wedge \neg q$ unless $q \vee r$"
  - z dôsledku 1: $p \wedge q$ unless q
  - jednoduchá disjunkcia: p unless $q \vee r$

Vlastnosti unless 6

Dôsledok 4.

$p \vee q$ unless r / p unless $q \vee r$

Dôkaz:

$p \vee q$ unless r	/* predpoklad */
$\neg\, q$ unless q	/* antireflexívnosť */
$p \wedge \neg q$ unless $q \vee r$	/* jednoduchá konjunkcia */
p unless $q \vee r$	/* z dôsledku 3. */

Dôsledok 5.

$\langle \forall j::p_j$ unless $p_j \wedge q_j\rangle$ / $\langle \forall j::p_j \rangle$ unless $\langle \forall j::p_j\rangle \wedge \langle \exists j::q_j\rangle$

Dôkaz: indukciou, IP (N = 2) použitím konjukcie

Vlastnosti ensures 1

Reflexívnosť:

p ensures p

Dôkaz: p unless p a $\forall$s platí $\{p \wedge \neg p\}$ s $\{p\}$. Keďže každý program obsahuje aspoň jeden príkaz, tvrdenie je dokázané.

Zoslabenie:

p ensures q, $q \Rightarrow r$ / p ensures r

Dôkaz:

- zoslabenie pre unless platí, stačí teda ukázať, že ak $\exists$s také, že $\{p \wedge \neg q\}$ s $\{q\}$, potom $\{p \wedge \neg r\}$ s $\{r\}$
- Vyplýva to z $q \Rightarrow r$, $\neg r \Rightarrow \neg q$ a teda

$$p \wedge \neg r \Rightarrow p \wedge \neg q$$

Vlastnosti ensures 2

Nemožnosť:

p ensures false / $\neg p$

Dôkaz: z "p ensures false" vieme, že $\exists$s, {p} s {false}

teda $p \equiv$ false

Konjunkcia

p unless q, p' ensures q' /

$(p \wedge p')$ ensures $(p \wedge q') \vee (p' \wedge q) \vee (q \wedge q')$

Dôkaz: podobný ako pre unless

Disjukcia

p ensures q / $p \vee r$ ensures $q \vee r$

Dôkaz: z definície

Vlastnosti ensures 3

Dôsledok 1.

$p \Rightarrow q$ / p ensures q

Dôkaz: p unless q a zoslabenie

Dôsledok 2.

$p \vee q$ ensures r / p ensures $q \vee r$

Dôkaz:

$p \vee q$ unless r /* z predpokladu */
p unless $q \vee r$ /* dôsledok 4 pre unless */
$p \vee q$ ensures r /* predpoklad */
p ensures $q \vee r$ /* konjunkcia a zoslabenie */

Vlastnosti ensures 4

Dôsledok 3.

p ensures $q \vee r$ / $p \wedge \neg q$ ensures $q \vee r$

Dôkaz:

p ensures $q \vee r$	/* predpoklad */
$(p \wedge q) \vee (p \wedge \neg q)$ ensures $q \vee r$	/* rozpísaný predpoklad */
$p \wedge \neg q$ ensures $(p \wedge q) \vee (q \vee r)$	/* z dôsledku 2 */
$p \wedge \neg q$ ensures $q \vee r$	/* z predchádzajúceho zoslabením */

Vlastnosti leads-to 1

Technika dôkazov: indukcia vzhľadom na počet odvodení (dĺžka dôkazu) = *štrukturálna indukcia*

Nemožnosť:

$p \rightarrow$ false / $\neg p$

Dôkaz.

Základný prípad:

- predpokladajme p ensures false
- z nemožnosti pre ensures máme $\neg p$

Indukčný krok: nech tvrdenie platí pre odvodenie dĺžky n.

Uvažujme dva prípady podľa spôsobu odvodenia $p \rightarrow$ false

1. prípad ($p \rightarrow r$, $r \rightarrow$ *false* / $p \rightarrow$ false):

z $r \rightarrow$ false vyplýva $\neg r$ *(podľa indukčného predpokladu, keďže toto odvodenie je kratšie)*

Vlastnosti leads-to 2

Keďže $\neg r$ *z* $p \rightarrow r$ *dostaneme* $p \rightarrow$ false,

A keďže odvodenie $p \rightarrow r$ je kratšie ako n opäť podľa indukčného predpokladu dostaneme $\neg p$

2.Prípad $<\forall m: m \in W :: p'(m) \rightarrow \mathit{false}>$

$$\frac{<\forall m: m \in W :: p'(m) \rightarrow \mathit{false}>}{<\exists m: m \in W :: p'(m)> \rightarrow \mathit{false}}$$

.

a $p = <\exists m: m \in W :: p'(m)>$

$\forall m: m \in W :: [p'(m) \rightarrow \text{false}] \;/\; \neg p'(m)$

teda

$\langle \forall m: m \in W :: \neg p'(m) \rangle$

$\neg \langle \exists m: m \in W :: p'(m) \rangle$

$\neg p$

Vlastnosti leads-to 3

Implikačná teoréma:

$p \Rightarrow q \;/\; p \rightarrow q$

Dôkaz:

$p \Rightarrow q$	/* predpoklad */,
p ensures q	/* z predpokladu */
$p \rightarrow q$	/* z definície leads-to */

Disjunkčná veta (všeobecná): pre ľubovoľnú množinu W platí

$\langle \forall m : m \in W :: p(m) \rightarrow q(m) \rangle \;/$

$/ \; \langle \exists m : m \in W :: p(m) \rangle \rightarrow \langle \exists m : m \in W :: q(m) \rangle$

Dôkaz: z predikátového počtu a implikačnou vetou

$\langle \forall m : m \in W :: q(m) \Rightarrow \langle \exists n : n \in W :: q(n) \rangle \rangle$

$\langle \forall m : m \in W :: q(m) \rightarrow \langle \exists n : n \in W :: q(n) \rangle \rangle$

$\langle \forall m : m \in W :: p(m) \rightarrow q(m) \rangle$ /* predpoklad */

$\langle \forall m : m \in W :: p(m) \rightarrow \langle \exists n : n \in W :: q(n) \rangle \rangle$ /* z tranzitivity predchádzajúcich: */

$\langle \exists m : m \in W :: p(m) \rangle \rightarrow \langle \exists m : m \in W :: q(m) \rangle$ /* z disjunkcie: */

Vlastnosti leads-to 4

Cancellation

$p \rightarrow q \vee b,\ b \rightarrow r\ /\ p \rightarrow q \vee r$

Dôkaz:

$b \rightarrow r$ /* predpoklad */

$q \rightarrow q$ /* triviálne */

$q \vee b \rightarrow q \vee r$ /* disjunkcia predchádzajúcich dvoch*/

$p \rightarrow q \vee r$ /* z predpokladu $p \rightarrow q \vee b$ a predchádzajúceho*/

Vlastnosti leads-to 5

Progress-Safety-Progress (PSP) veta:

$p \rightarrow q$, r unless b / $p \wedge r \rightarrow (q \wedge r) \vee b$

Dôkaz. *Základný prípad:*

- p ensures q, r unless b
- treba ukázať, že $p \wedge r \rightarrow (q \wedge r) \vee b$
- z konjunkcie pre ensures a zoslabenia pravej strany

- *Indukčný krok (tranzitivita):*
  - $p \rightarrow q'$, $q' \rightarrow q$, r unless b
  - $p \wedge r \rightarrow (q' \wedge r) \vee b$ /* (a) */
  - $p \wedge q' \rightarrow (q \wedge r) \vee b$ /* (b) */

Vlastnosti leads-to 6

cancellation na (a) a (b)

- *(disjunkcia):*

  $\forall \langle \forall m\colon m \in W :: p'(m) \to q \rangle$

  - $p = \langle \forall m\colon m \in W :: p'(m) \rangle$
  - r unless b

  $\forall \langle \forall m\colon m \in W :: p'(m) \wedge r \to (q \wedge r) \vee b \rangle$ /* (c) */

- z disjunkcie na (c):

  $\forall \langle \exists m\colon m \in W :: p'(m) \wedge r \rangle \to (q \wedge r) \vee b$

  $\forall \langle \exists m\colon m \in W :: p'(m) \rangle \wedge r \to (q \wedge r) \vee b$

  - $p \wedge r \to (q \wedge r) \vee b$ /* z definície p a predošlého */

Vlastnosti leads-to 7

Completion veta:

pre množinu predikátov p_j, q_j, $0 \leq j < N$:

$\langle \forall j{::}\ p_j \to q_j \vee b \rangle$, $\langle \forall j{::}\ q_j$ unless $b \rangle$ /

/ $\langle \wedge j{::}\ p_j \rangle \to \langle \wedge j{::}\ q_j \rangle \vee b$

Dôsledok 1 (konečná disjunkcia).

$p \to q,\ p' \to q' \ /\ p \vee p' \to q \vee q'$

Dôkaz: špeciálny prípad všeobecnej disjunkcie

Dôsledok 2.

$p \wedge b \to q,\ p \wedge \neg b \to q \ /\ p \to q$

Dôkaz: vyplýva z dôsledku 1

Dôsledok 3.

$p \to q$, r is stable / $p \wedge r \to q \wedge r$

Dôkaz: z PSP theorem

Indukčný princíp pre leads-to

- W – dobre založená (well founded) množina je taká, že má reláciu usporiadania $\angle$ takú, že každá podmnožina má najmenší prvok

- metrika M: $States \rightarrow W$
  - $States$: stavy programu
  - $M(S)$ nahradíme iba M, ak S je jasné z kontextu

$$\langle \forall m: m \in W :: p \wedge M = m \longrightarrow (p \wedge M \angle m) \vee q \rangle \,/\, p \longrightarrow q$$

Z každého stavu kde platí *p* program dosiahne stav, v ktorom platí *q* alebo dosiahne stav, v ktorom *p* platí a hodnota *M* je nižšia.

Keďže hodnota *M* nemôže klesať donekonečna, určite raz musí platiť *q* (*q holds eventually*).

Veta o FP

Veta: Pre každý predikát p platí: (FP $\wedge$ p) is stable.

Dôkaz: Ak FP platí, tak ďalším vykonávaním programu sa nemení jeho stav. Ak FP $\wedge$ p, tak aj naďalej FP $\wedge$ p, čiže FP $\wedge$ p is stable.

Dôsledok: $p \longrightarrow q$ / FP $\Rightarrow$ ($p \Rightarrow q$)

Dôkaz:

FP $\wedge \neg q$ is stable (t.j. FP $\wedge \neg q$ unless false)	/* teoréma o FP */
$p \longrightarrow q$	/* predpoklad */
$p \wedge$ FP $\wedge \neg q \longrightarrow$ false	/* PSP teoréma */
$\neg(p \wedge$ FP $\wedge \neg q)$ FP $\Rightarrow$ ($p \Rightarrow q$)	/* nemožnosť leads-to */

Spojenie programov 1

nech F, G sú dva UNITY-programy

spojenie programov F a G, označenie F ▯ G

- spoja sa zodpovedajúce si časti z F a G
- predpokladá sa, že nevznikajú nekonzistentcie s:

  premennými
  ich inicializáciou
  always sekciou

Spojenie programov 2

Spojovacia veta:

1. p unless q in F $\square$ G =

   p unless q in F $\wedge$ p unless q in G

2. p ensures q in F $\square$ G =

   [p ensures q in F $\wedge$ p unless q in G] $\vee$

   [p ensures q in G $\wedge$ p unless q in F]

3. (FP of F $\square$ G) = (FP of F) $\wedge$ (FP of G)

Dôkaz:

3. Vyplýva priamo z definície

1. p unless q in F $\square$ G

   = $\langle$ $\forall$s: s in F $\square$ G :: $\{p \wedge \neg q\}$ s $\{p \vee q\}$ $\rangle$

   = $\langle$ $\forall$s: s in F :: $\{p \wedge \neg q\}$ s $\{p \vee q\}$ $\rangle$ $\wedge$

   $\langle$ $\forall$s: s in G :: $\{p \wedge \neg q\}$ s $\{p \vee q\}$ $\rangle$

   = p unless q in F $\wedge$ p unless q in G

Spojenie programov 3

2. p ensures q in F $\square$ G

$= p$ unless q in F $\square$ G $\wedge \langle \exists$s: s in F $\square$ G :: $\{p \wedge \neg q\}$ s $\{q\} \rangle$

$= p$ unless q in F $\square$ G $\wedge$

[$\langle \exists$s: s in F :: $\{p \wedge \neg q\}$ s $\{q\} \rangle \vee$

$\langle \exists$s: s in F :: $\{p \wedge \neg q\}$ s $\{q\} \rangle$]

= [p unless q in F $\square$ G $\wedge \langle \exists$s: s in F :: $\{p \wedge \neg q\}$ s $\{q\} \rangle$] $\vee$

[p unless q in F $\square$ G $\wedge \langle \exists$s: s in G :: $\{p \wedge \neg q\}$ s $\{q\} \rangle$]

= [p unless q in F $\wedge$ p unless q in G

$\wedge \langle \exists$s: s in F :: $\{p \wedge \neg q\}$ s $\{q\} \rangle$] $\vee$

[p unless q in F $\wedge$ p unless q in G

$\wedge \langle \exists$s: s in G :: $\{p \wedge \neg q\}$ s $\{q\} \rangle$]

= [p ensures q in F $\wedge$ p unless q in G] $\vee$

[p ensures q in G $\wedge$ p unless q in F]

Spojenie programov 4

Dôsledky:

1. *p* is stable in F G = (*p* is stable in F) $\wedge$ (*p* is stable in G)

2. *p* unless *q* in F, *p* is stable in G / *p* unless *q* in F G

3. *p* is invariant in F, *p* is stable in G / *p* is invariant in F G

4. *p* ensures *q* in F, *p* is stable in G / *p* ensures *q* in F G

5. (lokalita) Ak niečo z nasledujúcich tvrdení platí pre F, kde *p* je lokálne k F (ak používa len premenné lokálne ku F), tak to platí aj pre F G (G je ľubovoľné):
    - *p* unless *q*,
    - *p* is invariant
    - *p* ensures *q*

Spojenie programov 5

<u>Podmienené vlastnosti:</u>

Majme dve množiny nepodmienených vlastností - hypotéza a záver

Podmienené vlastnosti majú tvar:

hypotéza / záver

Podmienená vlastnosť programu F: hypotéza aj záver môžu byť vlastnosťami programu F, G, F □ G pre nejaký generický program G

význam: vezmeme hypotézu ako predpoklad, záver sa dá dokázať z „textu“ F

Spojenie programov 6

Príklad:

Program F
declare x: int
assign

$$y := -y \text{ if } x \leq 0 \wedge y > 0$$
$$[] \quad x := x - 1$$

end{F}

nech x sa nemení mimo F (y sa môže meniť aj mimo F)

príklad podmienenej vlastnosti:

- Hypotéza: $y \neq 0$ is stable in F $[]$ G
- Záver: $y > 0 \longrightarrow y < 0$ in F $[]$ G

Architektúry a zobrazenia

Návrh programu má dve časti:

UNITY program a dôkaz jeho správnosti
(v tejto fáze nemožno hovoriť o zložitosti)

popis architektúr, implementovanie programu pre túto architektúru
(potom už možno riešiť aj otázky zložitosti)

Asynchrónne-Shared-Memory architektúry

- ASM:
  - pevne daná množina procesorov a pamätí
  - s každou pamäťou je asociovaná množina procesov, ktoré z nej môžu čítať a množina procesov, ktoré do nej môžu zapisovať
- Zobrazenie:
  - priradí každý príkaz nejakému procesu
  - priradí (alokuje) každú premennú pamäti
  - špecifikuje "control flow" pre každý proces (postupnosť udávajúca ako sú príkazy v danom procesore vykonávané)
- Celé to musí spĺňať:
  - všetky premenné na ľavej strane každého príkazu alokovaného k procesoru sú v pamätiach, do ktorých môže tento zapisovať (okrem indexov polí) a každá premenná na pravej strane (a všetky indexy na ľavej strane) sú v pamätiach, ktoré môžu čítať
  - control flow musí byť taký, že každý príkaz alokovaný ku procesoru je vykonaný nekonečne veľa krát

Distribuovaná architektúra

- Pozostáva z:
  - pevnej množiny procesorov a kanálov
  - lokálnych pamätí pre každý procesor
- kanály:
  - error-free
  - prenášajú správy v rovnakom poradí ako boli poslané
  - pre každý kanál je 1 procesor, ktorý do neho posiela a 1 procesor, ktorý z neho prijíma
  - každý kanál má svoj buffer
  - procesor môže niečo poslať, ak buffer nie je plný a prijať, ak nie je prázdny
- Zobrazenie programov je rovnaké ako ASM

Distribuovaná architektúra

- Premenné: sú alokované buď do (lokálnych) pamätí alebo ku kanálom
- Musí to spĺňať:
  - najviac 1 premenná je alokovaná ku kanálu a jej typ je postupnosť
  - ak nie je plná, potom zapisovateľ pridá správu do postupnosti na koniec
  - ak nie je prázdna, čitateľ odoberie prvý člen postupnosti

Synchrónne architektúry

<u>Zobrazenia na Synchrónne architektúry</u>

- podobne ako ASM
- naviac procesory majú spoločné hodiny; v každom kroku každý procesor vykoná inštrukciu
  - viacero procesorov môže zapisovať do pamäte naraz, ak píšu to isté
  - viacerí môžu čítať
  - nemožno naraz čítať aj písať
- Príklad: $z := x + 1 \mid\mid y := x + 2$
  - 1. podkrok: každý procesor načíta x
  - 2. podkrok: jeden vypočíta $x + 1$, druhý $x + 2$
  - 3. podkrok: prvý zapíše z, druhý zapíše y

Synchrónne architektúry

Existuje viacero možných zobrazení, my sa obmedzíme na nasledujúce:

- presne jeden statement sa vykonáva v čase (bez ohľadu na počet procesorov)

Príklad: $z := x + 1 \mid\mid y := x + 2$ sa môže vykonať na 3 procesoroch takto:

- A vypočíta $x + 1$
- B vypočíta $x + 2$
- C odpočíva

Synchrónne architektúry

Musí to spĺňať:

- popis aké operácie v každom príkaze majú byť vykonané procesormi
- alokácia premenných
- špecifikácia jedného “control flow” pre všetky procesory
- konzistencia alokácií premenných (tak ako predtým)

Príklad:

- nech *op* je asociatívna operácia
- majme statement s $\equiv \langle op\ j\colon 1 \leq j \leq N :: x[j] \rangle$
- s môže byť vykonané v čase $O(\log N)$ na N procesoroch
- s môže byť vykonané v čase $O(N/K + \log(K))$ na K procesoroch (rozdelí sa do N/K skupín po K prvkov)

Výpočet maxima 1

<u>Úloha:</u> určiť $m = \langle \max j: 0 \leq j < N:: A[j] \rangle$ pre dané pole $A[0..N-1]$

<u>Sekvenčná architektúra (SA):</u>

- invariant $m \leq \langle \max j: 0 \leq j < N:: A[j] \rangle$
- $FP \equiv m \geq \langle \max j: 0 \leq j < N:: A[j] \rangle$
- FP možno písať aj takto:

$FP \equiv \langle \wedge\, j: 0 \leq j < N:: m \geq A[j] \rangle$

$FP \equiv \langle \wedge\, j: 0 \leq j < N:: m = \max(m, A[j]) \rangle$

Program Maximum1

initially $m = -\infty$

assign $\langle \square \quad j: 0 \leq j < N:: m := \max(m, A[j]) \rangle$

end {Maximum1}

Výpočet maxima 2

Každé z priradení sa môže *napríklad* vykonať viac ráz po sebe, čo nie je efektívne; dve stratégie, ako sa takejto neefektívnosti vyhnúť:

1. možnosť:

   initially $j = 0$

   $m, j := \max(m, A[j]), j + 1 \text{ if } j < N$

2. možnosť:

   initially $\langle \,||j: 0 \leq j < N:: e[j] = \text{true}\rangle$

   $m, e[j] := \max(m, A[j]), \text{false if } e[j]$

Výpočet maxima 3

<u>Paralelná architektúra (PA):</u>

- prvky A dáme ako listy binárneho stromu
- každý vnútorný vrchol má hodnotu maxima jeho synov
- koreň má maximum
- strom máme v poli $X[1..(2N-1)]$
- j-ty vrchol má synov $2j$ a $2j+1$
- initially $X[N..(2N-1)] = A[0..N-1]$

Program Maximum2
declare X: array $[1..(2N-1)]$ of integer
always
  $\langle \; || j: 0 \leq j < N:: X[N+j] = A[j] \; \rangle$
  $\square\langle \; \square\, j: 1 \leq j < N:: X[j] = \max(X[2j], X[2j+1]) \; \rangle$
end {Maximum2}

ľahko vidno, že $X[1] = \langle \; \max j: 0 \leq j < N:: A[j] \; \rangle$

Výpočet maxima 4

Ideme ušetriť pamäť: nech N = 2M

Program Maximum3
    assign $\langle \; || j: 0 \leq j < M:: A[j] = \max(A[2j], A[2j+1]) \rangle$
end {Maximum3}

- A[0] = maximum, v čase $O(\log N)$
- použijeme pomocnú premennú t
  - na začiatku t = N
  - v každom kroku jej hodnota bude $t := \lceil t/2 \rceil$
- nech $A^0[j]$ sú počiatočné hodnoty v $A[j]$
- invariant:

$\langle \max j: 0 \leq j < t:: A[j] \rangle = \langle \max j: 0 \leq j < N:: A^0[j] \rangle$

- raz určite t = 1 a potom $A[0] = \langle \max j: 0 \leq j < N:: A^0[j] \rangle$

Porovnanie dvoch neklesajúcich postupností 1

dané dve postupnosti f, g

f: $\langle \wedge i\colon 0 \leq i < N :: f[i] \leq f[i + 1] \rangle$

g: $\langle \wedge i\colon 0 \leq i < N :: g[i] \leq g[i + 1] \rangle$

Úloha: zistiť, či $\{ f[i] \mid 0 \leq i \leq N \} = \{ g[i] \mid 0 \leq i \leq N \}$

predpokladáme, že $f[0] = g[0]$, $f[N] = g[N]$ a tiež že $f[N]$ resp. $g[N]$ sú väčšie ako ostatné členy (dá sa to zaručiť tak, že položíme $f[0] = g[0] = -\infty$ a $f[N] = g[N] = \infty$)

invariant I

$0 \leq u \leq N \wedge 0 \leq v \leq N \wedge$

$\wedge \{ f[i] \mid 0 \leq i \leq u \} = \{ g[i] \mid 0 \leq i \leq v \}$

Porovnanie dvoch neklesajúcich postupností 2

Program Compare

declare u, v: int

initially u, v = 0, 0

assign

$u := u + 1$ if $u < N \wedge f[u] = f[u + 1]$

$\square$ $v := v + 1$ if $v < N \wedge g[v] = g[v + 1]$

$\square$ $u, v := u + 1, v + 1$ if $u < N \wedge v < N \wedge f[u + 1] = g[v + 1]$

end{Compare}

$$
\begin{aligned}
FP \equiv \quad & (u \geq N \vee f[u] \neq f[u + 1]) \wedge \\
& (v \geq N \vee g[v] \neq g[v + 1]) \wedge \\
& (u \geq N \vee v \geq N \vee f[u + 1] \neq g[v + 1])
\end{aligned}
$$

Porovnanie dvoch neklesajúcich postupností 3

Z invariantu I možno odvodiť invariant

$u = N \equiv v = N$

To použijeme na zjednodušenie FP

- Máme teda 2 možnosti:

  1. $u = N \wedge v = N$: z invariantu I f a g majú rovnakú množinu prvkov
  2. $u < N \wedge v < N$

$$\mathrm{FP} \Rightarrow (f[u] \neq f[u + 1]) \wedge$$
$$\wedge\ (g[v] \neq g[v + 1]) \wedge$$
$$\wedge\ (f[u + 1] \neq g[v + 1])$$

Porovnanie dvoch neklesajúcich postupností 4

Predpokladajme $f[u + 1] < g[v + 1]$; potom

$g[v] = f[u]$ /* z I */

$f[u] < f[u + 1]$ /* z FP */

$g[v] < f[u + 1] < g[v + 1]$, teda $f[u + 1]$ nie je medzi prvkami $g[\mathrm{I}]$, lebo g je neklesajúca

$\{ f[i] \mid 0 \le i \le \mathrm{N} \} \ne \{ g[i] \mid 0 \le i \le \mathrm{N} \}$

teda f a g nemajú rovnakú množinu prvkov

- $\mathrm{FP} \wedge \mathrm{I} \Rightarrow$

  $[u = \mathrm{N} \equiv v = \mathrm{N}] \wedge$

  $[u = \mathrm{N} \equiv \{ f[i] \mid 0 \le i \le \mathrm{N} \} = \{ g[i] \mid 0 \le i \le \mathrm{N} \}]$
- dá sa ukázať, že FP sa dosiahne

Porovnanie dvoch neklesajúcich postupností 5

Paralelná architektúra (PA):

- $f[u] \in \{ g[i] \mid 0 \le i \le N \} \equiv$
  $\langle \wedge\, v: 0 \le v < N :: \neg(g[v] < f[u] < g[v+1])\rangle$
- musíme teda počítať predikát
  $\langle \wedge\, u, v : 0 \le u < N, 0 \le v < N :: \neg(g[v] < f[u] < g[v+1])\rangle \wedge$
  $\langle \wedge\, u, v : 0 \le u < N, 0 \le v < N :: \neg(f[u] < g[v] < f[u+1])\rangle$
- zjednodušením tohto predikátu dostaneme
  $\langle \wedge\, u, v : 0 \le u, v < N ::$
  $f[u] = g[v] \vee$
  $g[v] \ge f[u+1] \vee$
  $f[u] \ge g[v+1] \,\rangle$
- $O(N^2)$ konjunkcií a každá je disjunkciou 3 častí:
  čas $O(\log N)$ na $O(N^2)$ synchrónnych procesoroch

Dosiahnuteľnosť v orientovanom grafe

Daný graf $G = (V, E)$

počiatočný vrchol: *init*

Dosiahnuteľnosť:

1. vrchol *init* je dosiahnuteľný
2. ak *u* je dosiahnuteľný a (*u*, *v*) je hrana t.j. (u,v) $\in E$, tak aj *v* je dosiahnuteľný
3. žiaden iný nie je dosiahnuteľný

Úloha: vypočítať r tak aby:

- $r[v]$ je true, ak vrchol *v* je dosiahnuteľný:

  $\langle \wedge\ v\colon v \in V :: r[v] \equiv$ vrchol v je dosiahnuteľný $\rangle$
- invariant

  $\langle \wedge\ v\colon v \in V :: r[v] \Rightarrow v$ je dosiahnuteľný $\rangle \wedge r[init]$

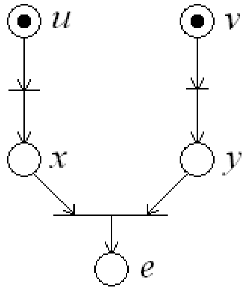
Dosiahnuteľnosť v orientovanom grafe

- FP $\equiv \langle \wedge\, u, v\text{: } (u, v) \in E :: r[u] \Rightarrow r[v] \rangle$
  FP $\equiv \langle \wedge\, u, v\text{: } (u, v) \in E :: r[v] = (r[u] \vee r[v]) \rangle$
  kedže $r[v] = (r[u] \vee r[v]) \equiv (r[u] \Rightarrow r[v])$
- platnosť invariantu zaručíme, ak na začiatku bude $r[v]$ = false pre $v \neq init$ *a* $r[init]$ = true

- Program Reach
  declare r: array[V] of boolean
  initially $\langle \Box\, v\text{: } v \in V :: r[v] = (v = init) \rangle$
  assign $\langle \Box\, u, v\text{: } (u, v) \in E :: r[v] := r[u] \vee r[v] \rangle$
  end{Reach}

- Ľahko možno ukázať, že FP sa dosiahne. Ako metriku môžeme použiť: M = $|\{\, v \mid \neg r[v] \,\}|$

Simulácia Petriho sietí



- Program PetriNet
  initially $u, v, x, y, e = 1, 1, 0, 0, 0$
  assign
  $\quad u, x := u - 1, x + 1$ if $u > 0$
  $\square\ v, y := v - 1, y + 1$ if $v > 0$
  $\square\ x, y, e := x - 1, y - 1, e + 1$ if $x > 0 \wedge y > 0$
  end{PetriNet}

Najkratšia cesta 1

- orientovaný graf: dvojica $G = (V, E)$, kde $E \subseteq \mathrm{V} \times \mathrm{V}$
- hrana z vrcholu i do j: dvojica $(i, j) \in E$
- cesta: postupnosť na seba nadväzujúcich hrán
- cyklus: cesta z vrcholu i do i
- váhovaný graf: graf, v ktorom každá cesta má celočíselnú váhu (tiež ohodnotený graf)
- dĺžka cesty: suma váh všetkých hrán na ceste
- váhová matica: (váhovaného grafu s N vrcholmi) je matica W typu N $\times$ N, v ktorej

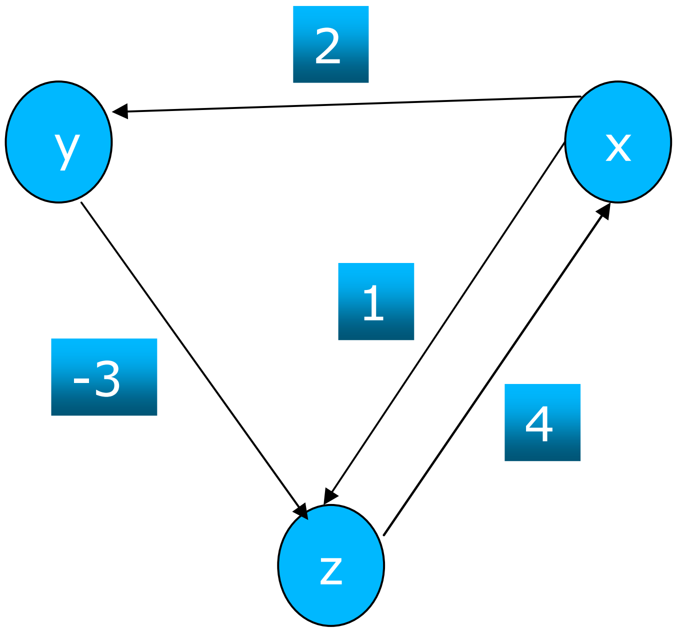
  $\mathrm{W}[i, j] =$
  - váha hrany (i, j), ak $(i, j) \in E$
  - 0, ak $i = j$
  - ∞, ak $i \neq j \wedge (i, j) \notin E$
- prázdna cesta: z vrcholu i do j vždy existuje; jej dĺžka: ∞

Najkratšia cesta 2

Problém najkratších ciest:

- daný je váhovaný graf *G* a jeho váhová matica
- *G* neobsahuje cykly s negatívnou dĺžkou
- vypočítať maticu D typu N $\times$ N, v ktorej D[*i*, *j*] je dĺžka najkratšej cesty z vrcholu *i* do *j*
- hodnotu D[*i*, *j*] budeme nazývať *vzdialenosť* *i*, *j*
- ak neexistuje cesta z vrcholu *i* do *j*, tak D[*i*, *j*] = ∞

Najkratšia cesta - Príklad



Najkratšia cesta 3

- <u>Špecifikácia:</u> program má počítať maticu d tak, že
  invariant FP $\Rightarrow$ (d = D)
  true $\rightarrow$ FP

- <u>Stratégia neformálne:</u>
  - nech $d[i, j]$ je dĺžka *nejakej* cesty z vrcholu i do j
  - ak nájdeme cestu kratšej dĺžky l (čiže $l < d[i, j]$), tak potom $d[i, j] := l$
  - konkrétne ak $\exists k$ také, že $d[i, k] + d[k, j] < d[i, j]$, tak cesta cez vrchol k je kratšia
  - $d[i, j] := \min \{ d[i, j], d[i, k] + d[k, j] \mid 0 \leq k < N\}$
  - nešpecifikujeme
    - ako vyberáme i, j, k
    - kedy sa operácie vykonávajú
    - ktoré procesory ich vykonávajú

Najkratšia cesta 4

Stratégia, formálny popis:

na začiatku $d[i, j] := W[i, j]$

$d[i, j]$ sa v priebehu výpočtu nebude zvyšovať, teda nemôže prekročiť $W[i, j]$

invariant:

$d[i, j]$ je dĺžka nejakej cesty z vrcholu i do j $\wedge$ $d[i, j] \leq W[i, j]$ (1)

$\mathrm{FP} \equiv \langle \wedge i, j, k{::}\ d[i, j] := \min(d[i, j], d[i, k] + d[k, j]) \rangle$ (2)

Najkratšia cesta 5

Aby sme zabezpečili, že vždy dosiahneme FP, ukážeme, že ak nejaký stav nie je FP, tak aspoň jedno $d[i, j]$ klesne.

Metrika suma $\forall d[i, j]$ sa nedá použiť keďže niektoré hodnoty môžu byť ∞,

Použijeme dvojice (*num*, *sum*) v lexikografickom usporiadaní, kde

- *num* = počet (i, j) tak, že $d[i, j] = \infty$
- *sum* = $\langle +i, j\text{: } d[i, j]$ je konečné $:: d[i, j]\rangle$

Metrika je ohraničená zdola, keďže nemáme cykly so zápornou dĺžkou, žiadna váha nie je $-\infty$

Progress condition: metrika klesá, ak stav nie je FP:

$$\neg\mathrm{FP} \wedge (num, sum) = (m, n) \rightarrow (num, sum) < (m, n) \qquad (3)$$

Najkratšia cesta 6

Dôkaz správnosti stratégie:

- metrika je ohraničená zdola a klesá, ak stav nie je FP $\Rightarrow$ FP sa nakoniec dosiahne
- v každom FP invariant platí, stačí teda ukázať, že matica, ktorá spĺňa (1) a (2) je riešením problému
- nemáme cykly zápornej dĺžky $\Rightarrow$ $\forall i, j\ \exists$najkratšia cesta s najviac N–1 hranami
- uvažujeme $\forall(x, y)$ také, že najkratšia cesta z x do y má najviac m hrán. Indukciou podľa m dokážeme, že vzdialenosť x, y je $d[x, y]$:

1. $m = 1$: triviálne $d[x, y] = W[x, y]$
2. nech tvrdenie platí pre m a nech najkratšia cesta z x do y má $m+1$ hrán. Predposledný vrchol tejto cesty nech je z, potom
   - cesta x, z má m hrán
   - cesta z, y má jednu hranu

podľa indukčného predpokladu

- $d[x, y] \leq d[x, z] + d[z, y]$

Najkratšia cesta 7

Formálny popis programu:

Program P1

initially $\langle \; || \; i, j :: d[i, j] = \mathrm{W}[i, j] \; \rangle$

assign $\langle \; \square i, j, k :: d[i, j] := \min(d[i, j], d[i, k] + d[k, j]) \; \rangle$

end{P1}

program má N^3 priradení

Najkratšia cesta - sekvenčná architektúra

Uvažujme cesty z i do j, v ktorých indexy vrcholov, čo sa na týchto cestách nachádzajú, sú menšie než k okrem koncových vrcholov i, j;

Nech H[i, j, k] je minimálna dĺžka takýchto ciest

Veta:

$$\langle \wedge i, j :: H[i, j, 0] = W[i, j] \rangle \wedge$$
$$\langle \wedge i, j, k :: H[i, j, k+1] = \min(H[i, j, k], H[i, k, k] + H[k, j, k]) \rangle \quad (4)$$

Dôkaz. Uvažujme cestu, ktorá dosahuje minimum H[i, j, k +1]

- ak k nie je v tejto ceste, tak H[i, j, k +1] = H[i, j, k]
- ak k je v tejto ceste, tak H[i, j, k +1] = H[i, k, k] + H[k, j, k]

Podľa definície d[i, j] = H[i, j, N] (vrcholy sú 0..N–1)

Najkratšia cesta - sekvenčná architektúra

Veta: Množina rovností (4) je “proper”.

Dôkaz.

- každé H[*i*, *j*, *k*] je na ľavej strane práve raz
- rovnice usporiadame tak, že *k* bude neklesajúce

Program P2

declare H: array[0..N–1, 0..N–1, 0..N]

always

⟨ || *i*, *j* :: H[*i*, *j*, 0] = W[*i*, *j*] ⟩

▯ ⟨ ▯*k* :: ⟨ *i*, *j* :: H[*i*, *j*, *k* +1] =

min(H[*i*, *j*, *k*], H[*i*, *k*, *k*] + H[*k*, *j*, *k*]) ⟩⟩

▯ ⟨ || *i*, *j* :: *d*[*i*, *j*] = H[*i*, *j*, N] ⟩

end

- $O(N^3)$ rovností
- $O(N^3)$ pamäte a času

Najkratšia cesta - sekvenčná architektúra

Program s explicitnou sekvencializáciou:

PASCAL-like program PP (sekvencializácia P1)

```
Progarm PP
for x := 0 to N–1 do
 for u := 0 to N–1 do
   for v := 0 to N–1 do
      d[u,v] := min(d[u,v], d[u,x] + d[x,v])
```

Najkratšia cesta - sekvenčná architektúra

Indexy v PP sú modifikované nasledujúcim spôsobom:
`(x,u,v):=(x,u,v)+1`, kde `(x,u,v)` je trojmiestne číslo v N-árnej sústave

Program P3 {Floyd–Warshall}

declare x, u, v: int

initially $\langle$ || i, j :: $d[i, j] = W[i, j]$ $\rangle$ || $x, u, v = 0, 0, 0$

assign

$d[u, v] := \min(d[u, v], d[u, x] + d[x, v])$

|| $(x, u, v) := (x, u, v) + 1$

if $(x, u, v) \neq (N - 1, N - 1, N - 1)$

end{P3}

Najkratšia cesta - paralelná synchrónna architektúra

O(N) krokov s O(N^2) procesormi

transformujeme program tak, že vyčleníme priradenia, ktoré sa môžu vykonať naraz; (spolu je priradení N^3), rozdelíme ich na N skupín po N^2 priradení.

Program P1′

initially $\langle \; || \; i, j :: d[i, j] = W[i, j] \; \rangle$

assign

$\langle \; \square \; k :: \langle \; || i, j :: d[i, j] := \min(d[i, j], d[i, k] + d[k, j]) \; \rangle$

end{P1′}

Najkratšia cesta - paralelná synchrónna architektúra

Uvažujeme, že máme architektúru s N^2 procesormi a určíme poradie vykonávania priradení

Program {parallel Floyd–Warshall} P4

declare k: int

initially $\langle \,||\, i, j :: d[i, j] = W[i, j] \rangle \,||\, k = 0$

assign

$\langle \,||i, j :: d[i, j] := \min(d[i, j], d[i, k] + d[k, j])\rangle$ if $k < N$

$||\ k := k + 1$ if $k < N$

end{P4}

program P4: potrebuje $O(N)$ krokov

Najkratšia cesta - paralelná synchrónna architektúra

$O(\log^2 N)$ krokov s $O(N^3)$ procesormi

P4 nie je vhodný, ak máme N^3 procesorov, pretože ich nevyužjeme všetky

Program P5

initially $\langle \; || \; i, j :: d[i, j] = W[i, j] \; \rangle$

assign $\langle \; || i, j :: d[i, j] := \langle \; \min k :: d[i, k] + d[k, j] \; \rangle\rangle$

end{P5}

program pozostáva z jediného príkazu, ktorý priradí N^2 hodnôt

po m-tom vykonaní príkazu platí nasledovný invariant:

$d[i, j]$ je dĺžka najkratšej cesty z i do j s najviac 2^m-1 vrcholmi medzi i a j

Najkratšia cesta - paralelná synchrónna architektúra

FP sa dosiahne po $O(\log N)$ vykonaniach príkazu

- na jedno vykonanie treba čas $O(\log N)$; ide o nájdenie minima a výpočet $d[i, k] + d[k, j]$
  - $d[i, k] + d[k, j]$ môže byť vypočítané na N^3 procesoroch v konštantnom čase
  - pre dané i, j minimum cez k môže byť vypočítané v čase $O(\log N)$ pomocou $O(N)$ procesorov
- teda každý krok možno urobiť v čase $O(\log N)$ a s $O(N^3)$ procesormi
- keďže výpočet má $O(\log N)$ krokov, tak program P5 vypočíta D v čase $O(\log^2 N)$ a s $O(N^3)$ procesormi

Najkratšia cesta – asynchrónna architektúra

P2 môže byť alokovaný do ASV s N^2 procesormi tak, že každý procesor počíta $H[i, j, k]$ pre dané i, j

využijeme nasledujúci vlastnosť:

$$H[i, j, k+1] \leq H[i, j, k]$$

ako dôsledok: môžeme použiť $H[i, j, k+m]$ namiesto $H[i, j, k]$ pre $m \geq 0$

Formulujme nasledujúci invariant:

$$d[i, j] = H[i, j, k]$$
$$\wedge\ d[i, j] \text{ je dĺžka nejakej cesty z } i \text{ do } j$$
$$\wedge\ k \leq N$$

Najkratšia cesta – asynchrónna architektúra

v P4 je synchrónnosť podstatná – jedno k sa používa pre výpočet každého $d[i, j]$. Teraz predpokladáme asynchrónne riešenie, v ktorom $d[i, j]$ sa počíta (i, j)-tym procesorom a k sa nahradí lokálnou premennou $k[i, j]$

zoslabením predošlého invariantu dostaneme invariant

$d[i, j] \leq \mathrm{H}[i, j, k[i, j]]$

$\wedge$ $d[i, j]$ je dĺžka nejakej cesty z i do j

$\wedge$ $k[i, j] \leq \mathrm{N}$

Najkratšia cesta – asynchrónna architektúra

základom pre nasledujúci program je toto pozorovanie (pre zjednodušenie r znamená $k[i, j]$):

Z predchádzajúceho invariantu vyplýva:

Teoréma. $(k[i, r] \geq r \wedge k[r, j] \geq r) \Rightarrow$

$$\min(d[i, j], d[i, r] + d[r, j]) \leq H[i, j, r + 1]$$

Dôkaz. Vieme, že

$H[i, j, k + 1] = \min(H[i, j, k], H[i, k, k] + H[k, j, k])$

musíme teda ukázať, že $(k[i, r] \geq r \wedge k[r, j] \geq r) \Rightarrow$

$\min(d[i, j], d[i, r] + d[r, j]) \leq \min(H[i, j, r], H[i, r, r] + H[r, j, r])$;

nasledovne:

$d[i, j] \leq H[i, j, r]$ /* z invariantu */

$d[i, r] \leq H[i, r, k[i, r]]$ /* z invariantu */

$H[i, r, k[i, r]] \leq H[i, r, r]$ /* z predpokladu */

$d[i, r] \leq H[i, r, r]$ /* z predošlých dvoch */podobne

ukážeme, že $d[r, j] \leq H[i, r, r]$

Najkratšia cesta – asynchrónna architektúra

Správnosť nasledujúceho programu vyplýva z invariantu a z nasledujúceho:

$$FP \Rightarrow \langle \wedge\ i, j :: k[i, j] \geq N \rangle$$

Pevný bod sa dosiahne, keďže metrika $\langle +\ i, j :: k[i, j] \rangle$ sa zvýši s každou zmenou stavu a je ohraničená zhora.

Program P6

declare k: array[0..N–1, 0..N–1] of integer

initially $\langle \,||\, i, j :: d[i, j], k[i, j] = W[i, j], 0 \rangle$

assign {r znamená pre zjednodušenie $k[i, j]$}

$\langle \square\ i, j :: d[i, j], r := \min(d[i, j], d[i, r] + d[r, j]), r + 1$

if $r < N \wedge k[i, r] \geq r \wedge k[r, j] \geq r \rangle$

end{P6}

Readers–Writers Problem 1

<u>Zadanie:</u>

- daný program (user), v ktorom sú dve premenné *nr*, *nw* také, že pre nejakú konštantu N platí

  invariant $0 \leq nr \leq N \wedge 0 \leq nw \leq N$ (1)

  initially $nr = nw = 0$

  hodnoty *nr*, *nw* sa menia len v nasledujúcich druhoch priradení v programe user:

  $nr := nr + 1$ {started read}

  $nr := nr - 1$ {end read}

  $nw := nw + 1$ {started write}

  $nw := nw - 1$ {end write}

  (v týchto priradeniach môžu byť aj iné premenné) a program user môže mať aj iné priradenia

Readers–Writers Problem 2

<u>Úloha:</u> modifikovať priradenia programu user tak, aby platilo:

invariant $nw \leq 1 \wedge (nr = 0 \vee nw = 0)$ (2)

- *nr*, *nw* znamenajú počet čítajúcich a zapisujúcich procesov (do súboru) v danom čase
- počet procesorov je obmedzený na N (invariant 1)
- hocikoľko veľa môže čítať, najviac jeden može zapisovať a zápis sa nemôže vykonať súčasne s iným zápisom alebo čítaním

Readers–Writers Problem 3

<u>Riešenie:</u>

ak $0 \leq nr \leq \mathrm{N} \wedge 0 \leq nw \leq \mathrm{N}$, tak

$[nw \leq 1 \wedge (nr = 0 \vee nw = 0)] \equiv [(nr + \mathrm{N} \times nw) \leq \mathrm{N}]$

nech $t = \mathrm{N} - (nr + \mathrm{N} \times nw)$

máme nový invariant $t \geq 0$ (3)

modifikovaný program user:

declare t: int

always $t = \mathrm{N} - (nr + \mathrm{N} \times nw)$

assign

	$nr := nr + 1$ if $t \geq 1$	{started read}
$\Box$	$nr := nr - 1$	{end read}
$\Box$	$nw := nw + 1$ if $t \geq \mathrm{N}$	{started write}
$\Box$	$nw := nw - 1$	{end write}

Readers–Writers Problem 4

- Toto riešenie nezaručuje „progress" ani pre čítajúcich ani pre zapisujúcich. Nemáme požiadavku, že čítanie alebo zápis niekedy raz skončí, a teda nemôžeme tvrdiť, že iné čítanie alebo zápis začnú.
- *nr* a *nw* nám neumožňujú formulovať tvrdenia o jednotlivých čítaniach a zápisoch
- Predpokladajme, že všetky čítania skončia v konečnom čase. Modifikujeme riešenie tak, že potom raz určite (*eventually*) dôjde aj na zápis.
- predpokladajme

$$nr = k \;\wedge\; k > 0 \longrightarrow nr \neq k \qquad (4)$$

Readers–Writers Problem 5

- nech nq je počet procesov čakajúcich na zápis
  invariant $nq \geq 0$

- Úloha: modifikovať program user tak, aby boli zachované predchádzajúce invarianty a naviac

  invariant $nq > 0 \rightarrow nw = 1$ (5)

- jednoduchá reštriktívna stratégia: zabrániť čítaniu, ak niekto čaká na zápis; má však zložitú distribuovanú implementáciu, musel by totiž poslať požiadavku na zápis všetkým čakateľom

Readers–Writers Problem 6

- menej prísna stratégia: ak *nq* $>$ 0, tak sa raz (*eventually*) zabráni čítaniu (started read)

- použijeme novú premennú *b*: boolean

  1. *b* $\Rightarrow$ niekto čaká na zápis

  2. ak niekto chce písať, tak začne alebo *b* bude platiť

  3. ak *b* platí, tak *b* ostáva platné, až kým sa napokon raz začne zapisovať

  4. ak *b* platí, nemožno čítať

- v nasledujúcom programe nie sú uvedené priradenia zvyšujúce *nq*, keďže tieto sa nemodifikujú

Readers–Writers Problem 7

Modifikácia programu user

declare t: integer, b: boolean

always $t = N - (nr + N.nw)$

initially b = false

assign

	$nr := nr + 1$ if $t \geq 1 \wedge \neg b$	{started read}
[]	$nr := nr - 1$	{end read}
[]	$nw, nq, b := nw + 1, nq - 1,$ false if $t \geq N \wedge nq > 0$	{started write}
[]	$nw := nw - 1$	{end write}
[]	$b := nq > 0$	{set b}

Readers–Writers Problem 8

Dôkaz (5), t.j ($nq > 0 \longrightarrow nw = 1$) plynie z (4) a (6)–(9):

$b \Rightarrow nq > 0$	(6) z programu
$nq > 0$ ensures $nw = 1 \vee b$	(7) z programu
$b \wedge nr = 0$ ensures $nw = 1$	(8) z programu
$b \wedge nr = k \wedge k > 0$ unless $b \wedge nr < k$	(9) z programu

(už sme neformálne urobili predtým)

- chceme ukázať, že $nq > 0 \longrightarrow nw = 1$:

  $b \wedge nr = k \wedge k > 0 \longrightarrow b \wedge nr < k$ (PSP 4, 9)

- aplikujeme indukciu

  $b \longrightarrow nr = 0 \wedge b$ (10)

- tranzitivita s (8) a (10): $b \longrightarrow nw = 1$ (11)
- kombináciou (7) a (11) dostávame výsledok.

Večerajúci filozofi 1

- Motivácia, Obrázok
- daný je súvislý, konečný, neorientovaný graf *G* bez slučiek
- jeho vrcholy reprezentujú procesy (filozofi)
- (u, v) je označenie hrany medzi u a v
- boolovská matica incidencie E

  $E[u, v] = E[v, u], \neg E[u, u]$
- pre $\forall u$ máme premennú u.dine s hodnotami t, h, e (u.dine má práve jednu z týchto hodnôt)
- označenie:

  $u.t \equiv (u.\text{dine} = t)$ *thinking*

  $u.h \equiv (u.\text{dine} = h)$ *hungry*

  $u.e \equiv (u.\text{dine} = e)$ *eating*

Večerajúci filozofi 2

možná zmena stavu premennej *u*.dine je len:

$$t \rightarrow h \rightarrow e \rightarrow t \rightarrow \ldots$$

realizovanie zmien stavu premennej *u*.dine:

program user: $t \rightarrow h$, $e \rightarrow t$

program os: $h \rightarrow e$

Úloha: navrhnúť program os tak, aby os ☐ user (user je daný) zabezpečil, že:

- susedia nejedia naraz
- hladný raz určite (*eventually)* bude jesť (ak žiadny proces neje večne, tak na každého hladného určite raz dôjde)

Večerajúci filozofi 3

<u>Špecifikácia pre user:</u>

- $u.t$ unless $u.h$ in user (udn1)
- stable $u.h$ in user (udn2)
- $u.e$ unless $u.t$ in user (udn3)
- podmienená vlastnosť pre user: (udn4)

  $\langle \forall (u, v) :: \neg(u.e \wedge v.e) \rangle$ /

  $\langle \forall u :: u.e \rightarrow \neg u.e \rangle$

<u>Špecifikácia pre user □ os:</u>

- invariant $\neg(u.e \wedge v.e \wedge E[u, v])$ in user □ os (dn1)
- $u.h \rightarrow u.e$ in user □ os (dn2)

Večerajúci filozofi 4

Obmedzenia pre os:

- constant $u.t$ in os (odn1)
- stable $u.e$ in os (odn2)

t.j. v os nie sú prechody z *eating* a prechody z a do *thinking*

Odvodená vlastnosť pre user:

- stable $\neg u.e$ in user

Odvodené vlastnosti pre os:

- stable $\neg u.h$ in os
- $u.h$ unless $u.e$ in os

Odvodené vlastnosti pre user □ os:

- $u.t$ unless $u.h$ in user □ os (dn3)
- $u.h$ unless $u.e$ in user □ os (dn4)
- $u.e$ unless $u.t$ in user □ os (dn5)

Večerajúci filozofi 5

Dôkaz odvodenej vlastnosti pre user (stable $\neg u.e$):

$u.t$ unless $u.h$ in user /* (udn1) */

$u.h$ unless false in user /* def. stable pre (udn2) */

$u.t \vee u.h$ unless false in user /* tranzitivita pre unless */

$u.t \vee u.h \equiv \neg u.e$ /* práve jedna z hodnôt */

$\neg u.e$ unless false in user $\equiv$ stable $\neg u.e$ in user

V ďalšom ideme špecifikovať vlastnosti os; teda všetko bude pre os, ak nebude výslovne uvedené inak.

Večerajúci filozofi: Spec1

Prvé priblíženie k riešeniu

<u>Spec1:</u>

(odn1), (odn2), (odn3), (odn4)

kde

- invariant $\neg(u.e \wedge v.e \wedge E[u, v])$ (odn3)
- (dn1), (dn3)–(dn5), $\langle \forall u :: u.e \rightarrow \neg u.e \rangle$ / (odn4)
  $\langle \forall u :: u.h \rightarrow u.e \rangle$

musíme ukázať, že:

- user □ os spĺňa (dn1), (dn2)
- os spĺňa obmedzenia (odn1), (odn2)

Večerajúci filozofi: Spec1

Dôkaz:

- (dn3) – (dn5) platí, lebo (odn1), (odn2) platí v os a (udn1) – (udn3) platí pre user
- $\neg(u.e \wedge v.e \wedge E[u, v])$ je stable in user lebo $\neg u.e$ aj $\neg v.e$ sú stable in user a $E[u, v]$ je constant
- (dn1) platí v user □ os z predchádzajúceho a z (odn3) v os
- $\langle \forall u :: u.e \rightarrow \neg u.e \rangle$ platí v user □ os lebo toto je dôsledok (udn4) a hypotézy (udn4) – (dn1) platí v user □ os
- $\langle \forall u :: u.h \rightarrow u.e \rangle$ platí v user □ os, lebo je to dôsledok (odn4) a predpoklad (odn4) platí
- (dn1) platí v user □ os z predchádzajúceho

Večerajúci filozofi: Spec2

- <u>Problém s implementovaním Spec1:</u>
  - hladný je (eats), ak nejedia susedia – spĺňa to (odn3)
  - problém je zabezpečiť (odn4), pretože jeden proces môže stále opakovať $t \rightarrow h \rightarrow e \rightarrow t \rightarrow h \rightarrow e \rightarrow \ldots$

Riešenie:

zavedieme asymetriu medzi procesmi – usporiadanie

hladný buď začne jesť alebo postúpi v usporiadaní nahor, až kým nie je navrchu

čiastočné usporiadanie dostaneme tak, že v pôvodnom grafe G orientujeme hrany (aby nevznikol cyklus), čím dostaneme nový graf G'

$u \rightarrow v$ iff u je vyššie (má vyššiu prioritu) ako v

orientácia je dynamická a chceme, aby platilo:

(a) G' je acyklický

(b) hladný proces nikdy neklesne zmenami orientácií

(c) hladný nakoniec vždy stúpa v usporiadaní, až dosiahne vrchol

Večerajúci filozofi: Spec2

Pravidlá pre zmenu orientácie G':

1. Hrana zmení orientáciu len keď odpovedajúci vrchol zmení stav z hladný na *eating*
2. Všetky hrany incidentné s *eating* smerujú k nemu – teda tento vrchol je nižšie v usporiadaní než susedia

Ľahko vidno, že pravidlá 1. a 2. zabezpečia platnosť (a), (b), (c).

$prior[u, v] = u$, ak u je vyššie než v v grafe G' (t.j. $u \to v$)

$prior[v, u] = prior[u, v]$

$u.\text{top} \equiv \langle \forall v\text{: } E[u, v] \wedge v.h\text{:: } prior[u, v] = u \rangle$

Večerajúci filozofi: Spec2

<u>Formálny popis: Spec2</u>

(odn1)–(odn3), (odn5)–(odn8), kde

invariant $u.e \wedge E[u, v] \Rightarrow prior[u, v] = v$ (odn5)

$(prior[u, v] = v)$ unless $v.e$ (odn6)

invariant G' je acyklický (odn7)

(dn1, dn3–dn5), (odn5)–(odn7), $\langle \forall u :: u.e \rightarrow \neg u.e \rangle$ / (odn8)

$\langle \forall u :: u.h \wedge u.\mathrm{top} \rightarrow \neg(u.h \wedge u.\mathrm{top}) \rangle$

čo znamená:

- *eating* má nižšiu prioritu než susedia (odn5)
- priorita sa mení, až keď začne jesť (odn6)

Večerajúci filozofi: Spec3

<u>Problém s implementovaním Spec2:</u>

Zo Spec2 by sme mohli odvodiť program s jednoduchým pravidlom:

> *hungry* proces *u* začne jesť, ak *u*.top a ak žiaden jeho sused neje. Ak začne jesť, tak *prior*[*u*, *v*] sa stane *v* pre všetkých susedov vrcholu *u*.

Toto je ale nevhodné pre distribuovanú architektúru:
hungry totiž musí najprv zistiť, či susedia nejedia

Večerajúci filozofi: Spec3

<u>Ďalšie zjemnenie - Spec3:</u>

jeho úlohou je nahradiť (odn3), teda invariant

$$\neg(u.e \wedge v.e \wedge E[u, v]),$$

vlastnosťou, ktorá sa dá implementovať v distribuovanej architektúre.

Zavedieme novú premennú *fork*[*u*, *v*], ktorá môže mať hodnotu *u* alebo *v*;

t.j. vidličku má len jeden zo susediacich vrcholov.

Proces môže jesť, ak má všetky vidličky.

Večerajúci filozofi: Spec3

Spec3:

(odn1), (odn2), (odn5) – (odn9), kde

invariant $u.e \wedge E[u, v] \Rightarrow fork[u, v] = u$ (odn9)

teda (odn9) nahradilo (odn3)

Veta: (odn9) $\Rightarrow$ (odn3)

Dôkaz:

$u.e \wedge v.e \wedge E[u, v] \Rightarrow u = v \wedge E[u, v]$ (z odn9)

ale $\neg(u = v \wedge E[u, v])$ keďže $\neg E[u, u]$, a teda

$\neg(u.e \wedge v.e \wedge E[u, v])$

Večerajúci filozofi: Spec4

<u>Ďalšie zlepšenie:</u>

Spec3 evokuje nasledovné riešenie:

- *non-eating* proces *u* pošle vidličku, ktorú zdieľa s *v* procesu *v*, ak má tento vyššiu prioritu než *u* alebo ak *u* je *thinking*
- *hungry* proces *u* je (eats), ak drží vidličky, ktoré zdieľa so susedmi a pre každého suseda *v* platí:
  - *u* má vyššiu prioritu ako *v*
  - *alebo v* je *thinking*

V distribuovanej architektúre máme opäť problém určiť priority

Večerajúci filozofi: Spec4

<u>Distribuovaná implementácia priorít:</u> nahradíme (odn5) a (odn6) tak, aby sa dali ľahko implementovať na distribuovanej architektúre.

Neformálne (tzv. „hygienické riešenie"):

- každej vidličke priradíme atribúty *clean*/*dirty*
- proces *u* má prioritu nad *v*, ak vidlička, ktorú zdieľajú, je
  - 1. *clean* a má ju *u*, alebo
  - 2. *dirty* a má ju *v*
- *eating* proces drží všetky vidličky (ktoré zdieľa so susedmi) a všetky sú *dirty* (odn10, zodpovedá odn5)
- proces držiaci *clean* vidličku ju stále drží a tá ostáva *clean*, až kým nezačne jesť (odn11)
- *dirty fork* ostáva *dirty*, až kým sa nepošle preč – vtedy sa aj vyčistí (odn12, zodpovedá odn6)
- *clean forks* držia len *hungry* (odn13)

Večerajúci filozofi: Spec4

nech $clean[u, v]$: boolean

$clean[u, v] = \text{true} \equiv$ vidlička od u a v je *clean*, inak je *dirty*

$prior[u, v] = u \equiv ((fork[u, v] = u) = clean[u, v])$

Spec4: (odn1), (odn2), (odn7), (odn8), (odn10)–(odn13), kde

invariant $u.e \wedge E[u, v] \Rightarrow fork[u, v] = u \wedge \neg clean[u, v]$ (odn10)

$fork[u, v] = v \wedge clean[u, v]$ unless $v.e$ (odn11)

$fork[u, v] = u \wedge \neg clean[u, v]$ unless

$fork[u, v] = v \wedge clean[u, v]$ (odn12)

$fork[u, v] = u \wedge clean[u, v] \Rightarrow u.h$ (odn13)

čiže

- namiesto (odn5), (odn9) máme (odn10)
- namiesto (odn6) máme (odn11), (odn12)

Večerajúci filozofi: Spec4

Motivácia pre zlepšenie: Spec4 evokuje nasledovné riešenie:

- *non-eating* proces *u* pošle vidličku hladnému (*hungry*) susedovi *v*, ak je táto vidlička *dirty*
- *hungry* proces *u* je (eats), ak drží všetky odpovedajúce vidličky a ak pre každého suseda *v* spoločná vidlička je *clean* alebo *v* je *thinking*; ak proces je (eats), zašpiní všetky vidličky

Problém: ako môže proces *u* zistiť, či sused *v* je hladný?

Večerajúci filozofi: Spec5

Neformálne: navrhneme mechanizmus, ako *v* informuje suseda *u*, že je *hungry*

- zavedieme *request-token* pre každdú dvojicu; má ju buď *u* alebo *v* (práve jeden zo susedov)
- ak *u* má aj vidličku aj *request-token* zdieľané s procesom *v*, tak *v* je hladný (odn14)

Večerajúci filozofi: Spec5

1. Poslanie *request-token* (odn15 a 16):
   proces *u* pošle *request-token* procesu *v*, ak
   1. *u* má *request-token*
   2. *u* nemá vidličku (*fork*)
   3. *u* je *hungry*

2. Poslanie vydličky (odn17 a 18):
   proces *u* pošle *fork* *v*, ak
   1. *u* má vidličku a *request-token*
   2. vidlička je špinavá
   3. *u* nie je *eating*

   keď sa *fork* pošle, tak sa zároveň vyčistí

Večerajúci filozofi: Spec5

3. Prechod z *hungry* do *eating* (odn19):

   hungry proces *u* je (eats), ak drží všetky *fork* a ak pre každého suseda *v* spoločná vidlička je *clean* alebo *u* nemá *request-token* zdieľaný s *v*; keď *u* je (eats), zašpiní všetky vidličky, čo má (odn10)

   ak *u*.top platí, t.j. *u* nemá hladného suseda s vyššou prioritou, tak pre každého suseda s vyššou prioritou *v* *u* drží vidličku, ale nie *request-token*

Večerajúci filozofi: Spec5

Formálne:

- pre každú dvojicu susedov zavedieme $rt[u, v]$ s hodnotami u a v (práve jedna z nich)
- skratka $u.mayeat$ značí „u môže jesť"
- podmienka, za ktorej u posiela *request-token* procesu v, je označená $sendreq[u, v]$
- podmienka, za ktorej u posiela vidličku procesu v, je označená $sendfork[u, v]$

u má *request-token* iff $rt[u, v] = u$

$u.mayeat \equiv \langle \forall v\colon E[u, v]::$

$\quad (fork[u, v] = u \wedge (clean[u, v] \vee rt[u, v] = v))\rangle$

$sendreq[u, v] \equiv (fork[u, v] = v) \wedge (rt[u, v] = u) \wedge u.h$

$sendfork[u, v] \equiv$

$\quad (fork[u, v] = u) \wedge \neg clean[u, v] \wedge (rt[u, v] = u) \wedge \neg u.e$

Večerajúci filozofi: Spec5

Spec5: (odn1), (odn2), (odn7), (odn10)–(odn19), kde

invariant $(fork[u, v] = u) \wedge (rt[u, v] = u) \Rightarrow v.h$	(odn14)
$rt[u, v] = u$ unless $sendreq[u, v]$	(odn15)
$sendreq[u, v]$ ensures $rt[u, v] = v$	(odn16)
$fork[u, v] = u$ unless $sendfork[u, v]$	(odn17)
$sendfork[u, v]$ ensures $fork[u, v] = v$	(odn18)
$(u.h \wedge u.\ mayeat)$ ensures $\neg(u.h \wedge u.mayeat)$	(odn19)

Večerajúci filozofi: Program pre os

Na začiatku predpokladáme, že:

všetci filozofi špekulujú (sú *thinking*)
všetky vidličky sú špinavé (*fork* sú *dirty*)

- *fork* a *request-token* sú na rôznych miestach
- *forks* sú tak, že graf *G′* je acyklický: napr. očíslujeme procesy a *fork* dostane proces s nižším číslom
- vyšší index = vyššia priorita

Večerajúci filozofi: Program pre os

Program os
always
    ⟨ □ *u*:: *u*.*mayeat* = ⟨∧*v*: *E*[*u*, *v*]::
            (*fork*[*u*, *v*] = *u* ∧ (*clean*[*u*, *v*] ∨ *rt*[*u*, *v*] = *v*))⟩ ⟩
    □ ⟨ □ *u*, *v*: *E*[*u*, *v*]::
      sendreq[*u*, *v*] = ((*fork*[*u*, *v*] = *v*) ∧ (*rt*[*u*, *v*] = *u*) ∧ *u*.*h*)
    □ *sendfork*[*u*, *v*] = ((*fork*[*u*, *v*] = *u*) ∧ ¬*clean*[*u*, *v*] ∧
                (*rt*[*u*, *v*] = *u*) ∧ ¬*u*.*e*) ⟩
initially
        ⟨ □ *u*:: *u*.dine = *t* ⟩
    □   ⟨ □(*u*, *v*):: *clean*[*u*, *v*] = false ⟩
    □   ⟨ □(*u*, *v*): *u* < *v*:: *fork*[*u*, *v*], *rt*[*u*, *v*] = *u*, *v* ⟩
assign
    ⟨ □ *u*:: *u*.dine := *e* if *u*.*h* ∧ *u*.*mayeat*
        || ⟨ || *v*: *clean*[*u*, *v*] := false if *u*.*h* ∧ *u*.*mayeat* ⟩ ⟩
    □ ⟨ □(*u*, *v*):: *rt*[*u*, *v*] := *v* if *sendreq*[*u*, *v*]
        □ *fork*[*u*, *v*], *clean*[*u*, *v*] := *v*, true if *sendfork*[*u*, *v*] ⟩
end

Koordinácia schôdzí 1

- Profesori sú členmi komisií
- Každá komisia má pevný nenulový počet členov
- Profesor sa môže rozhodnúť, že sa chce zúčastniť schôdze komisie (je mu jedno ktorej)
- Čaká, až kým schôdza komisie, ktorej je členom, nezačne

Koordinácia schôdzí 2

Schôdza:

1) Môže začať, len keď všetci jej členovia čakajú
2) Dve schôdze sa nemôžu uskutočniť súčasne, ak majú spoločných členov

Predpoklad:

Schôdze netrvajú večne

Koordinácia schôdzí 3

Úloha:

Napísať protokol, ktorý zaručí, že ak všetci členovia nejakej komisie sa chcú zúčastniť schôdze, tak aspoň jeden člen sa zúčastní nejakej schôdze

- u – profesor
- x, y, - komisie
- x.mem – množina členov komisie x, x.mem $\neq \emptyset$

Koordinácia schôdzí 4

- u.g = true ak u čaká
- x.co = true ak komisia x sa zišla na schôdzi a zasadá
- x,y sú susedné komisie ak
  $x.mem \cap y.mem \neq \emptyset$, E[x,y] bude označovať susednosť
- x.g = true ak všetci členovia x čakajú, t.j
  $x.g \equiv \langle \forall u\text{: } u \in x.mem :: u.g \rangle$
- $x^{*}.co \equiv x.co \vee \langle \exists y\text{: } E[x,y] :: y.co \rangle$

Koordinácia schôdzí 5

Daný je program prof

Ideme navrhnúť program coord(inator) tak aby program

Sync(hronizator) = prof $\square$ coord

mal nasledovné vlastnosti:

Špecifikácia sync

$\neg$ x.co unless x.g	sy1
$\neg$ (x.co $\wedge$ y.co $\wedge$ E[x,y])	sy2
x.g $\rightarrow$ x*.co	sy3

Koordinácia schôdzí 5

Špecifikácia prof

u.g unless $\langle \exists x: u \in x.mem :: x.co \rangle$ pr1

$\neg$x.co je stable pr2

(t.j. schôdzu nemôže začať prof)

Koordinácia schôdzí 6

$<\forall x: x.co \rightarrow \neg x.co>$ pr3

Odvodená vlastnosť pre prof

x.g unless x*.co pr4

Dôkaz: aplikácia pravidla pre jednoduchú konjunkciu na pr1

Koordinácia schôdzí 7

Obmedzenia pre coord:

Zdieľané premenné medzi prof a coord sú x.co a u.g cd1
u.g je constatnt cd2
x.co je stable cd3

Koordinácia schôdzí 8

prof <--- x.co, u.g ---> coord

cd1+cd2 – coord vie čítať ale nie meniť u.g

Poznámky:

- profesor môže čakať aj potom, čo schôdza začala
- Nič sa nehovorí o tom či sa a kedy profesor zúčastní/opustí schôdzu

Koordinácia schôdzí 9

Triviálne riešenie

Program coord1

always

< x :: x.g = <∀u: u ∈ x.mem :: u.g >

□ x*.co ≡ x.co ∨ <∃y: E[x,y] :: y.co >>

initially

< □ x:: x.co = false >

assign

< □ x:: x.co = true if x.g ∧ ¬ x*.co >

Koordinácia schôdzí 10

Ľahko vidno, že coord1 spĺňa cd1-cd3
Ukážeme, že ak prof spĺňa pr1-pr3 tak
sync = prof □ coord spĺňa sy1-sy3.

Sy1 $(\neg$ x.co unless x.g$)$

$\neg$ x.co unless x.g v coord1
$\neg$ x.co je stable v proof

Koordinácia schôdzí 11

sy2 $(\neg (x.co \wedge y.co \wedge E[x,y]))$

$\neg (x.co \wedge y.co \wedge E[x,y])$ v coord1

$\neg (x.co \wedge y.co \wedge E[x,y])$ je stable v prof

sy3 $(x.g \rightarrow x^*.co)$

x.g ensures x*.co v coord1

x.g unless x*.co pr4

Problém s implementáciou coord1 – nevieme bez centrálneho riadenia

Koordinácia schôdzí 12

- Problém pripomína večerajúcich filozofov
- komisia ..... filozof
- dve susedné komisie nemôžu zasadať .... susední filozofi nemôžu jesť
- len tá komisia, ktorá „konzumuje“ môže zasadať – x.co => x.e

Koordinácia schôdzí 13

--------------------sync--------------------

--------user---------------

prof <- x.co, u.g -> middle <- x.co, u.g -> os

----------coord------------

sync = prof ▯ middle ▯ os

user a os sú z večerajúcich filozofov

usr = prof ▯ middle

coord = middle ▯ os

Koordinácia schôdzí 14

Špecifikácia pre user:

- $u.t$ unless $u.h$ in user (udn1)
- stable $u.h$ in user (udn2)
- $u.e$ unless $u.t$ in user (udn3)

user a os zdieľajú len x.dine

v os x.t je constant a

x.e je stable

Pre návrh middle musíme určiť prechody:

1) $t \to h$

2) $e \to t$

Koordinácia schôdzí 15

- ak je komisia v stave t tak sa stane h iba ak všetci jej členovia čakajú
- ak prof □ middle bude spĹňať user tak z riešenia filozofov vieme, že hladná komisia bude nakoniec jesť
- niektorí členovia komisie nemusia čakať v okamihu keď táto začne jesť – môžu sa zúčastniť inej schôdze medzi $h \rightarrow e$

Koordinácia schôdzí 16

Program middle1

always

< x :: x.g = <∀u: u ∈ x.mem :: u.g >

initially

< ▯ x:: x.co = false >

assign

<▯ x::

x.dine := h if x.t ∧ x.g

▯

x.dine := t if x.e ∧ ¬ x.co >

Koordinácia schôdzí 17

Program os1

transformujeme príkazy os (z filozofov) tak aby sa vykonal nasledovný príkaz vždy keď sa zmení hodnota u.dine z h na e

x.co := x.g

Musíme ukázať že:

middle ▯ os1	spĺňa cd1-cd3
prof ▯ middle	spĺňa požiadavky na user
sync1 = prof ▯ middle ▯ os1	spĺňa „sync“

Koordinácia schôdzí 18

Lema 1. x.co => x.e v sync1

(t.j. nemôžu sa konať naraz susedné schôdze)

Dôkaz.

Na začiatku x.co = false a teda x.co => x.g platí. Keďže

x.co => x.e ≡ ⌉x.co v x.e , stačí ukázať, že

⌉x.co v x.e je stabilné v prof, middle a os1. Jediný zaujímavý prípad je os1. ⌉x.co unless x.e ale x.e je stabilné v os1 a z toho ⌉x.co v x.e je stabilné v os1

Koordinácia schôdzí 19

middle ▯ os1

z textu vidno, že to spĺňa cd1-cd2. Ukážeme že aj cd3 (x.co je stable)

z prechádzajúcej lemy vieme, že ⌉x.e=>⌉x.co
x.co sa nemení v middle a v os1 by sa mohlo zmeniť len ak by sa zmenilo x.e čo sa nemôže

Koordinácia schôdzí 20

prof □ middle (= usr)

udn1-udn3 platia lebo platia v middle a v prof sa nemení x.dine,
udn4 $(x.e \rightarrow \neg x.e)$ plynie z pr3

prof □ middle □ os1 (= sync)

z podmienok pre filozofov vieme:

$< \forall x,y\!: E[x, y] :: \neg(x.e \wedge y.e) >$	sy4
$x.h \rightarrow x.e \wedge (x.co = x.g)$	sy5
$x.e \rightarrow \neg x.e$	sy6

sy1-sy3 sa z tohto ľahko ukážu.

Koordinácia schôdzí 21

Ideme „počítať" u.g na asynchrónnej architektúre

u.g sa môže vyhodnotiť asynchrónne hlasovaním pre každé u.g, u $\in$ x.mem keď sa realizuje prechod $h \rightarrow e$

- x.prp : boolean – výsledok hlasovania
- x.prp je true ak u.g. je true pre všetkých doteraz hlasujúcich
- hlasovanie končí, ak x.prp je false alebo všetci hlasovali

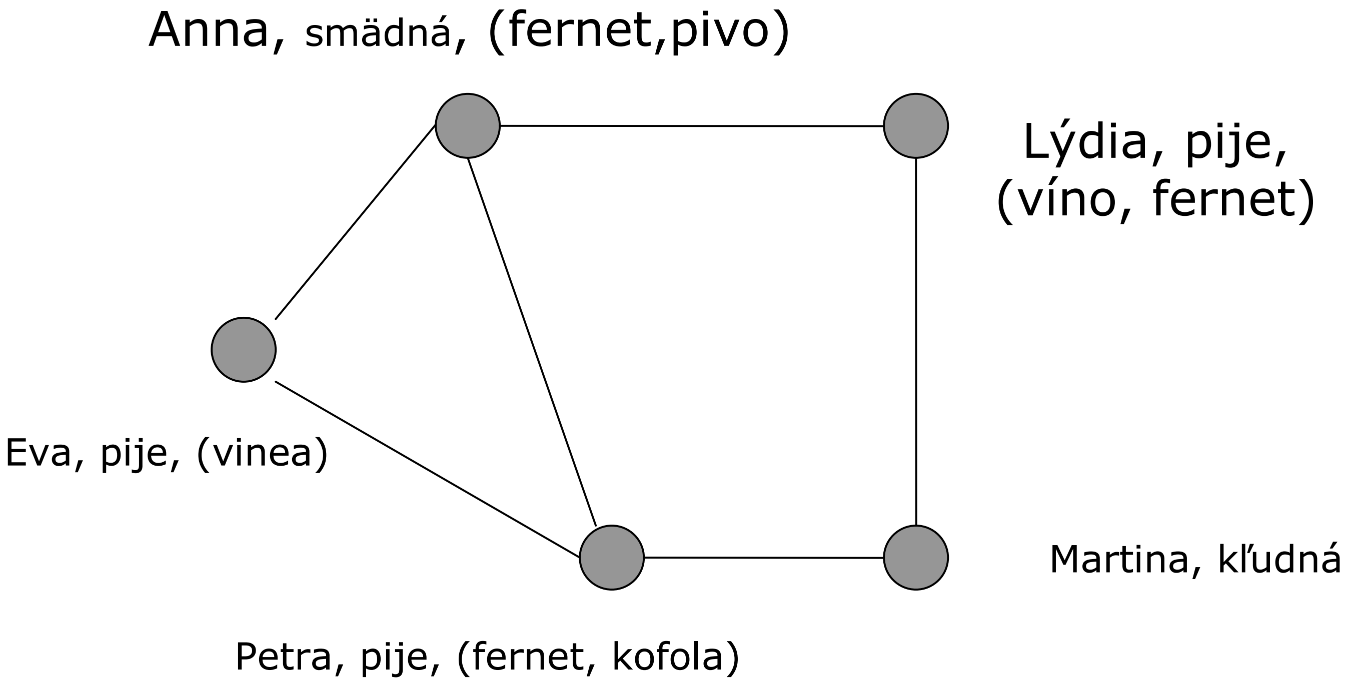
Pijúce filozofky 1

G=(V,E) neorientovaný, konečný, súvislý graf bez slučiek
filozofka – môže byť v 3 stavoch: kľudná, smädná, pijúca.
stavy mení v poradí:
kľudná $\rightarrow$ smädná $\rightarrow$ pijúca $\rightarrow$ kľudná

Každej smädnej alebo pijúcej (=nekľudnej)
filozofke je priradená nejaká množina nápojov (v okamihu prechodu z kľudná do smädná), ktorá sa pokiaľ je filozofka nekľudná nemení.

Úloha:
(1) zabezpečiť aby susedné filozofky nepili ak majú spoločný nápoj
(2) každá smädná bude raz piť za predpokladu, že nikto nepije večne

Pijúce filozofiky 2



Pijúce filozofky 3

situácia sa líši od večerajúcich filozofov – susedky môžu piť naraz, ak pijú rôzne nápoje, komu dať prednosť ak dopijú a opäť dostanú obe smäd?

„stav“ filozofky sa bude meniť v dvoch cykloch:

kľudná $\rightarrow$ smädná $\rightarrow$ pijúca $\rightarrow$ kľudná

thinking $\rightarrow$ hungry $\rightarrow$ eating $\rightarrow$ thinking

stav bude dvojica (stav_večerajúci, stav_pijúca)

Pijúce filozofky 4

Nemôžeme ovplyvniť prechody:

(x, kľudná) $\rightarrow$ (x, smädná)

(x, pijúca) $\rightarrow$ (x, kľudná) pre ľubovolný „večerajúci stav“ x

Predpokladáme, že:

1. (thinking, smädná) sa stane hungry alebo pijúca
2. hladná bude raz jesť (ak nikto neje večne)
3. konzumujúca a smädná sa stane meditujúcou a pijúcou, konzumujúca a nesmädná sa stane meditujúcou alebo smädnou

Pijúce filozofky 5

ideme navrhnúť osdrink
dané je userdrink

```
                    |------- osdrink-------|
userdrink           middle                  osdine
|----userdine------|
```

osdine ...... je os z večerajúcich filozofov
userdine ... je user z večerajúcich filozofov

bar = userdrink ▯ osdrink

u.k (kľudná), u.s (smädná), u.p (pijúca)
u.nap – množina nápojov pridelená u

Pijúce filozofky 6

Špecifikácia userdrink:

- u.k unless u.s udr1
- u.s je stable udr2
- u.p unless u.k udr3
- u.nap = N unless u.k udr4
- invariant u.nap = Ø $\equiv$ u.k udr5
- $\forall$u : u.p $\rightarrow \neg$ u.p udr6

udr4 - ak je u nekľudná, množina nápojov sa nemení a tá je prázdna ak je kľudná udr5

Odvodená vlastnosť: $\neg$ u.p je stable

Pijúce filozofky 7

Špecifikácia bar

invariant $\neg(u.p \wedge v.p \wedge E[u,v] \wedge (u.nap \cap v.nap \neq \emptyset))$	dr1
$u.s \rightarrow u.p$	dr2

Obmedzenia pre osdrink

u.k, u.nap =N sú constatnt	odr1
u.p je stable	odr2

Odvodené vlastnosti pre osdrink

$\neg$ u.s je stable	odr3
u.s unless u.p	odr4

Pijúce filozofky 8

Odvodené vlastnosti bar

u.k unless u.s	dr3
u.s unless u.p	dr4
u.p unless u.k	dr5
u.nap = N unless u.k	dr6
invariant u.nap = $\emptyset \equiv$ u.k	dr7
u.p $\rightarrow \urcorner$ u.p	dr8

Pijúce filozofky 9

userdrink	middle	osdine
$k \rightarrow s$	$s \rightarrow p$	
$p \rightarrow k$		
	$t \rightarrow h$	$h \rightarrow e$
	$e \rightarrow t$	

Pijúce filozofky 10

Špecifikácia middle

u.k, u.nap =N sú constatnt	mdr1
u.p je stable	mdr2
u.t unless u.h	mdr3
u.h je stable	mdr4
u.e unless u.t	mdr5
invariant $\neg(u.p \wedge v.p \wedge E[u,v] \wedge (u.nap \cap v.nap \neq \emptyset))$	mdr6
$(u.t \wedge u.s)$ ensures $\neg(u.t \wedge u.s)$	mdr7
$(u.e \wedge \neg u.s)$ ensures $\neg(u.e \wedge \neg u.s)$	mdr8
dr3-7, dr8, $<\forall u : u.p \rightarrow \neg u.p>$	mdr9

$<\forall u : u.e \wedge u.s \rightarrow u.t \wedge u.p>$

Pijúce filozofky 11

Prechod zo stavu smädná do pijúca

smädná u začne piť ak pre každú z jej
susediek v platí:

1) u konzumuje a v nepije nápoj, ktorý potrebuje u

alebo

2) nie je konflikt v nápojoch medzi u a v

Pijúce filozofky 12

Ideme zaručiť ako implementovať požiadavku, že v nepije niečo z u.nap.

Fľaškové riešenie:
pre každý nápoj máme fľašku (analógia s fork) a má ju u alebo v

bot[u,v,n] má hodnotu u alebo v – spoločná fľaška medzi u a v s nápojom n

Pijúce filozofky 13

Nová špecifikácia middle:

mdr1-5, mdr7-9 a

invariant

$u.p \Rightarrow \langle \forall n,v: n \in u.nap \wedge E[u,v] :: bot[u,v,n] = u \rangle$ mdr10

Presun fľašky:

smädná u získa fľašku s nápojom n, ktorú zdiela s v ak:

1) v ju nepotrebuje ($n \notin v.nap$)

alebo

2) u konzumuje a v nepije

Pijúce filozofky 14

Prechod zo smädná do pijúca

smädná u začne piť ak $\forall$n $\in$u.nap a pre všetky susedy v
spoločnú fľaša pre nápoj n má u a v nekonzumuje a
nepotrebuje n

Problémy:

musíme zistiť či v nekonzumuje a či n $\notin$ v.nap

Riešenie:

Zavedieme request-bottle token (rb), pomocou ktorého
proces
informuje susedku, že potrebuje fľašu

Pijúce filozofky 15

$u.t \wedge u.s$ neostane taký navždy – buď začne byť u hladná alebo začne piť (mdr7)

$u.e \wedge \neg u.s$ neostane taká navždy – buď začne u meditovať alebo začne byť smädná (mdr8)

prechod z $u.e \wedge u.s$ je $u.t \wedge u.p$ (mdr11)

smädná u začne piť ak $\forall n \in u.nap$ a $\forall$ v, E[u,v]:

- u drží fľašku n ktorú zdiela s v
- u drží fork, ktorú zdieľa s v alebo u nemá rb pre fľašku (mdr12-13)

Pijúce filozofky 16

poslanie fľašky

u pošle fľašku, ktorú zdieľa s v ak ju u má a má aj rb a u ju nepotrebuje (n $\notin$ u.nap) alebo u nepije a u nemá vidličku, ktorú zdieľa s v (mdr14-15)

poslanie rb

u pošle rb pre fľašku n ak u ho má ale nemá fľašku a u je smädný (mdr16-17)

Pijúce filozofky 17

u.maydrink $\equiv$

$< \forall n,v: n \in u.nap \wedge E[u,v] :: bot[u,v,n] = u \wedge$
$(rb[u,v,n]=v \vee fork[u,v]=u)>$

sendbot[u,v,n] $\equiv$

$bot[u,v,n]=u \wedge rb[u,v,n] =u \wedge$
$[n \notin u.nap \vee (fork[u,v]=v \wedge \neg u.p)]$

sendreq[u,v,n] $\equiv (rb[u,v,n]=u \wedge bot[u,v,n]=v \wedge n \in u.nap)$

Pijúce filozofky 18

špecifikácia middle

$(u.t \wedge u.s)$ unless $(u.t \wedge u.p)$	mdr11
$u.s$ unless $u.s \wedge u.maydrink$	mdr12
$(u.s \wedge u.maydrink)$ ensures $\neg(u.s \wedge u.maydrink)$	mdr13
bot[u,v,n]=u unless sendbot[u,v,n]	mdr14
sendbot[u,v,n] ensures $\neg$ sendbot[u,v,n]	mdr15
rb[u,v,n]=u unless sendreq[u,v,n]	mdr16
sendreq[u,v,n] ensures $\neg$ sendreq[u,v,n]	mdr17

Pijúce filozofky 19

Program middle

initially

⟨ □ *u*:: *u*.dine, u.drink = *t*, *k* ⟩

assign

⟨ □ *u*:: *u*.dine := *h* if *u*.*t* ∧ *u*.s

~ t if u.e ∧ ⌉ *u*.s ⟩

□ ⟨ □*u*: u.drink := p if u.s ∧ u.maydrink

|| u.dine := t if u.s ∧ u.maydrink ∧ u.e ⟩

□ ⟨ □ *u*, *v*, n: E[u,v]::

bot[u,v,n] := *v* if sendbot[*u*, *v*, *n*]

□ *rb*[*u*, *v*,*n*] := *v* if sendreq[u,v,n] ⟩

end

Mutual Exclusion

Ak by sme v probléme Večerajúcich filozofov uvažovali kompletný graf, tak len jeden filozof môže jesť v každom okamihu

Tento špeciálny prípad sa volá Mutual Exclusion Problem

V nasledujúcom ukážeme alternatívne riešenia tohto problému

Mutual Exclusion

Špecifikácia pre user:

$u.t$ unless $u.h$ in user

stable $u.h$ in user

$u.e$ unless $u.t$ in user

podmienená vlastnosť pre user:

$< \forall\ u,v : u \neq v :: \neg(\ u.e \wedge v.e) >$

$< \forall\ u :: u.e \rightarrow u.t >$

Mutual Exclusion

Úlohou je transformovať program user

na program

mutex = user′ ▯ G

kde user′ získame z user len pridaním príkazov, ktoré sa vykonajú súčasne s príkazom z user a G sú nové príkazy.

V G musí byť u.e stabilné a u.t konštantné. Jediné premenná, ktoré sa využívajú z user je u.dine

Mutual Exclusion

<u>Špecifikácia pre mutex:</u>

Invariant

$\neg(\text{u.e} \wedge \text{v.e} \wedge u \neq v)$ MX1

$u.\text{h} \rightarrow u.\text{e}$ MX2

Mutual Exclusion

Odvodené vlastnosti

$\neg$ u.e je stable	in user
u.*t* unless *u*.*h*	in mutex
u.h unless *u*.e	in mutex
u.*e* unless *u*.*t*	in mutex

Two Process Mutual Exclusion

Shared Memory architektúra

Ak niekoľko procesov chce zapisovať do premennej, tak môžu v nejakom (ľubovoľnom) poradí a vždy bude zapisovať lebn jeden.

Tento princíp ideme použiť v nasledovnom riešení. Použijeme pomocnú premennú turn.

Uvažujeme dva procesy x a y

u – jeden z nich, u’ – ten druhý

u.b – boolovská a stane sa true ak u zapisuje do turn

u.b – stane sa false, ak u prejde do thinking

Hladný proces u začne konzumovať – ak u.b je true a buď u’.b je false

Alebo u zapisoval do turn pred u’.

Two Process Mutual Exclusion

```
Program mutex2
declare
turn : hodnota x alebo y

initially
<  u :: u.dine, u.b = t, false>

transform - ak u.dine sa stane t tak  sa vykoná
u.b :=false

add
<  u ::
u.b, turn := true, u          if ¬ u.b ∧ u.h
 u.dine := e                 if u.b ∧ ¬(u’.b ∧ turn = u)
>
end
```

Two Process Mutual Exclusion

Dôkaz správnosti

Invariant

u.e => [u.b $\wedge$ $\neg$(u'.b $\wedge$ turn = u)] in mutex2

Z tohoto plynie MX1

Progres podmienka MX2

Ukážeme, že ak u je hladný, tak u.b sa stane true

Ak $\neg$(u'.b $\wedge$ turn = u) platí v tom čase, u má prioritu a tak u bude konzumovať

Inak (ak platí (u'.b $\wedge$ turn = u)) tak u' ma prioritu, bude jesť a po skončení u'.b sa stane false a prioritu bude mať u.

Two Process Mutual Exclusion

Zavedieme asynchrónnosť

– turn sa nastaví na u až keď sa u.b nastaví na true

Nová boolovská premenná u.r pre u.

u.r platí ak u.b bolo nastavené na true a do turn ešte len bude zapísané u

Two Process Mutual Exclusion

Program mutex2’
initially
< ▯ u :: u.dine, u.b, u.r = t, false, false>

transform - ak u.dine sa stane t tak sa vykoná
u.b :=false

add
< ▯ u ::

	u.b, u.r := true, true	if ¬ u.b ∧ u.h
▯	u.r, turn := false, u	if u.r
▯	u.dine := e	if u.b ∧ ¬u.r ∧ ¬(u’.b ∧ ¬u’.r ∧ turn = u)

>
end

N- Processes Mutual Exclusion

Rozdelíme procesy do dvoch neprázdných množín.

Rekuzívne aplikujeme algoritmus pre dva procesy na tieto množiny a ich podmnožiny.

Mutual Exclusion – Bakery Program

Zákazníci prichádzajú do pekárstva a berú si časenky, podľa ktorých sú obsluhovaní

issue – obsahuje „ďalšiu“ časenku, zvyšuje sa o jedna pri každom odobratí časenky

serve – obsahuje číslo, ktoré je práve obsluhované, alebo ktoré bude obsluhované ako ďalšie, ak danom okamihu nikto nie je obsluhovaný, zvyšuje sa o jedna po každom obslúžení zákazníka

Mutual Exclusion – Bakery Program

Na začiatku platí

issue = serve

Predpokladajme N zákazníkov

Stačia nám čísla 0 .. N-1 a budeme používať aritmetiku modulo N, + ... $\oplus$

Mutual Exclusion – Bakery Program

```
Program Bakery Program
declare
issue, serve: 0..N-1
num: array[processes] of 0..N

initially
issue, serve = 0,0
< □ u :: u.dine, u.num = t, N>

transform - ak u.dine sa stane t tak  sa vykoná
u.num, serve :=N, serve ⊕ 1

add
< □ u ::
     u.num, issue := issue, issue ⊕ 1            if u.num = N ∧ u.h
□    u.dine := e                                 if u.num = serve
>
end
```

Mutual Exclusion – Bakery Program

Alternatívny Bakery program

Každá premenná bude lokálna procesu

Zmena významu - u.num=0 ak u je thinking

Zavedieme novú premennú rank pre každý proces

u.rank je počet procesov, ktoré majú hodnotu num rovnú 0 plus počet procesov v pre ktoré je (u.num, u) lexikograficky menšie alebo rovné (v.num, v) – procesu majú číselne id, takže ich možno porovnávať

Vyšší rank znamená bližšie k tomu, aby proces získal službu

Mutual Exclusion – Bakery Program

Hladný proces u nastaví u.num na nenulovú hodnotu jej nastavením nad každé v.num – má tak rank nižší, ako každý iný hladný proces.

Pre rôzne u, v platí u.num ≠ v.num (aj ranky sú teda rôzne).

Ak je proces u hungry a u.num ≠ 0 a rank.u =N
tak u môže začať jesť.

Z rôznosti rankov máme zaručenú mutual exclusion.

Progres je zaručený, keďže rank pre žiadny proces nikdy neklesne a raz stúpne, keď iný proces doje.

u ≤ v ak (u.num, u) je lexikograficky menej alebo rovné ako (v.num,v)

Mutual Exclusion – Bakery Program

```
Program Bakery 2
always
< □ u :: u.rank = <+ v: (v.num =0) ∨ (u ≤ v) :: 1>

initially
< □ u :: u.dine, u.num = t, 0>

transform - ak u.dine sa stane t tak  sa vykoná
u.num:=0

add
< □ u ::
    u.num:= <max v :: v.num> +1      if u.num = 0 ∧ u.h
□   u.dine := e                      if u.num ≠ 0 ∧ u.h ∧ u.rank=N
>
end
```

Triedenie 1

Zadanie: $x[1..N]$ of integer (*navzájom rôzne*)

Úloha: nájsť pole $y[1..N]$ také, že

$$y \text{ je permutáciou } x \wedge \langle \forall i\colon 1 \leq i < N\colon\colon y[i] < y[i+1] \rangle \quad (1)$$

Dve základné stratégie:

- redukcia počtu zle usporiadaných dvojíc
- utriedenie časti poľa

Triedenie 2

1. Reduckia počtu zle usporiadaných dvojíc

(zatiaľ nešpecifikujeme, koľko dvojíc sa permutuje v každom kroku, v akom poradí sa permutujú a pod.)

- zaveďme metriku

  $M = \langle +i, j\text{: } 0 < i < j \leq \mathrm{N}\text{: } y[i] > y[j]\text{:: } 1 \rangle$
- stratégia je formálne definovaná nasledujúcim invariantom, FP a progress podmienkou:

  invariant *y* je permutáciou *x* (2)

  FP $\equiv$ (M = 0) (3)

  $\langle \forall k\text{: } k > 0\text{:: } M = k \rightarrow M < k \rangle$ (4)
- Dôkaz správnosti: stačí ukázať, že platí

  (2) $\wedge$ (3) $\wedge$ (4) $\Rightarrow$ true $\rightarrow$ FP

  a že (1) platí v každom FP.

Triedenie 3

2. Utriedenie časti poľa

budeme používať premennú m, $1 \le m \le \mathrm{N}$ a v každom bode výpočtu bude platiť

(a) $y[m + 1 .. \mathrm{N}]$ je utriedené

t.j. $\langle \wedge\, i, j\!: m < i < j \le \mathrm{N}\!:: y[i] < y[j] \,\rangle$

(b) všetky prvky $y[1 .. m]$ sú menšie než prvky $Y[m + 1 .. \mathrm{N}]$

t.j. $\langle \wedge\, i, j\!: 1 \le i \le m < j \le \mathrm{N}\!:: y[i] < y[j] \,\rangle$

keď zlúčime (*a*) a (*b*) dohromady, dostaneme

$\langle \wedge\, i, j\!: 1 \le i < j \le \mathrm{N} \wedge m < j :: y[i] < y[j] \,\rangle$

Na začiatku $m = \mathrm{N}$ to zaručuje a ak chceme znížiť m, tak musíme permutovať $y[1 .. m]$ tak, aby pre nejaké k, $1 < k \le m$, pole $y[k .. m]$ bolo utriedené a elementy tohto poľa $y[k .. m]$ boli väčšie než prvky poľa $y[1 .. k - 1]$; potom položíme $m := k - 1$

Triedenie 4

Formálne:

invariant *y* je permutáciou *x*

$\wedge\ 1 \leq m \leq \mathrm{N}$

$\wedge\ \langle\ \wedge\ i, j\text{: } 1 \leq i < j \leq \mathrm{N} \wedge m < j :: y[i] < y[j]\ \rangle$ (5)

$\mathrm{FP} \equiv (m \leq 1)$ (6)

$\langle\ \forall k\text{: } k > 1 :: m = k \rightarrow m < k\ \rangle$ (7)

Triedenie – Jednoduché riešenia 1

budeme uvádzať len assign sekciu a predpokladať, že

initially $\langle \; || \; i: 1 \leq i \leq N:: y[i] = x[i] \; \rangle$

1. Reduckia počtu zle usporiadaných dvojíc

Program P1

assign

$\langle \; \Box \; i: 1 \leq i < N ::$

$\quad y[i] := \min(y[i], y[i+1])$

$|| \quad y[i+1] := \max(y[i], y[i+1]) \; \rangle$

end{P1}

Triedenie – Jednoduché riešenia 2

Program P1′

assign

$\langle$ □ i: $1 \leq i < N$ :: $y[i], y[i+1] := \mathrm{sort2}(y[i], y[i+1])$

end{P1′}

pričom funkcie min, max a sort2 majú nasledovný význam:

min: $2^N \rightarrow N$; vráti minimum množiny prirodzených čísel

max: $2^N \rightarrow N$; vráti maximum množiny prirodzených čísel

sort2: $N \times N \rightarrow N \times N$; vráti usporiadanú dvojicu čísel

napr. min(7, 5, 1976) = 5, max(7, 5, 1976) = 1976, sort2(7, 5) = (5, 7)

Triedenie – Jednoduché riešenia 3

2. Utriedenie časti poľa

Nech f je funkciou y a m taká, že vráti index najväčšieho prvku $y[1 .. m]$.

P2 vymení prvky $y[m]$ a $y[f(y, m)]$ a zníži m o jednotku.

Program P2

declare

  x: integer

always

  $x = f(y, m)$ { teda $y[x] \geq y[i]$, $1 \leq i \leq m$ }

initially

  $m = N$

assign

  $y[m], y[x], m := y[x], y[m], m - 1$ if $m > 1$

end{P2}

Triedenie – Sekvenčné architektúry 1

Priame zobrazenie na SA dáva:

P1: má N – 1 príkazov, ktoré sa vykonávajú v cykle cez rastúce *i*, celý program potrebuje $O(\mathrm{N}^2)$ krokov

P2: podobne

V nasledujúcom ukážeme lepšiu realizáciu P1 a P2 na SA.

Z P2 odvodíme heapsort, zložitosť $O(\mathrm{N}.\log \mathrm{N})$ krokov.

Jadrom je zefektívnenie výpočtu maxima poľa $y[1 .. m]$.

Prvky $y[1 .. m]$ dáme do haldy tak, že $y[1]$ je maximum.

Triedenie – Sekvenčné architektúry 2

Program skeleton–heapsort
declare
    m: integer
initially
    $m = N$
assign
        $y[m], y[1], m := y[1], y[m], m - 1$
            if $y[1] = \langle \max j: 1 \leq j \leq m :: y[j] \rangle \wedge m > 1$
    ▯   príkazy na permutáciu $y[1 .. m]$ tak, že $y[1]$ je maximum
end

Triedenie – Sekvenčné architektúry 3

<u>strom:</u> $y[i]$ má synov $y[2i]$ a $y[2i + 1]$ (a opačne *byť_synom*)

<u>potomok:</u> nereflexívny tranzitívny uzáver relácie *byť_synom*

zavedieme pole *top*[1 .. N] of boolean,

$top[i] \Rightarrow \quad y[i]$ je väčšie ako $y[j]$,

kde j je potomok i a $1 \le j \le m$

formálne:

invariant

$\langle \forall\ i, j\!: 1 \le i \le m \wedge j \le m \wedge j$ je potomok $i \wedge top[i] :: y[i] > y[j] \rangle$ (8)

teda

$top[1] \Rightarrow y[1] = \langle \max j\!: 1 \le j \le m :: y[j] \rangle.$

Triedenie – Sekvenčné architektúry 4

Budeme používať nasledujúci

invariant $\langle\ \forall\ i:\quad m/2 < i \leq N :: top[i]\ \rangle$ (9)

teda $top[i]$ platí pre všetky listy a usporiadanú časť y.

Predpokladajme, že N je nepárne.

Myšlienka: ak $top[2i]$ a $top[2i + 1]$ platia, tak $top[i]$ bude platiť, ak $y[i] := \max(y[i], y[2i], y[2i + 1])$.

Nech $l[i]$ označuje index toho syna i, ktorý je väčší, ak i nemá syna, potom $l[i] = 2i$

Triedenie – Sekvenčné architektúry 5

Program P3 {nedeterministický heapsort}

declare

  l: array [1 .. N] of integer

  top: array [1 .. N] of boolean

always

  $\langle \square\ i: 1 \le i \le N ::$

  $l[i] = \quad 2i + 1 \quad$ if $y[2i + 1] > y[2i] \wedge (2i + 1 \le m) \sim$

  $\qquad\qquad 2i \qquad$ if $\neg(y[2i + 1] > y[2i] \wedge (2i + 1 \le m)) \quad \rangle$

initially

  $\langle \square\ i: 1 \le i \le N :: top[i] = (2i > N) \,||\, y[i] = x[i] \rangle \ \square\ m = N$

assign

  $y[m], y[1], m := y[1], y[m], m - 1$ if $top[1] \wedge m > 1$

  $||$ $\quad top[1], top[\lceil m/2 \rceil] :=$ false, true if $top[1] \wedge m > 2$

  $\square$ $\quad$ { usporiadanie $y[l[i]]$ a $y[i]$, ak $top[2i]$ a $top[2i + 1]$ }

  $\langle \square\ i: 1 \le i \le N/2::$

  $y[l[i]], y[i], top[i], top[l[i]] :=$ sort2$(y[l[i]], y[i])$, true, $(2.l[i] > m)$

  $\qquad$ if $(2i \le m) \wedge \neg top[i] \wedge top[2i] \wedge top[2i + 1] \rangle$

end{P3}

Triedenie – Sekvenčné architektúry 6

Dôkaz. Musíme ukázať, že P3 spĺňa (5), (6), (7). Z textu programu zrejme platia (5), (8), (9).

Dôkaz (7): $\langle \forall k \colon k > 1 :: m = k \rightarrow m < k \rangle$

Ukážeme, že platí

$$\text{true} \rightarrow top[1] \qquad (10)$$

$$m = k \text{ unless } m < k \qquad (11)$$

$$m = k \wedge k > 1 \wedge top[1] \rightarrow m < k \qquad (12)$$

Potom $m = k \rightarrow (m = k \wedge top[1]) \vee m < k$ z PSP aplikované na (10), (11) a z predchádzajúceho a (12) máme

$$m = k \wedge k > 1 \rightarrow m < k$$

(11) a (12) platia priamo z P3, ostáva dokázať (10).

Triedenie – Sekvenčné architektúry 7

Dôkaz (10): true $\rightarrow$ *top*[1]

Zavedieme metriku

$d = \langle +i: top[i]::$ počet vrcholov v podstrome s koreňom $i \rangle$

Vykonaním príkazu, ktorý položí *top*[*i*] := true pre nejaké *i* sa zvýši hodnota *d* (lebo podstrom s koreňom jeho synom je menší)

Teda máme

$d = D$ ensures $top[1] \vee d > D$

avšak hodnota *d* je ohraničená, teda

true $\rightarrow$ *top*[1]

Dôkaz (6): FP $\equiv$ $(m \leq 1)$

$m > 1 \rightarrow m \leq 1$ /* zo (7) */

Ak $p \rightarrow q$, vieme, že platí FP $\Rightarrow (p \Rightarrow q)$, teda ak $p \rightarrow \neg p$, vieme, že FP $\Rightarrow \neg p$. Nech $p \equiv m > 1$, potom FP $\Rightarrow m \leq 1$. Z textu programu vieme, že $m \leq 1 \Rightarrow$ FP.

Triedenie – Paralelné architektúry (synchrónne) 1

zistíme, čo sa dá vykonať paralelne

Program P4

assign

$\langle\ ||\ i{:}\ 1 \leq i < \mathrm{N} \wedge \mathrm{even}(i){::}\ y[i], y[i+1] := \mathrm{sort2}(y[i], y[i+1])\ \rangle$

$\square\quad \langle\ ||\ i{:}\ 1 \leq i < \mathrm{N} \wedge \mathrm{odd}(i){::}\ y[i], y[i+1] := \mathrm{sort2}(y[i], y[i+1])\ \rangle$

end{P4}

O(N) procesormi vykonanie P4 trvá *O*(N). Nasledujúci program priamo „počíta“ kroky.

Triedenie – Paralelné architektúry (synchrónne) 2

Program P5

declare m: integer

initially $k = 0 \ \square\ y[0] = -\infty \ \square\ y[N + 1] = \infty$

assign

$\langle \ || \ i: 1 \leq i \leq N :: y[i] := \min(y[i], y[i + 1]) \text{ if } (i = k \bmod 2)$

$\sim \quad \max(y[i], y[i - 1]) \text{ if } (i \neq k \bmod 2) \ \rangle$

$|| \quad k := k + 1 \text{ if } k < N$

end{P5}

Triedenie – Rank sort 1

Pre každý prvok z *y* vypočítame počet prvkov menších alebo rovných v *y* – táto hodnota je poradové číslo (pozícia) tohoto prvku v usporiadanom *y*

Program P6

declare *r*: array[1 .. N] of integer

always

$\langle \; || \; i: 1 \leq i \leq N :: r[i] = \langle +j: 1 \leq j \leq N \wedge x[j] \leq x[i] :: 1 \rangle \rangle$

▯ $\langle \; || \; i: 1 \leq i \leq N :: y[r[i]] = x[i] \rangle$

end{P6}

Triedenie – Rank sort 2

Dôkaz: treba ukázať nasledujúce tri vlastnosti programu P6:

1. rovnosti sú proper
2. *y* je permutácia *x*
3. *y* je usporiadané

Výraz $\langle +j\!: 1 \leq j \leq N \wedge x[j] \leq x[i] :: 1 \rangle$ možno vypočítať v čase $O(\log N)$ s $O(N)$ procesormi.

Takže všetky $r[i]$ možno vypočítať v čase $O(\log N)$ s $O(N^2)$ procesormi.

Alebo v čase $O(N)$ s $O(N)$ procesormi, ak jeden procesor ráta $r[i]$, a to v čase $O(N)$.

Protokol na komunikáciu cez chybové kanály (Faulty channels)

Proces sender má prístup k nekonečnej postupnosti dát *ms*

Proces *receiver*: jeho výstupom je postupnosť *mr*, ktorá spĺňa nasledovnú špecifikáciu:

invariant $mr \subseteq ms$

$|mr| = n \rightarrow |mr| = n + 1$

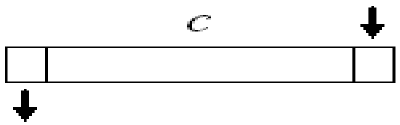
Faulty channels 2

Ak procesy komunikujú cez neohraničený spoľahlivý kanál “c”, riešenie je jednoduché:

c, ms := c; head(ms), tail(ms) {sender}

$\square$ c, mr := tail(c), mr; head(c) if $c \neq null$ {receiver}

kde initially $c = mr = null$ a head(p); tail(p) $\equiv$ p.



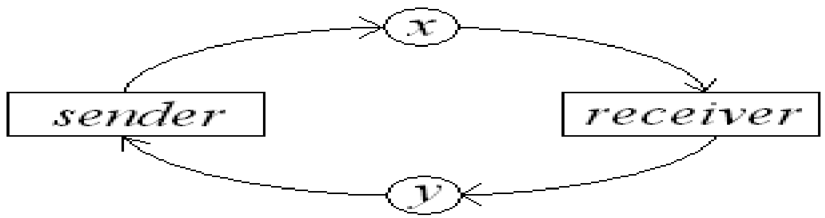
Faulty channels 3

Budeme riešiť problém, ak *c* je chybné – správa môže byť stratená alebo duplikovaná, ale len *konečne veľa ráz*

Zjednodušený problém:

sender a *receiver* komunikujú cez zdieľané premenné *x* a *y*:

- *sender* píše do *x* a číta *y*
- *receiver* píše do *y* a číta *x*



Faulty channels 4

- správa je <u>stratená</u>, ak *sender* zapíše do x prv, než to *receiver* prečítal
- správa je <u>zduplikovaná</u>, ak *receiver* znova číta x bez toho, že by sa medzitým obsah x zmenil
- podobne platí strata/duplikovanie pre y
- predpokladáme, že sender posiela postupnosť 1, 2, 3… (teda správy očíslujeme a namiesto správ posielame len ich čísla)
- nech:

  ks – posledné číslo poslané *sender*om

  kr – posledné číslo prijaté *receiver*om
- initially $ks = 1 \wedge kr = 0$
- invariant $kr \leq ks$ {prijíma sa len to, čo bolo poslané}
- $kr = n \rightarrow kr = n + 1$ {nakoniec sa prijmú všetky správy}

Faulty channels 5

požadovaný program musí spĺňať túto špecifikáciu a naviac jeho príkazy sa dajú rozdeliť do dvoch skupín (pre *sender* a *receiver*):

- *ks* vie čítať a písať len *sender* (sRW)
- *kr* vie čítať a písať len *receiver* (rRW)

- *x*, *y* sa môžu vyskytovať v oboch skupinách
- do *x* vie priraďovať len *sender*
- do *y* vie priraďovať len *receiver*

Faulty channels 6

zjemníme progress podmienku:

$$kr = n \rightarrow ks = n + 1 \rightarrow kr = n + 1$$

to sa zabezpečí príkazom

$$ks := kr + 1 \;\square\; kr := ks$$

pritom však toto nespĺňa podmienky (sRW) a (rRW)

opäť zjemníme progress podmienku:

$$kr = n \rightarrow y = n \rightarrow ks = n + 1 \rightarrow x = n + 1 \rightarrow kr = n + 1$$

- Program P1

  declare x, y, ks, kr: integer

  initially $x, y, ks, kr = 1, 0, 1, 0$

  assign

	$y := kr$	$\{receiver\}$
$\square$	$ks := y + 1$	$\{sender\}$
$\square$	$x := ks$	$\{sender\}$
$\square$	$kr := x$	$\{receiver\}$

  end

Faulty channels 7

x je chybový kanál, cez ktorý sa posiela ks
y je chybový kanál, cez ktorý sa posiela potvrdenie

invariant $y \leq kr \leq x \leq ks \leq y + 1$ (I1)

Dôkaz:

- na začiatku (I1) platí
- každé priradenie $a := b$ spĺňa $a \leq b$,

a teda vykonanie $a := b$ zachová $a \leq b$

Faulty channels 8

progress podmienka

$kr = n \rightarrow y = n \rightarrow ks = n + 1 \rightarrow x = n + 1 \rightarrow kr = n + 1$

Dôkaz: ukážeme pre $y = n \rightarrow ks = n + 1$, ostatné podobne

- $y = n$ ensures $ks = n + 1$ vyplýva z nasledovného:
- $y = n \wedge ks \neq n + 1 \Rightarrow y = kr = x = ks$ {z (I1)}
- $\{y = n\}\ ks := y + 1\ \{ks = n + 1\}$
- $\{y = n \wedge ks \neq n + 1\}\ s\ \{y = n \vee ks = n + 1\}$

Faulty channels 9

Prenos ľubovoľnej postupnosti dát

ms – nekonečná postupnosť (lokálna *sender*u)

mr – nekonečná postupnosť (lokálna *receiver*u)

ms[*j*], $j > 0$ označuje *j*-ty element *ms*

do *x* sa píše dvojica *x*.dex, *x*.val

Program P2

  declare *x*: (integer, data item), *y*, *ks*, *kr*: integer

  initially *y*, *ks*, *kr* = 0, 1, 0 ▯ *mr* = *null* ▯ *x* = (1, *ms*[1])

  assign

$y := kr$	{*receiver*}
▯ $ks := y + 1$	{*sender*}
▯ $x := (ks, ms[ks])$	{*sender*}
▯ $kr, mr := x.\mathrm{dex}, mr\ ;\ x.\mathrm{val}$ if $kr \neq x.\mathrm{dex}$	{*receiver*}

end{P2}

Faulty channels 10

Správnosť P2:

invariant $mr \subseteq ms$

$|mr| = n \longrightarrow |mr| = n + 1$

Pozorovanie: (I1) platí aj pre P2, kde miesto x je x.dex

označenie x.val $\in ms$ ak $\exists j : x.\text{val} = ms[j]$

platí nasledovný

invariant $x.\text{val} \in ms \land mr \subseteq ms \land |mr| = kr$ (I2)

(*Hint:* z (I1) $kr \neq x.\text{dex} \Rightarrow kr + 1 = x.\text{dex}$)

Progress podmienka má podobný dôkaz ako v P1

Poznámka. Z (I1) a *hint*u $kr := x.\text{dex}$ if $kr \neq x.\text{dex}$ možno v P2 nahradiť $kr := kr + 1$ if $kr \neq x.\text{dex}$ a zároveň možno $kr := y + 1$ nahradiť $ks := ks + 1$ if $ks = y$

Faulty channels 11

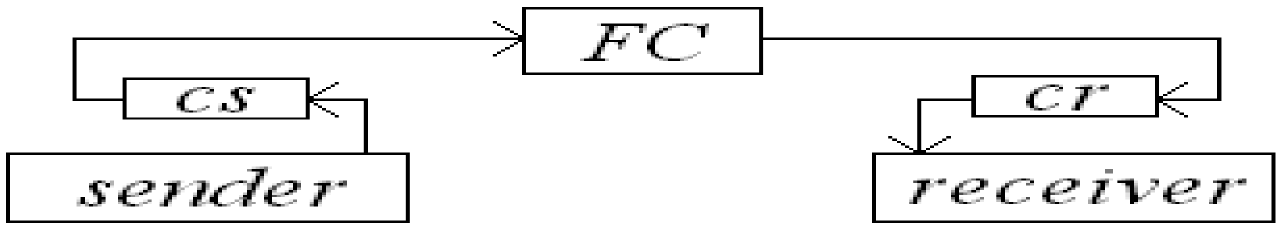
V predchádzajúcich programoch duplikovanie a strata správy boli „pod kontrolou“ *sender*a a *receiver*a

špecifikácia chybného (resp. chybového) kanála:

- každá správa môže byť konečne veľa krát stratená
- každá správa môže byť konečne veľa krát duplikovaná
- správy sú prenášané v nezmenenom poradí
- správy nemôžu byť poškodené

Faulty channels 12

situácia s chybovým kanálom FC a bezchybovými kanálmi *cs*, *cr*:



Faulty channels 13

FC môže len odobrať (head) z *cs*

cs je suffix cs^0 je stable v FC

(všeobecne kvantifikované, cs^0 je konštantná postupnosť)

FC môže pridať na koniec *cr*

cr^0 je prefix *cr* je stable v FC

FC môže stratiť správu z *cs* alebo duplikovať správu do *cr*

- $\underline{cs}$: celá „história" cs
- $\underline{cs} - cs$: prefix $\underline{cs}$ bez *cs*

u loss *v*: *v* je možný výstup z chybného kanála, za predpokladu, že vstup je *u*

$(null \text{ loss } null)$

$(u \text{ loss } v) \Rightarrow (u;m \text{ loss } v)$ *strata*

$(u;m \text{ loss } v) \Rightarrow (u;m \text{ loss } v;m)$ *duplikácia*

invariant $(\underline{cs} - cs)$ loss $\underline{cr}$

Faulty channels 14

ak správa *m* (a žiadna iná) je nekonečne veľa krát pridaná do *cs*, tak sa *m* raz (*eventually*) objaví v *cr*

Predpokladáme množinu predikátov *p*.*m* (predikát existuje pre každé *m*)

- ak *p*.*m* platí, žiadne iná správa ako *m* nemôže byť pridaná do *cs*
- *p*.*m* platí nanajvýš pre jedno *m* (počas „výpočtu“)
- ak *p*.*m* je true, tak raz (*eventually*) bude false alebo *m* je pridané do *cs* (t.j. zvýši sa dĺžka $\underline{cs}$)

za týchto predpokladov FC zaručí, že ak raz *p*.*m* je true, potom sa raz (*eventually*) bude rovnať false alebo *m* sa objaví v *cr*

Predpoklad:

$p.m \wedge p.n \Rightarrow m = n$

$p.m \wedge cs = null$ unless $\neg p.m \vee cs = \langle\langle m \rangle\rangle$

$p.m \wedge (cs = cs^0) \wedge (cs \neq null)$ unless $\neg p.m \vee cs = \mathrm{tail}(cs^0) \vee cs = cs^0;m$

$p.m \wedge |\underline{cs}| = k \rightarrow \neg p.m \vee |\underline{cs}| > k$

Záver: $p.m \rightarrow \neg p.m \vee (m \in cr)$

Faulty channels 15

Program simulujúci FC

strata správy

$cs := \text{tail}(cs) \text{ if } cs \neq null$

duplikácia správy

$cr := cr;\text{head}(cs) \text{ if } cs \neq null$

správny transfér

$cs, cr := \text{tail}(cs), cr;\text{head}(cs) \text{ if } cs \neq null$

neobsahuje predpoklad, že len *konečne veľa* chýb možno vykonať za sebou

zavedieme premennú b: boolean, ktorá značí „chyba nenastane"

Faulty channels 16

Program FC

declare b: boolean

initially b = false

assign

cs := tail(cs) if $cs \neq null \wedge \neg b$

$\square$ cr := cr;head(cs) if $cs \neq null \wedge \neg b$

$\square$ b, cs, cr := false, tail(cs), cr;head(cs) if $cs \neq null$

$\square$ b := true

end

Faulty channels 17

Protokol ▯ FC pre komunikáciu s dvoma chybnými kanálmi (pre *cr*, *cs* a pre *ackr*, *acks*) má splňať nasledujúce podmienky:

invariant $mr \subseteq ms$

$|mr| = n \rightarrow |mr| = n + 1$

Modifikujme P2 nasledovným spôsobom:

Faulty channels 18

Program P3
declare *ks*, *kr*: integer
initially
        ks, *kr* = 1, 0
    $\square$ *cs*, *cr*, *acks*, *ackr* = *null*, *null*, *null*, *null*
    $\square$ *mr* = *null*
assign
        ackr := *ackr*;*kr*
    $\square$ *ks* := *ks* + 1 if *acks* ≠ *null* ∧ *ks* = head(*acks*)
        || *acks* := tail(*acks*) if *acks* ≠ *null*
    $\square$ *cs* := *cs*;(*ks*, *ms*[*ks*])
    $\square$ *kr*, *mr* := *kr* + 1, *mr*;head(*cr*).val if *cr* ≠ *null* ∧ *kr* ≠ head(*cr*).dex
        || *cr* := tail(*cr*)
end

Faulty channels 19

Pri dokazovaní správnosti protokolu použijeme upravené invarianty (I1), (I2)

invariant $y \le kr \le x.\mathrm{dex} \le ks \le y + 1$ (I1)

invariant $x.\mathrm{val} \in ms \wedge mr \subseteq ms \wedge |mr| = kr$ (I2)

(I1) sa dá prepísať (x, y môžu byť nedefinované, ak zodpovedajúce kanály sú *null*) nasledovným spôsobom:

$\langle \forall y \colon y \in acks \vee y \in ackr :: y \le kr \wedge ks \le y + 1 \rangle \wedge$

$\langle \forall x \colon x \in cs \vee x \in cr :: kr \le x.\mathrm{dex} \le ks \rangle \wedge$

$kr \le ks \le kr + 1$

podobne sa dá prepísať aj (I2)

potrebujeme ešte jeden invariant:

invariant $cs.\mathrm{dex}$, $cr.\mathrm{dex}$, $acks$, $ackr$ sú neklesajúce postupnosti (I3)

Úloha: dôkaz (I1), (I2), (I3) a nasledujúcej progress podmienky:

$kr = n \rightarrow kr = n + 1$

Faulty channels 20

Z (I1) pre P3 máme

ak $cr \neq null$ tak $kr \leq \text{head}(cr).\text{dex} \leq kr + 1$

teda

$kr \neq \text{head}(cr).\text{dex} \equiv (kr \bmod 2 \neq [\text{head}(cr).\text{dex}] \bmod 2)$

$ks = \text{head}(acks) \equiv (ks \bmod 2 = \text{head}(acks) \bmod 2)$

P3 môže byť zjednodušené tak, že nahradíme *kr*, *ks* nasledujúcimi: *kr* mod 2, *ks* mod 2

dostávame Alternating Bit Protocol

Global snapshots 1

<u>Úloha</u>: modifikovať daný program tak, aby sme mohli zaznamenať stav jeho výpočtu

Toto je problémom pri asynchrónnych a distribuovaných systémoch

<u>Príklad:</u>

Program P

declare x, y, z: integer

initially x, y, z = 0, 0, 0

assign $x := z + 1 \;\square\; y := x \;\square\; z := y$

end

Global snapshots 2

stavom P je trojica hodnôt x, y, z

ľahko vidno, že jediná možná postupnosť stavov P je
$(0, 0, 0) \rightarrow (1, 0, 0) \rightarrow (1, 1, 0) \rightarrow (1, 1, 1) \rightarrow (2, 1, 1) \rightarrow (2, 2, 1) \rightarrow \ldots$

P spĺňa invariant
invariant $(x \geq y) \wedge (y \geq z) \wedge (z + 1 \geq x)$

ak hodnotu x zaznamenáme v stave (0, 0, 0) a hodnoty y a z v stave (1, 1, 0), tak zaznamenaný stav (0, 1, 0) nie je stavom žiadneho výpočtu

Global snapshots 3

Podrobnejšia špecifikácia:

- pre zjednodušenie budeme predpokladať, že stav sa zaznamenáva len raz (ľahko sa to rozšíri pre viacnásobné použitie)
- transformujeme daný program tak, že potom tento zaznamená stav pôvodného programu
- *zaznamenaný stav* – stav, ktorý sa môže vyskytnúť vo výpočte, a to medzi tým, keď je zaznamenávanie (záznam) inicializované a ukončené
- zaznamenávanie je inicializované, ak premennej *begun* je priradená hodnota true
- *begun* je stable

Global snapshots 4

- nezaoberáme sa tým, kedy sa *begun* stane true
- predpokladáme, že ak *begun* sa stane true, ak v konečnom čase sa ukončí zaznamenávanie
- pod *stavom* myslíme stav (t.j. hodnoty premenných) pôvodného programu
- [*s*0] *v* [*s*1], kde *s*0, *s*1 sú stavy a *v* je konečná postupnosť príkazov, znamená: ak začneme v *s*0 a vykonáme *v*, tak skončíme v *s*1
- *init* – stav, pri ktorom je zaznamenávanie inicializované (*begun* $\leftarrow$ true)
- *rec* – stav zaznamenaný transformovaným programom

Global snapshots 5

- *cur* – okamžity stav výpočtu
- požadujeme, aby existovali postupnosti *u*, *v* také, že [*init*] *u* [*rec*] $\wedge$ [*rec*] *v* [*cur*] ak bolo zaznamenávanie ukončené
- stavové premenné:
  x.done – platí práve vtedy, keď *x* bolo zaznamenané
- zaznamenávanie je ukončené, ak $\forall x$ platí *x*.done
- zaznamenaná hodnota je uložená do *x*.rec
- stav *rec* je teda daný hodnotami $\langle\ \forall x{::}\ x.\mathrm{rec}\ \rangle$
- stav *cur* je daný momentálnymi hodnotami *x*

Global snapshots 6

invariant a progress podmienka:

invariant $\langle \wedge x{::}\ x.\mathrm{done} \rangle \Rightarrow \langle \exists u, v{:}\ [init]\ u\ [rec] \wedge [rec]\ v\ [cur] \rangle$
begun $\longrightarrow \langle \wedge x{::}\ x.\mathrm{done} \rangle$

neformálne:

- zaznamenávanie sa skončí v konečnom čase
- [*rec*] je na nejakej „ceste“ medzi [*init*] a [*cur*]

Program P1
    initially $\langle\ \|x{::}\ x.\mathrm{done} = \mathrm{false} \rangle$
    add $\langle\ \|x{::}\ x.\mathrm{rec}, x.\mathrm{done} = x, \mathrm{true\ if}\ begun \wedge \neg x.\mathrm{done} \rangle$
end{P1}

P1 je nevhodný pre asynchrónne a distribuované počítače

Global snapshots 7

<u>Zjemnenie pôvodnej špecifikácie:</u>

pre každý stav výpočtu (pôvodného programu) definujeme

$$x.\text{partial} = \begin{cases} x.\text{rec} & \text{if } x.\text{done} \\ \sim\ x & \text{if } \neg x.\text{done} \end{cases}$$

invariant a progress podmienka:

invariant *begun* $\Rightarrow \langle\, \exists u, v$: v obsahuje len zaznamenané premenné:: $[init]\ u\ [partial] \wedge [partial]\ v\ [cur]\,\rangle$

begun $\longrightarrow \langle\, \wedge x{::}\ x.\text{done}\,\rangle$

Global snapshots 8

neformálne:

- *partial* je kombináciou *rec* a *cur*
  - pre zaznamenané *rec* = *partial*
  - pre nezaznamenané *cur* = *partial*
- keď sa inicializuje zaznamenávanie, *init* = *partial*
- keď sa zaznamenávanie skončí, *rec* = *partial*
- zaznamenávanie nemení *partial*
- *v* necháva *partial* nezmenený

Výpočet pre program P:

Global snapshots 9

Events	State	Zaznamenané	partial	x.rec	y.rec	z.rec
initially	0, 0, 0	_	0, 0, 0	_	_	_
zmena stavu	1, 0, 0	_	1, 0, 0	_	_	_
záznam x	1, 0, 0	x	1, 0, 0	1	_	_
záznam y	1, 0, 0	x, y	1, 0, 0	1	0	_
zmena stavu	1, 1, 0	x, y	1, 0, 0	1	0	_
záznam z	1, 1, 0	x, y, z	1, 0, 0	1	0	0
zmena stavu	1, 1, 1	x, y, z	1, 0, 0	1	0	0

Global snapshots 10

<u>Pravidlo *R*:</u>

- Keď sa príkaz daného programu vykoná, tak platí jedno z nasledujúcich:
    - všetky premenné v danom príkaze sú zaznamenané,
    - všetky premenné v danom príkaze sú nezaznamenané;

- hodnota *x* môže byť zaznamenaná hocikedy potom, čo bolo zaznamenávanie inicializované, a prv, než sa vykoná príkaz obsahujúci *x* a nejaké už zaznamenané hodnoty.

- Tvrdenie: Každý program, ktorý implementuje *R*, spĺňa predchádzajúci invariant.

Global snapshots 11

Neformálny dôkaz:

Invariant platí, keď inicializujeme zaznamenávanie, za *u*, *v* vezmeme *null*.

Ukážeme, že invariant sa zachová po vykonaní príkazu *t*, ktorý spĺňa *R*.

- nech všetky premenné v *t* sú nezaznamenané a platí invariant pred vykonaním *t*. *v* obsahuje len premenné zaznamenané, teda *v* a *t* nemajú spoločné premenné. Môžu byť teda vykonané nezávisle na sebe;
    - za *u* vezmime *u*;*t*
    - za *v* vezmime *v*
- nech všetky premenné v *t* sú zaznamenané; potom *partial* sa nemení;
    - za *u* vezmime *u*
    - za *v* vezmime *v*;*t*

Global snapshots 12

Formálne:

V okamihu, keď sa inicializuje zaznamenávanie (*begun* $\leftarrow$ true) platí

$$init = cur \wedge partial = cur$$

teda invariant platí pre $u = v = null$.

Predpokladajme, že invariant platí pre nejaké u, v a nejaký výpočet.

Global snapshots 13

Ak sa zaznamená premenná, tak sa nemenia hodnoty *partial* alebo *cur* a teda môžeme zobrať pôvodné *u*, *v*.

Teraz budeme uvažovať, že sa vykoná prílaz *t*. Nech *cur′* je stav pred vykonaním *t* a *cur* je stav po vykonaní *t*.

Podobne pre *partial*.

Z invariantu máme:

$$[init]\ u\ [partial'] \wedge [partial']\ v\ [cur']$$

ďalej

$$[cur']\ t\ [cur]$$

Musíme nájsť *u′*, *v′* tak, aby platilo

$$[init]\ u'\ [partial] \wedge [partial]\ v'\ [cur]$$

(Vykonanie *t* neovplyvňuje *x*.done alebo *x*.rec.)

Global snapshots 14

Prípad 1.

t obsahuje nezaznamenané premenné.

Potom ľahko vidno

[*partial'*] *t* [*partial*]   (**)

a

[*init*] *u*;*t* [*partial*]

[*partial'*] *v*;*t* [*cur*]   (*)

t obsahuje len nezaznamenané premenné t.j. možno vymeniť poradie *t* a *v*, máme teda

[*partial'*] *t*;*v* [*cur*]

z predchádzajúceho (**)

[*partial*] *v* [*cur*]

Global snapshots 15

Prípad 2.

t obsahuje len zaznamenané premenné.

Z definícií

x.partial = x.rec if x.done ~ x.cur if $\neg x$.done

x.partial' = x.rec if x.done ~ x.cur' if $\neg x$.done

teda *partial* = *partial'*

[*init*] *u* [*partial*]

ďalej z [*partial'*] *v* [*cur'*] máme

[*partial*] *v*;*t* [*cur*]

Global snapshots 16

Jednoduché programy na zaznamenanie stavu

Asynchrónna shared-memory:

- v nejakom okamihu sa procesy dočasne zastavia
- zastavený proces ostane zastavený, kým sa nezaznamená globálny stav
- keď sú všetky zastavené, ich stavy sú zaznamenané
- po zaznamenaní sa opäť spustia

takto modifikovaný program triviálne spĺňa *R*

Global snapshots 17

Distribuovaná architektúra:

- využijeme proces *central*
- *central* pošle všetkým procesom požiadavku zastaviť výpočet
- každý proces po obdržaní tejto správy pošle potvrdenie *central*u a zastaví sa
- keď *central* dostane potvrdenie od každého procesu, všetkým pošle príkaz na zaznamenanie stavu
- každý proces zaznamená stav a pošle ho *central*u
- keď *central* dostane od každého jeho stav, pošle všetkým pokyn na pokračovanie
- keď procesy tento pokyn na pokračovanie dostanú, pošlú *central*u potvrdenie a pokračujú
- keď *central* dostane všetky potvrdenia o pokračovaní, je pripravený na ďalší záznam
- každý proces využije 6 správ, triviálne to spĺňa *R*

Global snapshots 18

zatiaľ sme ignorovali, čo s obsahom kanálov (proces nemôže čítať priamo cez kanál)

- stav kanálu vypočítame z toho, čo bolo poslané *mínus* to, čo bolo prijaté
- pre kanál *c* platí:
  - *c*.sent = *c*.received; *c*.state

t.j.

  - *c*.state = *c*.sent – *c*.received
- *c*.sent je lokálna tomu, čo posiela
- *c*.received je lokálna tomu, čo prijíma

Definujeme:

c.state.rec = *c*.sent.rec – *c*.received.rec

- takýto mechanizmus je dosť neefektívny, lebo *c*.received, *c*.sent môžu byť veľmi dlhé, no my potrebujeme len ich rozdiel

Modifikácia:

- keď proces zastaví činnosť, prv než zaznamená svoj stav, pošle na každý výstupný kanál *marker* (značku)

Global snapshots 19

Efektívne programy na zaznamenanie stavu

<u>Asynchrónna shared-memory:</u>

- predpokladajme, že premenná y v príkaze t bola zaznamenaná t.j. platí y.done a že iná premenná x v t nebola zaznamenaná, t.j. $\neg x$.done
- pravidlo R hovorí, že t nemôže byť vykonané skôr, než sa x nezaznamená
- zaznamenanie x možno dorobiť zároveň s t:

$\langle \; || x$: t obsahuje $x \wedge \neg x$.done::

x.rec, x.done = x, true if $\langle \; \exists y$: t obsahuje y:: y.done $\rangle\rangle \;\; || \; t$

Global snapshots 20

na začiatku predpokladajme, že zaznamenávanie je iniciované zaznamenávaním jednej špecifickej premennej *first*

$$\textit{first}.\text{rec}, \textit{first}.\text{done} := \textit{first}, \text{true if } \textit{begun} \wedge \neg\textit{first}.\text{done}$$

je isté, že v konečnom čase po zaznamenaní *first* bude zaznamenaná každá premenná, ktorá sa vyskytuje v nejakom príkaze *t* spolu s *first*

Definujme:

x je_spolu s $y \equiv \langle \exists$ príkaz t:: t obsahuje $x \wedge t$ obsahuje $y \rangle$

je_spolu* $\equiv$ tranzitívny uzáver je spolu

Platí:

x.done $\wedge$ x je_spolu s $y \longrightarrow y$.done

x.done $\wedge$ x je_spolu* s $y \longrightarrow y$.done

Global snapshots 21

ak pre nejakú premennú y neplatí

first je_spolu* s y

tak premenné rozdelíme do skupín a zaznamenávanie aplikujeme zvlášť pre každú skupinu

Program P2
initially $\langle\ ||x{::}\ x.\text{done} = \text{false}\ \rangle$
transform $\forall t$
$\langle\ ||x{:}\ t$ obsahuje $x \wedge \neg x.\text{done}{::}$
$x.\text{rec}, x.\text{done} = x, \text{true if } \langle\ \exists y{:}\ t \text{ obsahuje } y{::}\ y.\text{done}\ \rangle\rangle$
$||\ t$
add
$\mathit{first}.\text{rec}, \mathit{first}.\text{done} = \mathit{first}, \text{true if } \mathit{begun} \wedge \neg\mathit{first}.\text{done}\ \rangle$
end{P2}

Global snapshots 22

Distribuovaná architektúra:

1 Nech iniciátor je proces, ktorý iniciuje zaznamenávanie tak, že zaznamená svoj vlastný stav a pošle markre na všetky vychádzajúce kanály

2 Proces, ktorý ešte nezaznamenal svoj stav, po prijatí markra zaznamená svoj stav a pošle marker na všetky vychádzajúce kanály, ktoré mu patria

3 Stav kanála je zaznamenaný procesom, ktorý z neho prijíma správy – je to postupnosť správ prijatá po tom, čo jeho vlastný stav je zaznamenaný a prv, než sa príjme marker

Global snapshots 23

Použitie logických hodín pri distribuovanej architektúre

Každému procesu u priradíme nezápornú číselnú premennú u.m – „logické hodiny" a na začiatku u.m = 0

1. u.m sa zvýši súčasne s vykonaním každého príkazu u
2. Ak u pošle správu, tak k nej pridá „timestamp" shodnotou u.m
3. Ak v príjme správu s „timestamp" k tak do v.m sa dá hodnota väčšia než k
4. Každý proces u zaznamená svoj stav keď u.m. dosiahne Z, kde Z je nejaká konštanta
5. Správa je zaznamenaná, ak logický čas keď je poslaná je Z alebo menší a logický čas kedy je prijatá je väčší ako Z

Detekovanie stabilných vlastností 1

Detekovanie stabilných vlastností:

- je to problém, keď to musíme robiť počas behu programu
- rozdelíme úlohu na dve časti:
  1. zaznamenáme globálny stav
  2. zistíme, či zaznamenaný stav má požadovanú vlastnosť

Detekovanie stabilných vlastností 2

- claim ..... boolovská premenná
- W .... stabilná vlastnosť

claim detects W   ak:

invariant claim => W

$W \rightarrow$ claim

Detekovanie stabilných vlastností 3

- rec …………zaznamenaný stav
- W(s) hodnota W v stave s

Definujme:

$$\text{claim} \equiv W(\text{rec})$$

(predpokladajme, že počiatočná hodnota rec je taká, že $\neg$ W(rec))

Detekovanie stabilných vlastností 4

Riešenie:

rec:= globálny stav výpočtu if $\neg$ W(rec)

t.j. z času na čas sa zaznamenáva rec až kým neplatí W(rec)

Detekovanie stabilných vlastností 5

Dôkaz správnosti stratégie, t.j. toho, že platí:

W(rec) detects W

(W- hodnota W(st) pre momentalny stav st)

Dôkaz invariantu (invariant W(rec) => W)

 každý stav st po zaznamenaní je dosiahnuteľný z rec i.e. existuje v

 [rec] v [st]

Detekovanie stabilných vlastností 6

Keďže W je stable tak pre každé dva stavy s1,s2 také, že s2 je dosiahnuteľné z s1 platí:

$$W(s1) => W(s2)$$

a odtiaľ

$$W(rec) => W(s)$$

pre všetky stavy s, ktoré sa vyskytujú po ukončení zaznamenávania

Detekovanie stabilných vlastností 7

Dôkaz progress podmienky ($W \to W(rec)$)

ak zaznamenávanie je iniciované v stave keď W platí, tak W bude platiť aj v stave rec (keďže je W stabilné)

Platí W ensures W(rec)

keďže

$\{W \wedge \neg W(rec)\}$

rec:= globálny stav výpočtu if $\neg W(rec)$

$\{W(rec)\}$

Detekcia terminácie

Detekcia terminácie

daná množina procesov V, každý proces môže byť v dvoch stavoch:

- idle (dosiahol pevný bod)
- active ($\neg$ idle)

ak je process v idle tak jeho stav môže zmeniť iný proces u, ktorý s ním zdieľa premennú, ktorú u môže meniť (tento vzťah označíme grafom (V,E), kde (u,v) je hrana) t.j.:
v.idle := v.idle $\wedge$ u.idle

aktívny proces sa môže kedykoľvek stať idle

Detekcia terminácie

Program R0

assign

$\quad \langle \,\square\, u,v : (u,v) \in E :: v.idle := v.idle \wedge u.idle \,\rangle$

$\square \quad \langle \,\square\, u : u \in V :: u.idle := true \,\rangle$

end

terminácia nastane, keď všetky procesy sú idle, t.j.

$W \equiv \langle \wedge\, u : u \in V :: u.idle \,\rangle$

claim detects W

Detekcia terminácie

Program R1

initially claim = false

add

  claim := $< \wedge\ u : u \in V :: u.idle >$

tento program síce detekuje termináciu ale je vhodný len pre sekvenčné architektúry

Detekcia terminácie

Asynchrónna shared-memory architektúra

Stratégia:
invariant d = {u | u.idle}
claim detects (d=V)

```
Program R2
initially claim = false □ d = {u | u.idle}
assign
    < □ u,v :(u,v)∈ E:: v.idle := v.idle ∧ u.idle
        || d:= d - {v} if ¬u.idle >
□   < □ u : u ∈ V:: u.idle := true || d:= d ∪ {u} >
□   claim := (d=V)
end
```

Detekcia terminácie

nevýhoda R2 je, že d sa mení každým príkazom

Vylepšenie:
b – množina procesov
u.delta –množina procesov, ktoré u
aktivoval od poslednej zmeny b (u.delta je lokálna ku u)

Detekcia terminácie

Program R3
initially claim = false □ b=∅ □ < □ u : u.delta= ∅ >
assign
    < □ u,v :(u,v)∈ E:: v.idle := v.idle ∧ u.idle
        || u.delta:= u.delta ∪ {v} if ¬u.idle >
□ < □ u : u ∈ V:: u.idle := true >
□ < □ u : b,u.delta := b ∪ {u} −u.delta, ∅ if u.idle >
□ claim := (b=V)
end

Detekcia terminácie

Distribuovaná architektúra

- Idle proces sa stane aktívnym po prijatí správy
- Idle proces neposiela správy

Podmienka ktorú ideme detekovať:

- Všetky procesy sú idle
- Všetky kanály sú prázdne

Detekcia terminácie

Kanál budem reprezentovať ako

(u,v).c – kanál z u do v, je typu postupnosť

m = správa, nemusíme rozlišovať aká je, t.j. m označuje každú správu

Detekcia terminácie

Program

assign

{u pošle správu m cez kanál (u,v).c ak u je aktívny}

< □ (u,v) :: (u,v).c := (u,v).c;m if ¬u.idle >

{v sa stane aktívnym po prijatí správy z kanálu (u,v)}

□ < □ (u,v) :: v.idle,m,(u,v).c := false, head((u,v).c;m),
tail((u,v).c;m) if (u,v).c ≠ null >

□ < □ u.idle := true>

end

Detekcia terminácie

- Idle proces v distribuovanej architektúre odpovedá tomu v nedistribuovanej, ktorý ma navyše prázdne všetky kanály, čo z neho vedú

- Viacero možností ako detekovať prázdnost vychádzajúcich kanálov:

  - proces dostane potvrdenie o prijatí každej poslanej správy

  - proces pošle marker a dostane potvrdenie o jeho prijatí

- u.e detects .... $<\wedge v : (u,v) \in E :: (u,v).c = null>$

Detekcia terminácie

Program

initially claim = false ☐ b = empty ☐ < ☐ u.delta = empty>

assign

{u pošle správu m cez kanál (u,v).c ak u je aktívny}

< ☐ (u,v) :: (u,v).c, u.delta := (u,v).c;m, u.delta ∪ {v} if ¬u.idle>

{v prijme správu cez kanál (u,v).c }

☐< ☐ (u,v) ::v.idle, m, (u,v).c := false, head((u,v).c;m),
tail((u,v).c;m) if (u,v).c ≠ null >

☐ < ☐ u:: u.idle := true>

☐ < ☐ u:: b, u.delta := b ∪ {u} – u.delta, empty if u.idle ∧ u.e >

☐ claim := (b=V)

end

Detekcia terminácie

Nevýhodou predchádzajúceho riešenia je že každý proces u musí udržiavať u.delta, ktoré môže byť veľké

(neprázdna) množina J detekujúcich procesov - rôzna od V

j.flag – boolovská, pre každe j z J

množinu V si rozdelia procesy z J

g(v) – člen J, ktorý má na starosti v

ak j.flag platí pre všetky j, tak všetky procesy v b sú idle

Detekcia terminácie

Program

initially claim = false □ b = empty >

assign

< □ (u,v) :: v.idle := u.idle ∧ v.idle || g(u).flag := g(u).flag ∧ u.idle >

□ < □ u:: u.idle := true>

□ < □ j:: b, j.flag := empty, true if ¬j.flag>

□ < □ u:: b := b ∪ {u} if u.idle >

□ claim := [(b=V) ∧ < ∧ j :: j.flag >

end

Byzantská dohoda 1

Zadanie:

- dva druhy procesov: spoľahlivé a nespoľahlivé
- špeciálny proces generál: jeden z nich; aj on môže byť spoľahlivý alebo nespoľahlivý
- procesy navzájom komunikujú
- každý má lokálnu premennú, sú uložené v poli *byz*: proces x má premennú $byz[x]$

Úloha:

Navrhnúť program, ktorý keď vykonávajú spoľahlivé procesy, tak všetky si nastavia rovnakú hodnotu premennej *byz*.

Naviac, ak je generál spoľahlivý, tak touto hodnotou je počiatočná hodnota generála $d^0[g]$

Byzantská dohoda 2

- Predpokladáme, že je:

  t nespoľahlivých

  > 2.t spoľahlivých

  (inak algoritmus neexistuje: napr. 1 nespoľahlivý a 2 spoľahliví: toto je aj základ pre dôkaz)

- Označenie premenných:

  u, v, w– spoľahlivý proces

  x, y, z – ľubovoľný proces

  g – proces generál

- premenné spoľahlivých procesov budeme označovať *spoľahlivé*, podobne pre nespoľahlivé

Byzantská dohoda 3

Spec1:

$$byz[u] = byz[v]$$

a ak g je spoľahlivý, tak

$$byz[u] = d^0[g]$$

Ak g je spoľahlivý, tak druhá podmienka implikuje prvú.

Obmedzíme sa na prípad, keď $d^0[g]$ je boolovská.
V prípade, že $d^0[g]$ má 2^m možných hodnôt, zakódujeme ich do m bitov a spustíme paralelne na (teraz už boolovské) zložky binárneho kódu.

Byzantská dohoda 4

Premenné:

con – postupnosť matíc (presvedčenia)

$con^r[x, y]$ – r-tá matica a jej prvok $[x, y]$

$con^r[x, y]$ – je boolovská a lokálna pre x

$d^r[x]$ – je boolovská a lokálna pre x

(hodnota x v r-tom kole dohadovania)

$con^r[u, *] = \langle + x\colon con^r[u, x] :: 1 \rangle$

Byzantská dohoda 5

Spec2:

(B1) $\langle \wedge u\colon u \neq g :: \neg d^0[u] \rangle \wedge \langle \wedge u, x :: \neg con^0[u, x] \rangle$

(A1) $con^r[u, v] = d^{r-1}[v]$

(A2) $con^{r-1}[u, x] \Rightarrow con^r[v, x]$

(E1) $d^r[u] = d^{r-1}[u] \vee (con^r[u, *] \geq r \wedge con^r[u, g])$

(E2) $byz[u] = d^{t+1}[u]$

Správnosť Spec2 (ukážeme, že Spec2 $\Rightarrow$ Spec1)

Byzantská dohoda 6

Veta: Ak g je spoľahlivý, tak platí

$$\langle \forall r\colon r \geq 1 :: d^r[u] = d^0[g] \rangle$$

Dôkaz: indukciou cez r

$r = 1$:

z (E1):	$d^1[u] = d^0[u] \vee (con^1[u, *] \geq 1 \wedge con^1[u, g])$
ale	$con^1[u, g] \Rightarrow con^1[u, *] \geq 1$
a z (A1)	$con^1[u, g] = d^0[g]$
takže	$d^1[u] = d^0[u] \vee d^0[g]$
ak	$u = g$, tak $d^1[g] = d^0[g] \vee d^0[g]$
ak	$u \neq g$, tak $d^1[u] = \text{false} \vee d^0[g] = d^0[g]$

Byzantská dohoda

$r > 1$:

$d^r[u] = d^{r-1}[u] \vee (con^r[u, *] \geq r \wedge con^r[u, g])$

z predpokladu vieme, že:

$$d^{r-1}[u] = d^0[g]$$

z (A1) $\quad con^r[u, g] = d^{r-1}[g] \; (= d^0[g])$

teda $\quad d^r[u] = d^0[g] \vee (con^r[u, *] \geq r \wedge d^0[g])$

z čoho

$$d^r[u] = d^0[g]$$

Byzantská dohoda 7

Veta: $\langle \forall u, v{:}{:}\ d^{t+1}[u] = d^{t+1}[v] \rangle$

Dôkaz: Ak g je spoľahlivý, tak platnosť vyplýva z predošlej vety. Nech teda g je nespoľahlivý.

Nech r je najmenšie také, že $d^r[u]$ platí pre nejaké u. Ak také r neexistuje, veta platí. Ukážeme, že platí $r \leq t$ a $d^{r+1}[v]$ pre $\forall$ v.

$\neg d^{r-1}[u] \wedge d^r[u]$	/* výber r a u */
$con^r[u, *] \geq r \wedge con^r[u, g]$	/* z (E1) */
$con^r[u, x] \Rightarrow con^{r+1}[v, x]$	/* z (A2) */
$\neg con^r[u, u]$	/* z (A1), $\neg d^{r-1}[u]$ */
$con^{r+1}[v, u]$	/* z (A1), $d^r[u]$ */
$con^{r+1}[v, *] > con^r[u, *]$	/* z predošlých troch */
$con^{r+1}[v, *] \geq r + 1$	/* z predošlých */
$con^{r+1}[v, g]$	/* z 2.riadku a (A2) */
$d^{r+1}[v]$	/* z predošlých dvoch */

Byzantská dohoda 8

Teraz ukážeme, že $r \leq t$, čím dostaneme $d^{r+1}[u] \Rightarrow d^{t+1}[v]$ z (E1).

Vyberali sme r tak, že bolo najmenšie, teda

$\neg d^{r-1}[w]$

$\neg con^{r}[u, w]$ /* z (A1) */

t.j. con pre spoľahlivé neplatí (v r-tom kole), *môže* to platiť teda len pre nespoľahlivé (nie nutne pre všetky), ktorých je t. Čiže

$con^{r}[u, *] \leq t$

$r \leq con^{r}[u, *] \leq t$

Byzantská dohoda 9

Vlastnosti (A1) a (A2) sa nedajú zabezpečiť pri neautorizovanom komunikovaní, preto ich musíme zjemniť.

Zavedieme nové premenné:

$obs^r[x, y]$ – of boolean (bude $obs^r[u, v]=con^r[u,v]$)

$sum^r[x, y]$ – of integer (odhad x pre koľko w platí $obs^r[w,y]$)

$val^r[x, y]$ – of boolean (odhad x hodnoty $d^r[y]$)

<u>Spec3:</u> (B1), (E1), (E2) a nasledujúce:

(B2) $\neg obs^0[u, x]$

(A3) $0 \leq sum^r[u, x] - \langle +w: obs^r[w, x]:: 1 \rangle \leq t$

(E3) $obs^{r+1}[u, x] = (obs^r[u, x] \vee sum^r[u, x] > t \vee val^r[u, x])$

(E4) $val^r[u, v] = d^r[v]$

(E5) $con^r[u, x] = sum^r[u, x] > 2.t$

Byzantská dohoda 10

Vysvetlenie:

(A3) $sum^r[u, x]$– odhad u-čka, pre koľko w platí $obs^r[w, x]$

(E4) $val^r[u, x]$ – odhad u hodnoty $d^r[x]$
(ak x je spoľahlivý, tak je to presne $d^r[x]$)

(E3) $obs^{r+1}[u, x]$ platí, ak
$obs^r[v, x]$ platí pre nejaké spoľahlivé v
$(obs^r[u, x] \vee sum^r[u, x] > t)$
alebo u odhaduje, že $d^r[x]$ platí

(E5) $con^r[u, x]$ platí, ak $obs^r[w, x]$ platí pre viac než t
spoľahlivých procesov w

Byzantská dohoda 11

<u>Správnosť Spec3</u> (ukážeme, že Spec3 $\Rightarrow$ Spec2)

Z (A3) vieme, že

(D1) $\langle\ \exists w{::}\ obs^r[w, x]\ \rangle \vee sum^r[u, x] \leq t$

(D2) $\langle\ \forall w{::}\ obs^r[w, x]\ \rangle \Rightarrow sum^r[u, x] > 2.t$

(D3) $sum^r[u, x] > 2.t \Rightarrow sum^r[v, x] > t$

Dôkaz D3:

$$sum^r[u, x] > 2.t \Rightarrow \langle\ +w{:}\ obs^r[w, x]{::}\ 1\ \rangle > t$$

$$sum^r[v, x] \geq \langle\ +w{:}\ obs^r[w, x]{::}\ 1\ \rangle > t$$

Byzantská dohoda 12

Lema 1: $\langle\ \forall r\colon r \geq 0{::}\ obs^{r+1}[u, v] = d^r[v]\ \rangle$

Dôkaz: indukciou cez r

$r = 0$

z (E3): $obs^1[u, v] = (obs^0[u, v] \vee sum^0[u, v] > t \vee val^0[u, v])$

z (B2) $(\forall w{::}\ \neg obs^0[w, v])$ a z (D1) máme:

$obs^1[u, v] = val^0[u, v]$

a z (E4)

$obs^1[u, v] = d^0[v]$

$r > 1$:

$obs^{r+1}[u, v] = (obs^r[u, v] \vee sum^r[u, v] > t \vee val^r[u, v])$

$sum^r[u, v] > t \Rightarrow \langle\ \exists w{::}\ obs^r[w, v]\ \rangle$ /* z (D1) */

$obs^r[u, x] \vee sum^r[u, x] > t \Rightarrow \langle\ \exists w{::}\ obs^r[w, v]\ \rangle$

$\langle\ \exists w{::}\ obs^r[w, v]\ \rangle \Rightarrow d^{r-1}[v]$ /* indukcia*/

$d^{r-1}[v] \Rightarrow d^r[v]$ /* z (E1) */

$val^r[u, v] = d^r[v]$ /* (E4) */

$obs^{r+1}[u, v] = d^r[v]$ /* z $\uparrow$ */

Byzantská dohoda 13

Lema 2: $\langle\ \forall r\colon r \geq 0{:}{:}\ con^r[u, v] = obs^r[u, v]\ \rangle$

Dôkaz: indukciou cez r

$r = 0$:

$sum^0[u, v] \leq t$	/* z (B2) */
$\neg con^0[u, v]$	/* z predošlého a (E5)*/

$r > 1$:

$obs^r[u, v] = d^{r-1}[v]$	/* lema 1 */
$\langle\ \forall w, w'{:}{:}\ obs^r[w, v] = obs^r[w', v]\ \rangle$	/* z ↑*/
$obs^r[u, v] = sum^r[u, v] > 2.t$	/* z ↑ a (D1), (D2) */
$obs^r[u, v] = con^r[u, v]$	/* z (E5) */

Byzantská dohoda 14

Veta: $\langle\ \forall r\colon r \geq 1 :: con^{r}[u, v] = d^{r-1}[v]\ \rangle$
Dôkaz: z liem 1 a 2

Veta: $\langle\ \forall r\colon r \geq 0 :: con^{r}[u, x] \Rightarrow con^{r+1}[v, x]\ \rangle$
Dôkaz:

nech $con^{r}[u, x]$, potom

$sum^{r}[u, v] > 2.t$	/* z (E5) */
$\langle\ \forall w :: sum^{r}[w, x] > t\ \rangle$	/* z (D3) */
$\langle\ \forall w :: obs^{r+1}[w, x]\ \rangle$	/* z (E3) */
$sum^{r+1}[v, x] > 2.t$	/* z (D2) */
$con^{r+1}[v, x]$	/* z (E5) */

Byzantská dohoda 15

Vlastnosti (B1) a (B2) sa už ľahko implementujú ako rovnice.
(E1) – (E5) už sú priamo rovnice, teraz ostáva prepísať (A3) do rovníc.

Premenné:

$robs^r[u, z, x]$ – lokálna k u, ktorá číta hodnotu $obs^r[z, x]$

$sum^r[u, x]$ – počet z takých, že je $robs^r[u, z, x]$ true

<u>Spec4:</u> (B1), (B2), (E1) – (E5) a nasledujúce:

(E6) $robs^r[u, z, x] = obs^r[z, x]$

(E7) $sum^r[u, x] = \langle +z: robs^r[u, z, x]:: 1 \rangle$

Byzantská dohoda 16

Vysvetlenie:

- proces *u* číta počet procesov *z*, pre ktoré platí $obs^r[z, x]$; výsledok si uloží do $sum^r[u, x]$
- započítajú sa všetky spoľahlivé *w*, pre ktoré platí $obs^r[w, x]$, a teda platí dolná hranica
- keďže máme *t* nespoľahlivých, hodnota $sum^r[u, x]$ nemôže prekročiť $\langle +w\colon obs^r[w, x] :: 1 \rangle$ o viac než *t*

Tým je dokázaná správnosť Spec4 (lebo Spec4 $\Rightarrow$ Spec3).

Byzantská dohoda 17

Program Byzantská dohoda

always

$\langle \square\, y \colon y \neq g :: d^{0}[y] = \text{false} \rangle$ (B1)

$\square \quad \langle \square\, y, x :: obs^{0}[y, x] = \text{false} \rangle$ (B2)

$\square \quad \langle \square\, r, y \colon 1 \leq r \leq t + 1 ::$

$\qquad d^{r}[y] = d^{r-1}[y] \vee (con^{r}[y, *] \geq r \wedge con^{r}[y, g]) \rangle$ (E1)

{ „$con^{r}[y, *]$” je skratka za „$\langle + z \colon con^{r}[y, z] :: 1 \rangle$” }

$\square \quad \langle \square\, y :: byz[y] = d^{t+1}[y] \rangle$ (E2)

$\square \quad \langle \square\, r, y, x \colon 0 \leq r \leq t + 1 ::$

$\qquad obs^{r+1}[y, x] = (obs^{r}[y, x] \vee sum^{r}[y, x] > t$

$\qquad\qquad \vee\, val^{r}[y, x])$ (E3)

$\qquad \square\, val^{r}[y, x] = d^{r}[x]$ (E4)

$\qquad \square\, con^{r}[y, x] = sum^{r}[y, x] > 2.t \rangle$ (E5)

$\square \quad \langle \square\, r, y, z, x \colon 0 \leq r \leq t + 1 :: robs^{r}[y, z, x] = obs^{r}[z, x] \rangle$ (E6)

$\square \quad \langle \square\, r, y, z, x \colon 0 \leq r \leq t + 1 ::$

$\qquad sum^{r}[y, x] = \langle +z \colon robs^{r}[y, z, x] :: 1 \rangle \rangle$ (E7)

end

Byzantská dohoda 18

Rovnice možno usporiadať v správnom poradí (splnenie požiadavky pre always-sekciu):

(B1), (B2), (E6), (E7), (E4), (E3), (E5), (E1), (E2)

Byzantská dohoda 19

Program Byzantská dohoda

initially

$\langle$ □ y: $y \neq g$ :: $d^0[y]$ = false $\rangle$

$\langle$ □$\langle$ □ y, x:: $obs^0[y, x]$, $val^0[y, x]$ = false, $d^0[x]$ $\rangle$

$\langle$ □$\langle$ □ $x,y, z,$ :: $robs^0[y, z, x] = obs^0[z, x]$ $\rangle$

$\langle$ □$\langle$ □ y, z, x: :: $sum^0[y, x] = \langle +z: robs^0[y, z, x] :: 1 \rangle \rangle$

For r = 1 to t+1

do

$\langle$ □ y, x: $obs^r[y, x] = (obs^{r-1}[y, x] \vee sum^{r-1}[y, x] > t \vee val^{r-1}[y, x])\rangle$

$\langle$ □ y, z, x: $robs^r[y, z, x] = obs^r[z, x]$ $\rangle$

$\langle$ □ y, z, x : $sum^r[y, x] = \langle +z: robs^r[y, z, x] :: 1 \rangle \rangle$

$\langle$ □ y, x: $con^r[y, x] = sum^r[y, x] > 2.t$ $\rangle$

$\langle$ □ y: $d^r[y] = d^{r-1}[y] \vee (con^r[y, *] \geq r \wedge con^r[y, g])$ $\rangle$

$\langle$□ y, x: $val^r[y, x] = d^r[x]\rangle$

od

$\langle$□ y:: $byz[y] = d^{t+1}[y]$ $\rangle$

end

Backtrack

Špecifikácia:

Dané pole kladných čísiel A[0..N-1], N >0
a kladné číslo M.

Úloha: zistiť či súčet niektorých prvok poľa A sa rovná M.

Podmnožina prvkov z A sa dá jednoznačne reprezentovať rastúcou postupnosťou indexov.

Hrubá sila - úplné prehľadávanie - vygenerujú sa všetky podmnožiny a pre každú sa zistí, či sa súčet rovná M.

Systematický spôsob generovania – lexikografické usporiadanie.

Efektívnejšie riešenie:

Ak pre postupnosť indexov seq odpovedajúci súčet prevýši M, tak žiadne riešenie nemôže obsahovať všetky indexy zo seq.

Odstránime posledný index v seq a skúmame lexikograficky nasledujúcu postupnosť.

Pre jednoduchosť predpokladajme, že:

1) Pre každé i, A[i] neprevyšuje M

2) A [i] je menšie alebo rovné ako A [i+1]

Nech v je index, ktorý uvažujeme pridať do seq.

Nech sum je súčet prvkov s indexmi v seq plus A[v].

Nech top(seq) je posledný prvok v seq, to čo ostane bude pop(seq).

Pridáme posledny prvok do A, A [N]=M.

Úlohou je zistiť, či existuje príslušná postupnosť neobsahujúca N.

Program

```
declare sum, v : integer, seq : sequence of integer
always   sum = <+ i : i ∈ seq :: A [i] >   +   A [v]
initially seq, v = null, 0
assign
seq, v := seq;v, v+1              if sum < M
          ~ pop(seq), top(seq) +1    if sum > M
end
```

Dôkaz správnosti programu

- ukážeme že vždy dosiahne pevný bod
- pre každý pevný bod platí:

v je rôzne od N iff súčet nejakej podmnožiny A[0..N-1] je rovný M.

Invariant

(seq;v) je rastúca postupnosť indexov z 0..N
a
žiadna postupnosť lexikograficky menšia ako (seq;v) nie je riešením

$FP \quad \equiv \; (sum = M)$

$\qquad \equiv \; (\langle + \; i: i \in seq :: A[i] \rangle + A[v]) = M$

Ak $v \neq N$ pre pevný bod, tak seq;v je riešenie

Problém je NP úplný.

Úloha:

Vynásobiť matice $M_{1,}\ M_{2,\ \ldots\ ,}\ M_{N,}$ N > 0

tak aby sme minimalizovali počet skalárnych operácii.

Násobenie je asocitívne - $M_1 \times (M_2 \times M_3) = (M_1 \times M_2) \times M_3$

Ak násobíme matice s dimeziou u,v a v,w

tak potrebujem u x v x w skalárnych operácií.

Výsledný počet operácií môže závisieť od poradia násobenia.

Riešenie hrubou silou – výskúšanie všetkých možných postupností násobenia. (exponenciálne riešenie)

Lepšie riešenie:

Pre N=1 je riešenie triviálne – počet skalárnych operácií je rovný 0.

Pre N > 1, posledné matičné násobenie je medzi A a B, kde A je výsledok násobenia M_1 x ... x M_k a B je výsledok násobenia M_{k+1} x ... x M_N .

Nech matica M_i má r[i-1] riadkov a r[i] stĺpcov.

Potom A má dimenziu r[0] x r[k] a B má dimenziu r[k] x r[N].

Celkový počet potrebných operácii je r[0] x r[k] x r[N] plus počet operácii potrebných na výpočet A a B.

V prípade optimálneho riešenia aj počet násobení pre A a B musí byť optimálny.

Nech g[i,j] je minimálny počet operácií potrebných pre vynásobenie M_i x ... x M_j.

```
Program
always
<▯ i : 1 ≤ i ≤ N :: g[i,i] = 0 >
▯
<▯ i,j : 1 ≤ i < j ≤ N :: g[i,j] =
                    <min k : i ≤ k < j :: g[i,k] + g[k+1,j] + r[i-1] x
r[k] x r[j]>
>
end
```

Množina rovností je proper.

Výpočet g[i,j] potrebuje na sekvenčnom počítači O(N) krokov a spolu teda $O(N^3)$ krokov

Paralelné riešenie:

Výpočet g[u,v] potrebuje hodnotu g[x,y] iba ak (v-u) > (y-x).

Naraz teda môžeme počítať g[u,v] a g[x,y] ak (v-u) = (y-x).

T.j. naraz môžeme počítať g[i, i+t] pre „všetky“ i a t.

```
Program
always
<□ i : 1 ≤ i ≤ N :: g[i,i] = 0 >
□
<□ t : 1 ≤ t < N
< || i,j : 1 ≤ i < j ≤ N ∧ j = i+t ::
g[i,j] = <min k : i ≤ k < j :: g[i,k] + g[k+1,j] + r[i-1] x r[k] x r[j]>
>
>
end
```

Pre každé t najviac O(N) premenných je definovaných.
Výpočet každej potrebuje O(log N) krokov s $O(N^2)$ procesormi
Spolu potrebujeme O(N log N) krokov s $O(N^2)$ procesormi.

Daný orientovaný graf s vrcholmi očíslovanými od 0 po N.

Daná matica E, E[u,v] je true, ak existuje hrana z u do v.

Vzdialenosť u - dist(u) – je počet hrán na ceste od 0 po u ak cesta existuje, inak ∞.

dist(0)=0

Úloha: nájsť dist(u) pre všetky vrcholy.

Prehľadávanie do šírky

Vrchol so vzdialenosťou k+1 ma predchodcu so vzdialenosťou k.

```
Program
k: ineteger
d: array[0..N] of integer
initially
k=0 || d[0]=0
|| <|| v : 0 < v ≤ N :: d[v] = ∞ >
assign
<|| v : 0 < v ≤ N :: d[v] := min(d[v],k+1)
                                   if < ∃u: 0 < u ≤ N ∧ E[u,v]:: d[u]=k>
>
|| k := k+1 if k < N
end
```

Úloha: nájsť cestu z vrcholu 0 do vrcholu dest v orientovanom grafe.

Prehľadávanie do hĺbky.

Na začiatku cesta obsahuje len 0.

Cestu predlžujeme.

Ak posledný jej vrchol nemá nasledníka, odstránime ho a skúšame preposledný vrchol.

Pre jednoduchosť predpokladáme, že každý vrchol ma aj následníka mimo cesty.

Zavedieme špeciálny vrchol N, ku ktorému vedie hrana od každého iného.

Zavedieme číselné pole p

Pre každé x, $0 \leq x \leq N$

($p[x] < 0$ => x nikdy nebolo pridané do cesty)

$\wedge$

($0 \leq p[x] < N$ => x je na ceste a

ak $x \neq 0$, p[x] je predchodca x na ceste)

$\wedge$

($N \leq p[x]$ => x bolo odstránené z cesty)

Program

declare

u,v : integer

p: array[0..N] of integer

always

$v = \langle \min j : 0 \le j \le N \wedge E[u,j] \wedge p[j] < 0 :: j \rangle$

initially

$\langle □ j : 0 \le j < N :: E[j,N] = true \rangle$

$□\ u=0\ □\ p[0]=1$ {pre N > 1 vrchol 0 je na ceste}

$□\ \langle □\ j : 0 < j \le N :: p[j] = -1 \rangle$

assign

$u,p[v] := v,u$ if $v \ne N \wedge u \ne dest$ {rozšírenie cesty}

$□\ u,p[u] := p[u], N$ if $v = N \wedge u \ne 0 \wedge u \ne dest$ {backtrack}

end

Logické programovanie

Program – množina logických implikácií

Vykonanie programu – určí, či sú implikácie konzistentné

Aby sme určili, že záver C vyplýva z množiny implikácií S, vykonáme logický program, ktorý pozsotáva z:

- S
- C => false   (neplatí C)

Ak vykonanie ukáže, že je to nekonzistentné, tak C vyplýva z S

Príklad:

Mária je pekná

Ján je príjemný, láskavý a silný

Aj je Ján bohatý alebo Mária má rada Jána, tak je Ján šťastný

Ak Ján má rád Máriu a Ján je láskavý alebo Ján je príjemný a silný, tak Mária ho má rada

Ak je Mária pekná, tak ju Ján má rád

Je Ján šťastný?

Logické programovanie

true => pekná[Mária]

true => láskavý[Ján]

true => príjemný[Ján]

true => silný[Ján]

bohatý [Ján] ∨ rád [Mária,Ján] => šťastný[Ján]

(rád [Ján, Mária] ∧ láskavý[Ján]) ∨

(príjemný[Ján] ∧ silný[Ján]) => rád[Mária, Ján]

pekná[Mária] => rád [Ján, Mária]

? šťastný [Ján]

Implikáciu

$$b \Rightarrow x$$

môžeme previesť na rovnicu

$$x = x \vee b$$

a potom hľadáme riešenie množiny rovníc.

Logické programovanie

Na začiatku sú všetky premenní false.

Riešením je hodnota premenných v FP.

Implikáciu b => false nebudeme prepisovať do UNITY – ak b je true v FP tak je to nekonzistentné

Logické programovanie

$$b \Rightarrow x$$

sa prepíše ako

$$x := x \vee b$$

Vždy to dosiahne FP: počet premenných je konečný, na začiatku sú všetky false a môžu prejsť len z false do trues

Logické programovanie

Program

initially všetky premenné false

assign

    pekná[Mária] := true

▯   láskavý[Ján] := true

▯   príjemný[Ján] := true

▯   silný[Ján] := true

šťastný[Ján] := šťastný[Ján] $\vee$ bohatý [Ján] $\vee$ rád[Mária,Ján]

rád[Mária, Ján] := rád[Mária, Ján] $\vee$ (rád [Ján, Mária] $\wedge$ láskavý[Ján])
$\vee$ (príjemný[Ján] $\wedge$ silný[Ján])

rád [Ján, Mária] := rád [Ján, Mária] $\vee$ pekná[Mária]

end

šťastný[Ján] platí v FP, teda program to implikuje

Platí aj $\neg$bohatý [Ján] ale to nemožno odvodiť (možno odvodiť len pozitívne tvrdenia)

Logické programovanie

Doteraz sme uvažovali konečný prehľadávací priestor

Uvažujeme funkcie a relácie.
Každú budem reprezentovať ako množinu, do ktorej možno len pridávať a nie uberať.

Príklad:

$f(x,y) = (x! = y)$

a nech $t(x,y,z)=(x*y=z)$

true => $f(0,1)$

$f(x,v) \wedge t(x+1,v,u) \Rightarrow f(x+1,u)$

Pridáme záver $f(n,m) \Rightarrow$ false

Logické programovanie

UNITY:

f,t .... F, T množiny

b => f(x,y) prepíšeme ako

F :=F $\cup$ {(x,y)} if b

b(x,y,z) => f(x,y) prepíšeme ako

F :=F $\cup$ < $\cup$ x,y,z: (x,y,z) $\in$B:: (x,y)>

Program može skončiť ak pre nejaké m, (n,m) $\in$ F

Program Faktoriál

declare done : boolean

always done = $\langle \exists m :: (n,m) \in F \rangle$

initially F = $\{(0,1)\}$

assign

$F := F \cup \langle \cup\, x,v,u : (x,v) \in F \wedge (x+1,v,u) \in T :: (x+1,u) \rangle$ if $\neg$done

end

Vo všeobecnosti nevieme, či to skončí.

Paralelizmus:

Inštrukcie pre paralelizmus (||, ...)

Paralelné spracovanie dát (rozdelenie dát)

„task" paralelizmus (distribuované počítanie)

Program Composition Notation

Elementárny blok:

priradenie, volanie programu napísaného v PCN, C, Pascal, Fortran, ...

Zložený blok:

{; <blok>^{1}} |,

{|| <blok>^{1}} |,

{□ <blok>^{1}} |,

{? <guard -> blok>^{1}}

Progarm-call@location

(lokácia – premenná,číslo, relatívna lokácia)

Počítače – 0, ..., n

Procesové algebry

Sekvenčné komunikujúce procesy

Communicating Sequential Processes (CSP), 1978, C. A. R. Hoare

Calculus of Communicating Systems (CCS), 1980, R. Milner

Algebra of Communicating Processes (ACP), 1982, J. Bergstra and J.W. Klop

Ambient calculus, 1998, L. Cardelli and A. D. Gordon

Pi-calculus, 1999, R. Milner, J. Parrow and D. Walker\bigskip\pause ...

časové, pravdepodobnostné, stochastické PA, iné komunikačné mechanizmy, lokality, ...

Jazyky odvodené z procesových algebier alebo využívajúcich procesové algebry

- -Wrightm (CSP),
- -Timed Communicating Object Z (Object-Z a Timed CSP),
- -Circus (CSP a Z),
- -CspCASL (CSP),
- -Ease (CSP),
- -Occam (CSP),
- -JCSP (CSP a Occam),
- -C++CSP (CSP),
- -Acute, BPML, Occam
- -Pi, Pict, JoCaml (Pi-kalkulus),
- -LOTOS (CSS a CSP),
- -mu CRL (micro Common Representation Language), (ACP plusabstraktné dátové typy),

Softvérové nástroje

- Concurrency Workbench (CCS),
- Concurrency Workbench for the New Century (CWB-NC), (CCS),
- ProBE a CSP Type checker,
- FDR (Failure Divergence Refinement) (CSP),
- CADP (Construction and Analysis of Distributed Processes) (Lotos),
- Coq - proof assistant (CCS),
- ProB (rozšítrené CSP, ...),
- Casper (CSP),
- mCRL2 (kombinácia viacerých PA)

Temporálne logiky

propozičná logika vs. logika prvého rádu
globálna vs. kompozičná
vetviaci sa čas vs. lineárny čas
časové body vs. časové intervaly
diskrétny čas vs. spojitý čas
minulosť vs. budúcnosť
distribovanosť vs. lokálnosť
....

Temporálne logiky

- časová os:
  - dvojica (S, <), kde < je úplné usporiadanie
  - izomorfná s (N, <)

- čas:
  - diskrétny
  - počiatočný okamih
  - nekonečný

- *AP*: atomické propozície (ozn. *P*, *Q*, …)

- štruktúra lineárneho času: trojica $M = (S, x, L)$, kde

  S – množina stavov

  x: N $\rightarrow$ S – postupnosť stavov

  L: $S \rightarrow$ ***P***(*AP*) – určuje pre daný stav, ktoré atomické propozície v tomto stave platia (teda ktoré *AP* sú true)

Temporálne logiky

- označenie:
  - postupnosť stavov $x = s_0, s_1, s_2, s_3, \ldots$
  - definujeme $x^i = s_i, s_{i+1}, s_{i+2}, s_{i+3}, \ldots$ (teda $x = x^0$)

- základné temporálne operátory:

  $\Diamond p$ – eventually p (raz určite p)(textovo Fp)

  $\Box p$ – vždy p (textovo Gp)

  $\bigcirc p$ – nasledujúci krát p (textovo Xp)

  $p \mathbin{\mathrm{U}} q$ – p until q (raz q začne platiť, a do vtedy platí p)

Temporálne logiky

Syntax: množina PLTL formúl je definovaná ako najmenšia množina generovaná nasledujúcimi pravidlami:

- každá *AP* (atomická propozícia) *P* je formula
- ak p, q sú formuly, tak $p \wedge q$, $\neg p$ sú formuly
- ak p, q sú formuly, tak p U q, Xp sú formuly

zavedieme označenia:

Fp = true U p $\Diamond p$
Gp = $\neg$F$\neg p$ $\Box p$
$F^{\infty}p$ = GFp (nekonečne veľa krát)
$G^{\infty}p$ = FGp (skoro všade)

Temporálne logiky

<u>Sémantika:</u>

$x \models P$ iff $P \in L(s_0)$ pre atomickú propozíciu P
$x \models p \wedge q$ iff platí $x \models p$ a $x \models q$
$x \models \neg p$ iff neplatí $x \models p$
$x \models (p \text{ U } q)$ iff $\exists j\ (x^j \models q$ and $\forall k < j$: $x^k \models p)$
$x \models \text{X}p$ iff $x^1 \models p$
$x \models \text{F}p$ iff $\exists j\ (x^j \models p)$
$x \models \text{G}p$ iff $\forall j\ (x^j \models p)$
$x \models \text{F}^\infty p$ iff $\forall k\ \exists j \geq k\ (x^j \models p)$
$x \models \text{G}^\infty p$ iff $\exists k\ \forall j > k\ (x^j \models p)$

<u>Príklady:</u>

$p \Rightarrow \text{F}q$	ak p platí teraz, tak raz bude platiť q
$\text{G}(p \Rightarrow \text{F}q)$	vždy keď platí p, tak raz začne platiť aj q
$\text{p} \wedge \text{G}(p \Rightarrow \text{X}p) \Rightarrow \text{G}p$	temporálna formulácia indukcie

Temporálne logiky

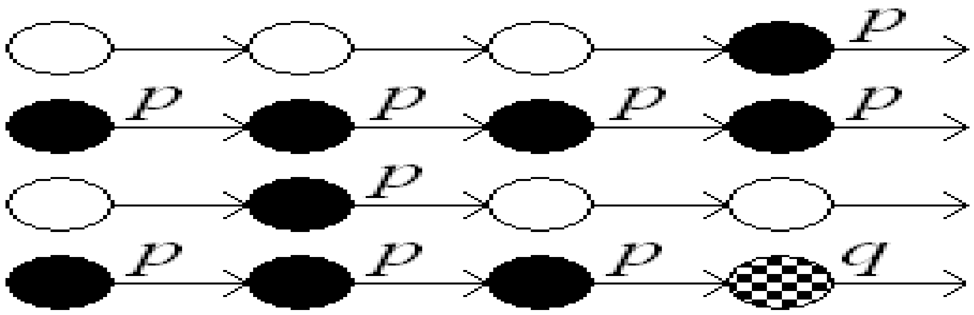
Príklady použitia operátorov F, G, X a U v propozičnej lineárnej temporálnej logiky

F*p*

G*p*

X*p*

p U *q*



Temporálne logiky

Variácie

Slabé until

Silné p U q sa zvykne označovať ako p U_s q alebo p $U_\exists$ q

$x \models p\ U_\forall\ q$ iff $\forall\ j\ ((\forall k \leq j,\ x^k \models \neg q) \Rightarrow x^j \models p)$

t.j. môže sa stať, že q nebude platiť nikdy

p $U_\exists$ q ..... p $U_\forall$ q $\wedge$ Fq

p $U_\forall$ q .... p $U_\exists$ q $\vee$ Gp

Temporálne logiky

Miesto (N, $<$) zoberiem len podmnožinu I
(môže byť aj konečná)

Gp	pre všetky nasledujúce stavy v I platí p
Fp	pre nejaký stav v I platí p
$X_\forall p$	(weak nextime) ak existuje následný stav v I tak v ňom platí p
$X_\exists p$	(strong nextime) existuje následný stav v I a v ňom platí p

$X_\exists p$ ..... $\neg X_\forall \neg p$

$X_\forall p$ ..... $\neg X_\exists \neg p$

Temporálne logiky

$G^{-}p$	vždy v minulosti platilo p
$F^{-}p$	niekedy v minulosti platilo p
$X^{-}p$	naposledy platilo p
$p\ U^{-}q$	niekedy platilo q a odvtedy platí q

Temporálne logiky

Branching (time) tempopral logic

- temporálna štruktúra: trojica $M = (S, R, L)$, kde
  S – množina stavov
  $R \subseteq S \times S$ taká, že $\forall s \in S\ \exists t \in S\ (s, t) \in R$
  $L: S \to \boldsymbol{P}(AP)$ – určuje pre daný stav, ktoré atomické propozície v tomto stave platia (teda ktoré AP sú true)

- M možno chápať ako značkovaný orientovaný graf s vrcholmi S, hranami danými R a vrcholy majú značky dané L

- Hovoríme, že M je
  - *acyklický*, ak nemá orientované cykly
  - *stromová štruktúra*, ak každý vrchol má nanajvýš jedného predchodcu
  - *strom*, ak je stromová štruktúra a má koreň
- označenie:
  - plná cesta $x = (s_0, s_1, s_2, s_3, \ldots)$: pre $\forall i$: $(s_i, s_{i+1}) \in R$
  - definujeme $x^i = (s_i, s_{i+1}, s_{i+2}, s_{i+3}, \ldots)$

Temporálne logiky

- CTL (Computational Tree Logic)
- CTL^* (Full Branching Time Logic) – silnejšia ako CTL, historicky mladšia; najprv popíšeme CTL^*

- Syntax: (*stavové* a *path* („cestové“) formuly)

  - každá atomická propozícia je stavová formula (S1)
  - ak p, q sú stavové formuly, tak $p \wedge q$, $\neg p$ sú stavové formuly (S2)
  - ak p je path formula, tak Ep, Ap sú stavové formuly (S3)

  - každá stavová formula je aj path formula (P1)
  - ak p, q sú path formuly, tak potom aj $p \wedge q$, $\neg p$ sú path formuly (P2)
  - ak p, q sú path formuly, tak potom aj p U q, Xp sú path formu (P3)

Temporálne logiky

CTL* tvoria stavové formuly

CTL tvoria pravidlá (S1), (S2), (S3) a pravidlo (P0):

ak *p*, *q* sú stavové formuly, tak *p* U *q*, X*p* sú stavové formuly (P0)

CTL operátory:

A – pre všetky budúcnosti
E – existuje budúcnosť

za A alebo E vždy nasleduje jeden z operátorov G, F, X, U
dá sa ukázať, že CTL* má väčšiu vyjadrovaciu silu než CTL

Temporálne logiky

Sémantika: (pre CTL*)

(S1) $M, s_0 \models p$ iff $p \in L(s_0)$

(S2) $M, s_0 \models p \wedge q$ iff platí $M, s_0 \models p$ a $M, s_0 \models q$

$M, s_0 \models \neg p$ iff neplatí $M, s_0 \models p$

(S3) $M, s_0 \models \mathrm{E}p$ iff $\exists x$ v M tak, že $M, x \models p$

$M, s_0 \models \mathrm{A}p$ iff pre $\forall x$ v M platí $M, x \models p$

(P1) $M, x \models p$ iff $M, s_0 \models p$

(P2) $M, x \models p \wedge q$ iff platí $M, x \models p$ and $M, x \models q$

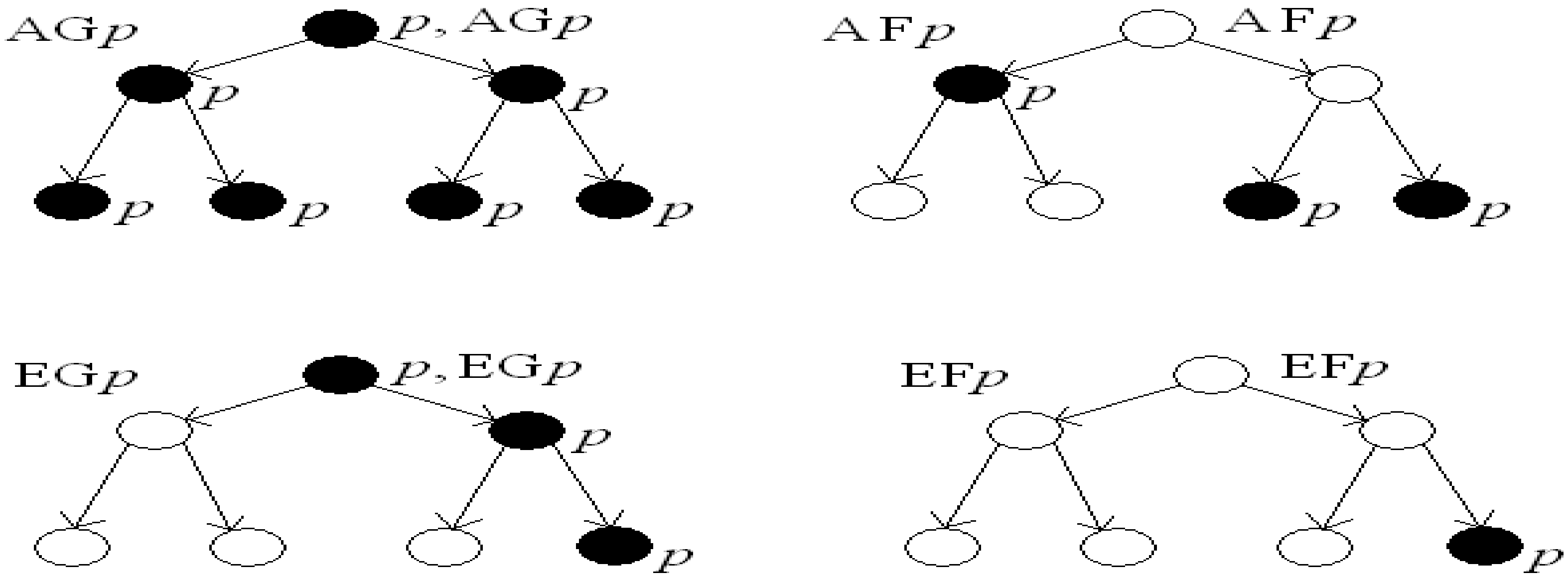
$M, x \models \neg p$ iff neplatí $M, x \models p$

(P3) $M, x \models (p \mathrm{U} q)$ iff $\exists j$ ($M, x^j \models q$ a $\forall k$ ($k < j$ implikuje $M, x^k \models p$)

$M, x \models \mathrm{X}p$ iff $M, x^1 \models p$

Temporálne logiky

Príklady použitia operátorov A, E v propozičnej branching temporal logic (značky sú pri vrcholoch; vrchol je vyplnený, ak má značku *p*)



Temporálne logiky

- množina schém axióm
- množina odvodzovacích (inferenčných) pravidiel

formula *p* je *dokázateľná* (zapisujeme |- *p*), ak pre ňu existuje *dôkaz*, t.j. konečná postupnosť formúl taká, že na jej konci je *p* a každá formula v nej je buď prípad axiómy alebo vyplýva z predchádzajúcich použitím nejakého odvodzovacieho pravidla

Temporálne logiky

<u>Schémy axióm:</u>

tautológie propozičnej logiky	(Ax1)
$\mathrm{EF}p \equiv \mathrm{E}(\mathrm{true}\ \mathrm{U}\ p)$	(Ax2)
$\mathrm{AG}p \equiv \neg\mathrm{EF}\neg p$	(Ax2′)
$\mathrm{AF}p \equiv \mathrm{A}(\mathrm{true}\ \mathrm{U}\ p)$	(Ax3)
$\mathrm{EG}p \equiv \neg\mathrm{AF}\neg p$	(Ax3′)
$\mathrm{EX}(p \vee q) \equiv \mathrm{EX}p \vee \mathrm{EX}q$	(Ax4)
$\mathrm{AX}p \equiv \neg\mathrm{EX}\neg p$	(Ax5)
$\mathrm{E}(p\ \mathrm{U}\ q) \equiv q \vee (p \wedge \mathrm{EXE}(p\ \mathrm{U}\ q))$	(Ax6)
$\mathrm{A}(p\ \mathrm{U}\ q) \equiv q \vee (p \wedge \mathrm{AXA}(p\ \mathrm{U}\ q))$	(Ax7)
$\mathrm{EX}\ \mathrm{true} \wedge \mathrm{AX}\ \mathrm{true}$	(Ax8)
$\mathrm{AG}(r \Rightarrow (\neg q \wedge \mathrm{EX}r)) \Rightarrow (r \Rightarrow \neg\mathrm{A}(p\ \mathrm{U}\ q))$	(Ax9)
$\mathrm{AG}(r \Rightarrow (\neg q \wedge \mathrm{EX}r)) \Rightarrow (r \Rightarrow \neg\mathrm{AF}q)$	(Ax9′)
$\mathrm{AG}(r \Rightarrow (\neg q \wedge (p \Rightarrow \mathrm{AX}r))) \Rightarrow (r \Rightarrow \neg\mathrm{E}(p\ \mathrm{U}\ q))$	(Ax10)
$\mathrm{AG}(r \Rightarrow (\neg q \wedge \mathrm{AX}r)) \Rightarrow (r \Rightarrow \neg\mathrm{EF}q)$	(Ax10′)
$\mathrm{AG}(p \Rightarrow q) \Rightarrow (\mathrm{EX}p \Rightarrow \mathrm{EX}q)$	(Ax11)

Temporálne logiky

Odvodzovacie pravidlá:

ak |- *p*, tak potom |- AG*p* (R1, zovšeobecnenie)

ak |- *p* a |- $p \Rightarrow q$, tak potom |- *q* (R2, modus ponens)

Veta: Deduktívny systém s axiómami (Ax1)–(Ax11) a odvodzovacími pravidlami (R1), (R2) je sound (zdravý, korektný) and complete (úplný) pre CTL.

Vzťah CTL a MU-kalkulus

$A\,(p\ U\ q)$ .... $\mu Z\ \ q \vee (p \wedge (\langle - \rangle tt \wedge [-]Z))$

$E\,(p\ U\ q)$ ..... $\mu Z\ \ q \vee (p \wedge \langle - \rangle Z)$

$AF\ p$ .... $\mu Z\ \ p \vee (\langle - \rangle tt \wedge [-]Z)$

$EF\ p$ .... $\mu Z\ \ p \vee \langle - \rangle Z$

Verifikácia vlastností systémov vyjadrených pomocou logických formúl

1.theorem proving

axiomy logiky+vlastnosti systému/programu |- theorema

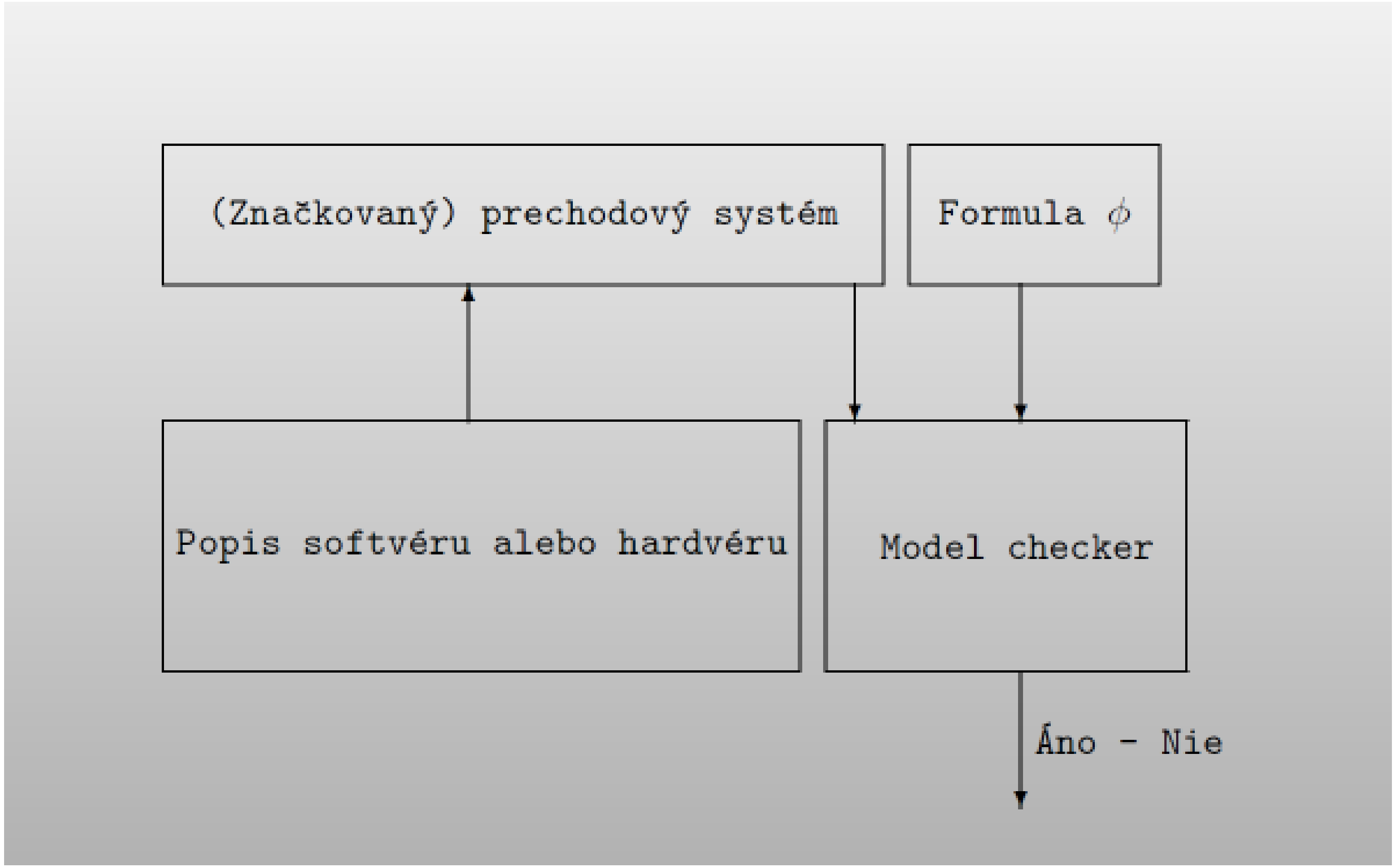
- automatický alebo poloautomatický theorem proving alebo proof verification

- v závislosti od odpovedajúcej logiky sa zložitosť riešenia pohybuje od triviálneho až po nemožné.

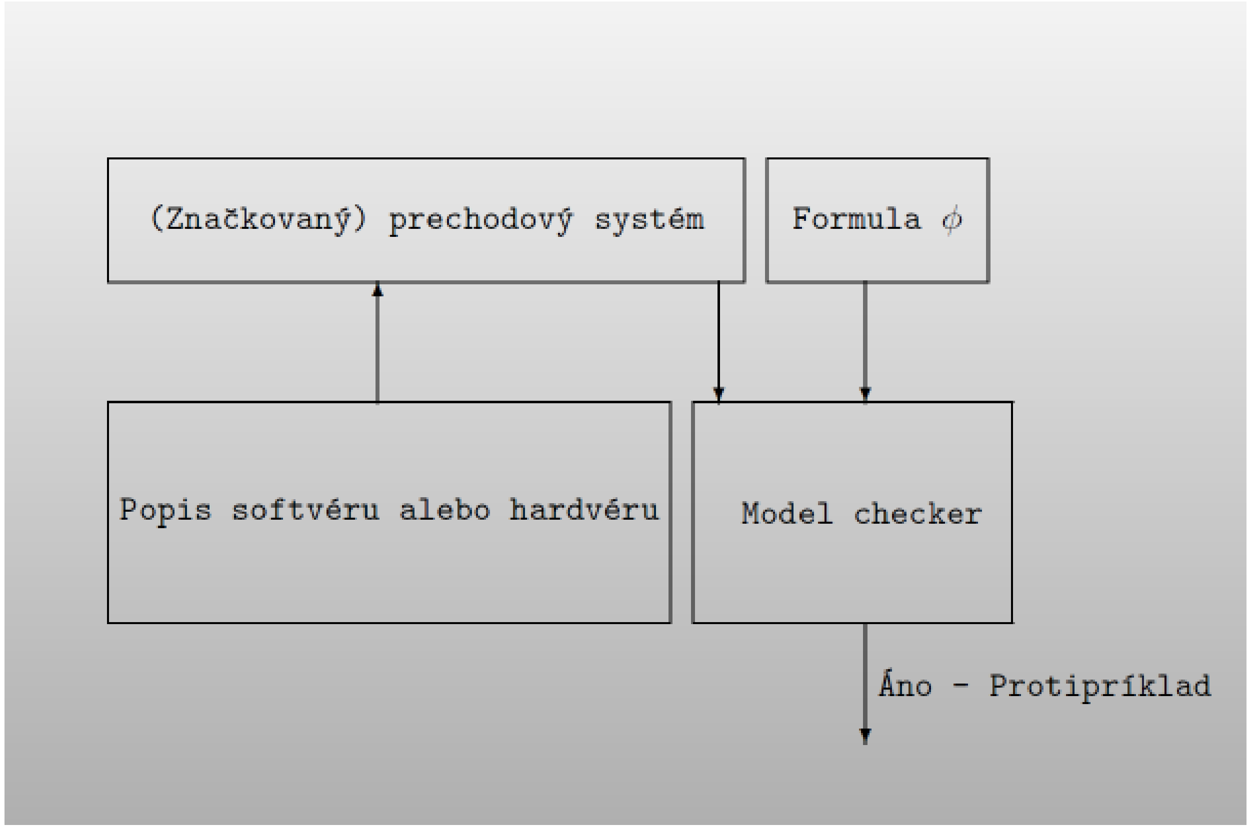
Praktické využitie: návrh a verifikácia integrovaných obvodov, (verifikácia aritmetických operácií, ...) Existuje množstvo softvérových nástrojov.

2. model checking

Model Checking



Model Checking



Úloha: daný (značkovaný) prechodový systém M (Kripkeho štruktúra) a formula temporálnej alebo modálnej logiky φ, úlohou je nájsť všetky stavy M také, že

$M,s \models \varphi$

Otcovia zakladatelia E. M. Clarke, E. A. Emerson, and J. Sifakis (roku 2007 Turingová cena za ich práce v tejto oblasti)

Model checking problem – prehľadávanie grafu - vrcholy = stavy

"state explosion" problém

- symbolické algoritmy nevytvárať graf explicitne , ale reprezentovať ho implicitne napr. pomocou formúl propozičnej logiky

- ohraničený model checking algoritmy - vytvorenie ZPS je obmedzene na k krokov a skúma sa, či neprestane platiť formula, vhodné pre niektoré modely

- "partial order reduction" redukuje sa počet skúmaných prelínaní konkurentných procesov - nemá význam skúmať zvlášť všetky možnosti ak neovplyvňujú platnosť formuly

- abstrakcia (abstract interpretation) - zjednodušenie modelu – ten spravidla nespĺňa rovnaké vlastnosti, ďalšie zjemňovanie je možné, zdravá abstrakcia - vlastnosti vyhovujú aj pôvodnému systému, úplnosť spravidla neplatí

- protipríkladom riadená abstrakcia - vytvoríme abstraktnejší modela keď nájdeme protipríklad tak zisťujeme či odpovedá modelu alebo vznikol zlou abstrakciou, v prvom prípade výsledok beriem ako výsledok. Ak nenájdeme protipríklad zjemníme model- ....

Temporálne logiky

Model Checking

daná štruktúra M a formula p
Úloha: zistiť, či M je modelom pre p

Branching–Time–Logic–Model–Checking

daná konečná štruktúra $M = (S, R, L)$ a BTL formula p

Úloha: pre každý stav $s \in S$ určiť, či platí $M, s \models p$ a ak áno, s sa označí značkou "p"

Temporálne logiky

Príklad: CTL model checking prer AFp:

predpokladajme, že už máme vrcholy, pre ktoré platí p už "označené"

ideme označovať tie, pre ktoré platí AF*p*:

1. ak je vrchol označený *p*, tak ho označíme aj AF*p*

2. (opakuje sa, kým sa dá) označ vrchol s AF*p*, ak všetci jeho následníci sú označení AF*p*
3. označ $\neg$AF*p* tie, ktoré nie sú označené AF*p*