

# Úvod do paralelného programovania

- Damas Gruska, I20,  
damas.gruska@gmail.com  
[www.ii.fmph.uniba.sk/~gruska](http://www.ii.fmph.uniba.sk/~gruska)

# Organizácia kurzu

Organizácia kurzu :

Prednášky nájdete na tejto stránke, vždy v deň, kedy sú v rozvrhu.

Budete dostávať domáce úlohy, ich odovzdanie je povinné v termíne do nasledovnej prednášky, do 13.10 hod. Toleruje sa max jedna úloha odovzdaná po termíne.

Riešenia mi posielajte ako jedno pdf-ko (ak nebude uvedené inak), klúdne to môže byť aj rukou písané, na adresu

[damas.gruska@gmail.com](mailto:damas.gruska@gmail.com)

Úlohy nebudem známkovať.

Cvičenia budú pre tých, ktorí mali problém s úlohou alebo si nie sú istí, či majú dobre vyriešenú úlohu. Budú sa konať po prednáške.

V prípade nápadnej podobnosti odovzdaných úloh, ďalšie budem hodnotiť známku Fx.

Počas semestra budete písať dva testy. Prvý zhruba v polovici semestra a druhý na poslednej prednáške. Hodnotenie týchto testov bude základ pre hodnotenie kurzu. Ak s ním nebudete súhlasiť, môžete si známku opraviť. Kto testy z nejakých dôvodov nebude môcť písať, bude ich musieť dopísať dodatočne.

Samotné slidy nie sú určené na samoštúdium!!!

## TOP 10 Sites for November 2020

For more information about the sites and systems in the list, click on the links or view the complete list.

<a href="#">1-100</a>	<a href="#">101-200</a>	<a href="#">201-300</a>	<a href="#">301-400</a>	<a href="#">401-500</a>
-----------------------	-------------------------	-------------------------	-------------------------	-------------------------

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	<b>Supercomputer Fugaku</b> - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	<b>Summit</b> - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	<b>Sierra</b> - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	<b>Sunway TaihuLight</b> - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	<b>Selene</b> - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	<b>Tianhe-2A</b> - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	<b>JUWELS Booster Module</b> - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	<b>HPC5</b> - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252

# Distribúované 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 KASYO

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  
P.S. Pacheci: *An Introduction to Parallel Programming*, Elsevier, 2011  
M.J. Sottile: *An Introduction to Concurrency in Programming Languages*, CRC Press, 2009  
A. Udaya Shankar, *Distributed Programming*, Springer, 2013,  
R. Harper, *Practical Foundation for Programming Languages*, Cambridge University Press, 2016

# 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 (synchronná, 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 byť 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 priradovací 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 funkcie  $f, g, h$  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:
  - $r = \min\{ t \mid \text{com}(t) \}$
- 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  $r := \min\{ u \mid 0 \leq u \leq z \wedge \text{com}(u) \}$   
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  $r := f(r) \square r := g(r) \square r := h(r)$

end {P2}

*Dôkaz správnosti:*

- 1: invariant  $(0 \leq r) \wedge \forall u(0 \leq u < r \Rightarrow \neg \text{com}(u))$   
(z toho vieme, že  $r \leq z$ )  
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  $\equiv \text{com}(r)$



## 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  $\neg \text{FP} \wedge r = K$  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  $\neg \text{FP} \wedge r = K \equiv K < f(K) \vee K < g(K) \vee K < h(K)$ ;
  - 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

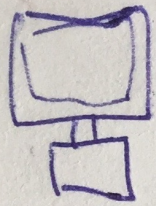
správnosť týchto prístupov aj programu P3 plynie z dôkazu správnosti pre P2

initially

$x := 1 \square y := 2$

assign

$P_1 \square P_2 \square \dots \square P_5$



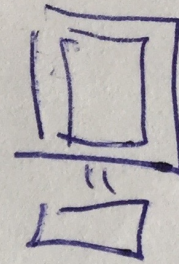
$x := 1;$   
 $y := 2$   
while true  
do  $P_1$



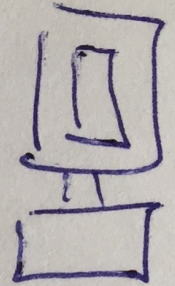
$x := 1$   
 $y := 2$   
while true  
do  $P_2$



$P_3$



~~$P_4$~~



$P_5$

# 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)

**declare\_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 priradovacích 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

$x, y, z := 0, 1, 2$

$x, y := 0, 1 \parallel z := 2$

$\langle \parallel j: 0 \leq j \leq N :: A[j] := B[j] \rangle$

znamená  $A[0] := B[0] \parallel \dots \parallel A[N] := B[N]$

$x := -1 \text{ if } y < 0 \sim 0 \text{ if } y = 0 \sim 1 \text{ if } y > 0$

# UNITY Program Structure 3

Structure of Assignment Statement (`assign_stat`)

`assign_stat` → `assign_comp` { || `assign_comp` }

Assignment component (`assign_comp`)

`assign_comp` → `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 (synchronne) 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** →  $\langle \mid \mid \text{quant assign\_stat} \rangle$

Quantification (**quant**)

**quant** → **variable\_list**: **bool\_expr** ::

premenné z **variable\_list** sa nazývajú *viazané*

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

„prípado 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 „prípado“, „prípado“ 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  $A[j]$

$\langle \parallel j: 0 \leq j \leq N :: A[j] := \max(A[j], B[j]) \rangle$

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

$\langle \parallel j, k: 0 \leq j \leq N \wedge 0 \leq k \leq N :: U[j, k] := 0 \text{ if } j \neq k \sim 1 \text{ if } j = k \rangle$

$\langle \parallel j, k: 0 \leq j \leq N \wedge 0 \leq k \leq N \wedge j \neq k :: U[j, k] := 0 \rangle \parallel \langle \parallel j: 0 \leq j \leq N :: U[j, j] := 1 \rangle$

$\langle \parallel j: 0 \leq j \leq N :: U[j, j] := 1 \parallel \langle \parallel k: 0 \leq k \leq N \wedge j \neq k :: U[j, k] := 0 \rangle \rangle$

# 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 netvorí počas výpočtu

# UNITY Program Structure 6: Assign section

Príklady:

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

$(N + 1)^2$  statements:

$\langle \forall j, k: 0 \leq j \leq N \wedge 0 \leq k \leq N :: U[j, k] := 0 \text{ if } j \neq k \sim 1 \text{ if } j = k \rangle$

2 statements:

$\langle \forall j, k: 0 \leq j \leq N \wedge 0 \leq k \leq N \wedge j \neq k :: U[j, k] := 0 \rangle$   
 $\square \langle \forall j: 0 \leq j \leq N :: U[j, j] := 1 \rangle$

$N + 1$  statement\_lists, každý z nich má 2 statements

$\langle \forall j: 0 \leq j \leq N :: U[j, j] := 1 \square \langle \forall k: 0 \leq k \leq N \wedge j \neq k :: U[j, k] := 0 \rangle \rangle$

# UNITY Program Structure 7: Initially section

syntax rovnaká ako **assign\_section**, namiesto symbolu :=  
sa používa =

definuje iníciaľne 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 iníciaľne hodnoty sú  
dobře definované

# UNITY Program Structure 7: Initially section

initially\_section definuje *inicial condition*; je to najsilnejší predikát, ktorý platí na začiatku  
získa sa nahradením  $\parallel$  za  $\wedge$  a podmienené výrazy tvaru

$x = e_0$  if  $b_0 \sim \dots \sim e_n$  if  $b_n$  výrazom  
 $(b_0 \Rightarrow (x = e_0)) \wedge \dots \wedge (b_n \Rightarrow (x = e_n))$

Čo *nie* je ekvivalentné s

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

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  $N = 3 \langle || k: 0 \leq k < N :: A[N - k] = k \rangle$
- preloženie počiatočnej podmienky pre

$$\langle \exists j, k: 0 \leq j \leq N \wedge 0 \leq k \leq N :: U[j, k] = 0 \text{ if } j \neq k \sim 1 \\ \text{if } j = k \rangle$$

vyzerá nasledovne

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

# Sorting 1

Program sort1

assign

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

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

end{sort1}

Program sort2

assign

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

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

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

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

end{sort2}

## Sorting 2

Program sort3

assign

$\langle \forall k: 0 \leq k \leq 1 ::$

$\langle \forall j: 0 \leq j \leq N \wedge (k = j \bmod 2) ::$

$A[j], A[j + 1] := A[j + 1], A[j]$  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

$\langle n: 0 \leq n < N ::$

$c[n, 0] := 1 \parallel c[n, n] := 1$

$\parallel \langle k: 0 < k < n ::$

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

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

$x := \langle \min j: 0 \leq j \leq N :: A[j] \rangle$

## Quantified Expression

$\text{expr} \rightarrow \langle \text{op quant expr} \rangle$

$\text{op} \rightarrow \min \mid \max \mid + \mid \times \mid \wedge \mid \vee \mid \equiv \mid \dots$

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

*neutrálne prvky:*

min		max		+		×		∧		∨		≡	
∞		-∞		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**
- viacero výrokov v hypotéze znamená konjunkciu
- viacero výrokov v závere znamená disjunkciu

# Basic Concepts

hypotéza

-----

záver

-----  
{ $p$ } s {true}

-----  
{false} s { $q$ }

{ $p$ } s {false}

-----

$\neg p$



# Basic Concepts

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

---

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

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

---

$$\{p'\} s \{q'\}$$

# Dokazovanie tvrdení o priradovacích príkazoch

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

1. V  $q$  nahradíme za  $x$  výraz  $E$  (výsledok označíme  $q^x_E$ )

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

2. Ukážeme, že  $p \Rightarrow q^x_E$        $x < 2 \Rightarrow x + 3 < 10$

ak  $E = e_0$  if  $b_0 \sim \dots \sim e_n$  if  $b_n$  tak výraz  $q^x_E$  bude takýto:

$(b_0 \Rightarrow q^x_{e_0}) \wedge \dots \wedge (b_n \Rightarrow q^x_{e_n}) \wedge ((\neg b_0 \wedge \dots \wedge \neg b_n) \Rightarrow q)$

môže sa písať aj v tvare

$\{p \wedge b_0\} x := e_0 \{q\}$

...

$\{p \wedge b_n\} x := e_n \{q\}$

$\{p \wedge (\neg b_0 \wedge \dots \wedge \neg b_n) \Rightarrow q\}$

- Poznámka.
  - Znak  $\Rightarrow$  budeme používať ako označenie logickej implikácie.
  - Znak  $\equiv$  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.  $\langle \forall s: s \text{ in } F :: \{p\} s \{q\} \rangle$
2.  $\langle \exists s: s \text{ in } F :: \{p\} s \{q\} \rangle$

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 všech premenných
  - label – příkaz vykonaný v  $j$ -tom kroku

# Výpočtový model

- $R_0.state$  – iníciačný stav (ak nie sú dané iníciačné 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  $\{ R_k.label \mid 0 \leq k < j \}$  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  
 $\forall R \forall j: (p[R_j] \wedge R_j.label = s) \Rightarrow q[R_{j+1}]$

# 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  
 $x = k$  unless  $x > k$   
znamená  $\langle \forall k :: x = k$  unless  $x > k \rangle$

# Unless

(daný je program F)

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

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:  $p[R_j] \Rightarrow$

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

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

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



# 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

- $\text{inch} \wedge \neg \text{rcvd}$  unless  $\neg \text{inch} \wedge \text{rcvd}$

# 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  $\text{true}$  a vice versa.

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

*Dôkaz:*

$\neg q \equiv \text{true}$	<i>/* substitučná axióma */</i>
$p$ unless $q$	<i>/* predpoklad */</i>
$p$ unless $\text{false}$	<i>/* z predchádzajúcich */</i>
$p$ is stable	<i>/* z definície */</i>

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)

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

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:

$$\begin{aligned} & p[R_j] \Rightarrow \\ & \langle \exists m: m \geq j :: q[R_m] \rangle \wedge && \text{(niekedy platí } q \text{)} \\ & \langle \forall k: j \leq k < m :: (p \wedge \neg q)[R_k] \rangle && \text{(dovtedy platí } p \wedge \neg q \text{)} \end{aligned}$$

# Ensures - príklady

1.  $x$  je neklesajúca a raz stúpne

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

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

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

2. Program P:

$x := 0$  if  $x < 0$     $\square$     $x := 0$  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  $q$ “ ( $p \rightarrow q$ )

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

1.  $p$  ensures  $q$  /  $p \rightarrow q$
2.  $p \rightarrow q, q \rightarrow r$  /  $p \rightarrow r$  /\* tranzitivita \*/
3.  $\langle \forall m: m \in W :: p(m) \rightarrow q \rangle$

-----

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

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:  
 $p[R_j] \Rightarrow \langle \exists m: m \geq j :: q[R_m] \rangle$  /\*  $q$  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 ≠ 0 → x = 0".

x ≠ 0 ensures x ≥ 0

/\* z programu \*/

x ≠ 0 → x ≥ 0

/\* z predchádzajúceho \*/

x ≥ 0 ensures x = 0

/\* z programu \*/

x ≥ 0 → x = 0

/\* z predchádzajúceho \*/

x ≠ 0 → x = 0

/\* z tranzitivity \*/

# Pevný bod

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

definujme predikát FP pre program G:

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

Príklady

1.  $k := k + 1$

$FP \equiv k = k + 1 \equiv \text{false}$

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

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

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

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

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

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

# Vlastnosti unless 1

- Reflexívnosť a antireflexívnosť:

$p$  unless  $p$

$p$  unless  $\neg p$

Dôkaz:  $\{\text{false}\} S \{p\}$ ,  $\{p\} S \{\text{true}\}$  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

- Disjunkcia

$p$  unless  $q, p'$  unless  $q' /$

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

- Konjunkcia

$p$  unless  $q, p'$  unless  $q' /$

$/ (p \wedge p') \text{ unless } (p \wedge q') \vee (p' \wedge q) \vee (q \wedge q')$

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 disjunkcia

$$p \text{ unless } q, p' \text{ unless } q' / p \vee p' \text{ unless } q \vee q'$$

Jednoduchá konjunkcia

$$p \text{ unless } q, p' \text{ unless } q' / p \wedge p' \text{ unless } q \vee q'$$

Dôkaz: z predošlých viet a zoslabenia

“Tranzitivita”

$$p \text{ unless } q, q \text{ unless } r / p \vee q \text{ unless } r$$

Dôkaz: disjunkcia a zoslabenie

## Vlastnosti unless 4

- Dôsledok 1.

$p \Rightarrow q / p \text{ unless } q$

Dôkaz: zoslabením. Podľa reflexívnosti  $p \text{ unless } p$ ,  
z predpokladu  $p \Rightarrow q$  dostaneme  $p \text{ unless } q$

- Dôsledok 2.

$\neg p \Rightarrow q / p \text{ unless } q$

Dôkaz: zoslabením. Podľa antireflexívnosti  $p \text{ unless } \neg p$ ,  
z predpokladu  $\neg p \Rightarrow q$  dostaneme  $p \text{ unless } q$

# 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 konjunkcia:  $p \wedge \neg q$  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.

$$p \vee q \text{ unless } r / p \text{ unless } q \vee r$$

Dôkaz:

$p \vee q \text{ unless } r$	<i>/* predpoklad */</i>
$\neg q \text{ unless } q$	<i>/* antireflexívnosť */</i>
$p \wedge \neg q \text{ unless } q \vee r$	<i>/* jednoduchá konjunkcia */</i>
$p \text{ unless } q \vee r$	<i>/* z dôsledku 3. */</i>

Dôsledok 5.

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

Dôkaz: indukciou, IP ( $N = 2$ ) použitím konjunkcie



# 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  $\{\text{false}\}$   
teda  $p \equiv \text{false}$

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

Disjunkcia

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

Dôkaz: z definície

## Vlastnosti ensures 3

Dôsledok 1.

$$p \Rightarrow q / p \text{ ensures } q$$

Dôkaz:  $p$  unless  $q$  a zoslabenie

Dôsledok 2.

$$p \vee q \text{ ensures } r / p \text{ ensures } q \vee r$$

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.

$$p \text{ ensures } q \vee r / p \wedge \neg q \text{ ensures } q \vee r$$

Dôkaz:

$$p \text{ ensures } q \vee r$$

/\* predpoklad \*/

$$(p \wedge q) \vee (p \wedge \neg q) \text{ ensures } q \vee r$$

/\* rozpísaný predpoklad \*/

$$p \wedge \neg q \text{ ensures } (p \wedge q) \vee (q \vee r)$$

/\* z dôsledku 2 \*/

$$p \wedge \neg q \text{ ensures } q \vee r$$

/\* z predchádzajúceho zoslabením \*/

# Vlastnosti leads-to 1

Technika dôkazov: indukcia vzhľadom na počet odvození (dĺžka dôkazu) = *štruktúrálna indukcia*

Nemožnosť:

$$p \rightarrow \text{false} / \neg p$$

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 odvodu  $p \rightarrow \text{false}$

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

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

## Vlastnosti leads-to 2

Kedže  $\neg r$  z  $p \rightarrow r$  dostaneme  $p \rightarrow \text{false}$ ,

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

2.Prípád  $\langle \forall m: m \in W :: p'(m) \rightarrow \text{false} \rangle$

-----  
□  $\langle \exists m: m \in W :: p'(m) \rangle \rightarrow \text{false}$

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

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

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:

$$p \Rightarrow q \quad /* \text{ predpoklad } */,$$

$$p \text{ ensures } q \quad /* \text{ z predpokladu } */$$

$$p \rightarrow q \quad /* \text{ z definície leads-to } */$$

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

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

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

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

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

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

## Vlastnosti leads-to 4

Cancellation

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

Dôkaz:

$$b \rightarrow r \quad /* \text{predpoklad} */$$

$$q \rightarrow q \quad /* \text{triviálne} */$$

$$q \vee b \rightarrow q \vee r \quad /* \text{disjunkcia predchádzajúcich dvoch} */$$

$$p \rightarrow q \vee r \quad /* \text{z predpokladu } p \rightarrow q \vee b \text{ a predchádzajúceho} */$$



## Vlastnosti leads-to 5

Progress-Safety-Progress (PSP) veta:

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

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):

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

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

- $r$  unless  $b$

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

- z disjunkcie na (c):

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

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

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

## Vlastnosti leads-to 7

Completion veta:

pre množinu predikátov  $p_j, q_j, 0 \leq j < N$ :  
 $\langle \forall j :: p_j \rightarrow q_j \vee b \rangle, \langle \forall j :: q_j \text{ unless } b \rangle /$   
 $/ \langle \wedge j :: p_j \rangle \rightarrow \langle \wedge j :: q_j \rangle \vee b$

Dôsledok 1 (konečná disjunkcia).

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

Dôkaz: špeciálny prípad všeobecnej disjunkcie

Dôsledok 2.

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

Dôkaz: vyplýva z dôsledku 1

Dôsledok 3.

$p \rightarrow q, r \text{ is stable} / p \wedge r \rightarrow q \wedge r$

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

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

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:  $p \rightarrow q / FP \Rightarrow (p \Rightarrow q)$

Dôkaz:

$FP \wedge \neg q$  is stable

/\* teoréma o FP \*/

(t.j.  $FP \wedge \neg q$  unless false)

$p \rightarrow q$

/\* predpoklad \*/

$p \wedge FP \wedge \neg q \rightarrow \text{false}$

/\* PSP teoréma \*/

$\neg(p \wedge FP \wedge \neg q)$

/\* nemožnosť leads-to \*/

$FP \Rightarrow (p \Rightarrow q)$

# Spojenie programov 1

nech  $F, G$  sú dva UNITY-programy

spojenie programov  $F$  a  $G$ , označenie  $F \parallel G$

- 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  $F \parallel G =$

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

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

$[p$  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  $F \parallel G$ ) = (FP of  $F$ )  $\wedge$  (FP of  $G$ )

Dôkaz:

3. Vyplýva priamo z definície

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

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

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

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

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

## Spojenie programov 3

$$\begin{aligned}
 & 2. \ p \text{ ensures } q \text{ in } F \Box G \\
 & = \ p \text{ unless } q \text{ in } F \Box G \wedge \langle \exists s: s \text{ in } F \Box G :: \{p \wedge \neg q\} s \{q\} \rangle \\
 & = \ p \text{ unless } q \text{ in } F \Box G \wedge \\
 & \quad [\langle \exists s: s \text{ in } F :: \{p \wedge \neg q\} s \{q\} \rangle \vee \\
 & \quad \langle \exists s: s \text{ in } F :: \{p \wedge \neg q\} s \{q\} \rangle] \\
 & = \ [p \text{ unless } q \text{ in } F \Box G \wedge \langle \exists s: s \text{ in } F :: \{p \wedge \neg q\} s \{q\} \rangle] \vee \\
 & \quad [p \text{ unless } q \text{ in } F \Box G \wedge \langle \exists s: s \text{ in } G :: \{p \wedge \neg q\} s \{q\} \rangle] \\
 & = \ [p \text{ unless } q \text{ in } F \wedge p \text{ unless } q \text{ in } G \\
 & \quad \wedge \langle \exists s: s \text{ in } F :: \{p \wedge \neg q\} s \{q\} \rangle] \vee \\
 & \quad [p \text{ unless } q \text{ in } F \wedge p \text{ unless } q \text{ in } G \\
 & \quad \wedge \langle \exists s: s \text{ in } G :: \{p \wedge \neg q\} s \{q\} \rangle] \\
 & = \ [p \text{ ensures } q \text{ in } F \wedge p \text{ unless } q \text{ in } G] \vee \\
 & \quad [p \text{ ensures } q \text{ in } G \wedge p \text{ unless } q \text{ in } F]
 \end{aligned}$$



# Spojenie programov 4

Dôsledky:

1.  $p$  is stable in  $F \sqcup G = (p \text{ is stable in } F) \wedge (p \text{ is stable in } G)$
2.  $p$  unless  $q$  in  $F$ ,  $p$  is stable in  $G$  /  $p$  unless  $q$  in  $F \sqcup G$
3.  $p$  is invariant in  $F$ ,  $p$  is stable in  $G$  /  $p$  is invariant in  $F \sqcup G$
4.  $p$  ensures  $q$  in  $F$ ,  $p$  is stable in  $G$  /  $p$  ensures  $q$  in  $F \sqcup G$
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  $F \sqcup G$  ( $G$  je ľubovoľné):
  - $p$  unless  $q$ ,
  - $p$  is invariant
  - $p$  ensures  $q$

# Spojenie programov 5

Podmienené vlastnosti:

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,  $F \sqcup G$  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 if x ≤ 0 ∧ y > 0
```

```
□ 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  $F \sqcup G$
- Záver:  $y > 0 \rightarrow y < 0$  in  $F \sqcup G$

# 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 procesorov, ktoré z nej môžu čítať a množina procesorov, ktoré do nej môžu zapisovať
- Zobrazenie:
  - priradí každý príkaz nejakému procesoru
  - priradí (alokuje) každú premennú pamäti
  - špecifikuje "control flow" pre každý procesor (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

# Distribučovaná 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

## Zobrazenia na Synchrónne architektúry

- podobne ako ASM
- navyac procesory majú spoločné hodiny; v každom kroku každý procesor vykoná inštrukciu
  - viacero procesorov môže zapisovať do pamäte naraz, ak píšú to isté
  - viacerí môžu čítať
  - nemožno naraz čítať aj písať
- Príklad:  $z := x + 1 \parallel y := x + 2$ 
  - 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:  $z := x + 1 \parallel y := x + 2$  sa môže vykonať na 3 procesoroch takto:

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

# 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  $s \equiv \langle op\ j: 1 \leq j \leq N :: x[j] \rangle$
- $s$  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

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

Sekvenčná architektúra (SA):

- 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  $\langle \parallel j: 0 \leq j < N :: m := \max(m, A[j]) \rangle$

end {Maximum1}

## Výpočet maxima 2

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

1. možnosť:

initially  $j = 0$

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

2. možnosť:

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

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

# Výpočet maxima 3

## Paralelná architektúra (PA):

- 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

$\langle \mid j: 0 \leq j < N :: X[N+j] = A[j] \rangle$

$\square \langle \mid j: 1 \leq j < N :: X[j] = \max(X[2j], X[2j+1]) \rangle$

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 ⟨ ||j: 0 ≤ j < M:: A[j] = max(A[2j], A[2j+1])⟩  
end {Maximum3}
```

- $A[0] = \text{maximum}$ , 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  $A[0] = \langle \max j: 0 \leq j < N:: A^0[j] \rangle$

# DÚ

Program Maximum3

```
assign < ||j: 0 ≤ j < M:: A[j] = max(A[2j], A[2j+1])>  
end {Maximum3}
```

- Čo vieme povedať o prvkoch poľa (okrem A[0]) v
- čase  $O(\log N)$ ?
- Sformulujte nejakú safety podmienku a ukáže že platí
- Sformulujte nejakú progress podmienku a ukáže že platí
- Ako bude vyzeráť pole keď výpočet dosiahne FP

Program Maximum4

```
assign < ¶ j: 0 ≤ j < M:: A[j] = max(A[2j], A[2j+1])>  
End
```

Bude program fungovať? Prečo?

# Porovnanie dvoch neklesajúcich postupností 1

dané dve postupnosti  $f, g$

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

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

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

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

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

end{Compare}

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

# 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.  $u < N \wedge v < N$

$$\begin{aligned} \text{FP} \Rightarrow & (f[u] \neq f[u + 1]) \wedge \\ & \wedge (g[v] \neq g[v + 1]) \wedge \\ & \wedge (f[u + 1] \neq g[v + 1]) \end{aligned}$$

# Porovnanie dvoch neklesajúcich postupností 4

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

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

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

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

$$\{ f[i] \mid 0 \leq i \leq N \} \neq \{ g[i] \mid 0 \leq i \leq N \}$$

teda  $f$  a  $g$  nemajú rovnakú množinu prvkov

- $FP \wedge I \Rightarrow$

$$[u = N \equiv v = N] \wedge$$

$$[u = N \equiv \{ f[i] \mid 0 \leq i \leq N \} = \{ g[i] \mid 0 \leq i \leq N \}]$$

- dá sa ukázať, že FP sa dosiahne

# Porovnanie dvoch neklesajúcich postupností 5

Paralelná architektúra (PA):

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

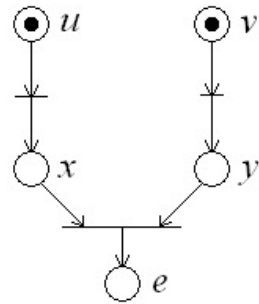
Úloha: vypočítať  $r$  tak aby:

- $r[v]$  je true, ak vrchol  $v$  je dosiahnuteľný:  
 $\langle \wedge v: v \in V :: r[v] \equiv \text{vrchol } v \text{ je dosiahnuteľný} \rangle$
- invariant  
 $\langle \wedge v: v \in V :: r[v] \Rightarrow v \text{ je dosiahnuteľný} \rangle \wedge r[\textit{init}]$

# Dosiahnuteľnosť v orientovanom grafe

- $FP \equiv \langle \wedge u, v: (u, v) \in E :: r[u] \Rightarrow r[v] \rangle$   
 $FP \equiv \langle \wedge u, v: (u, v) \in E :: r[v] = (r[u] \vee r[v]) \rangle$   
kedže  $r[v] = (r[u] \vee r[v]) \equiv (r[u] \Rightarrow r[v])$
- platnosť invariantu zaručíme, ak na začiatku bude  $r[v] = \text{false}$  pre  $v \neq \text{init}$  a  $r[\text{init}] = \text{true}$
- Program Reach  
declare  $r$ : array[ $V$ ] of boolean  
initially  $\langle \forall v: v \in V :: r[v] = (v = \text{init}) \rangle$   
assign  $\langle \forall u, v: (u, v) \in E :: r[v] := r[u] \vee r[v] \rangle$   
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

$u, x := u - 1, x + 1$  if  $u > 0$

$v, y := v - 1, y + 1$  if  $v > 0$

$x, y, e := x - 1, y - 1, e + 1$  if  $x > 0 \wedge y > 0$

end{PetriNet}

# Najkratšia cesta 1

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

$$W[i, j] =$$

- váňa hrany  $(i, j)$ , ak  $(i, j) \in E$
- 0, ak  $i = j$
- $\infty$ , ak  $i \neq j \wedge (i, j) \notin E$
- prázdna cesta: z vrcholu  $i$  do  $j$  vždy existuje; jej dĺžka:

$\infty$

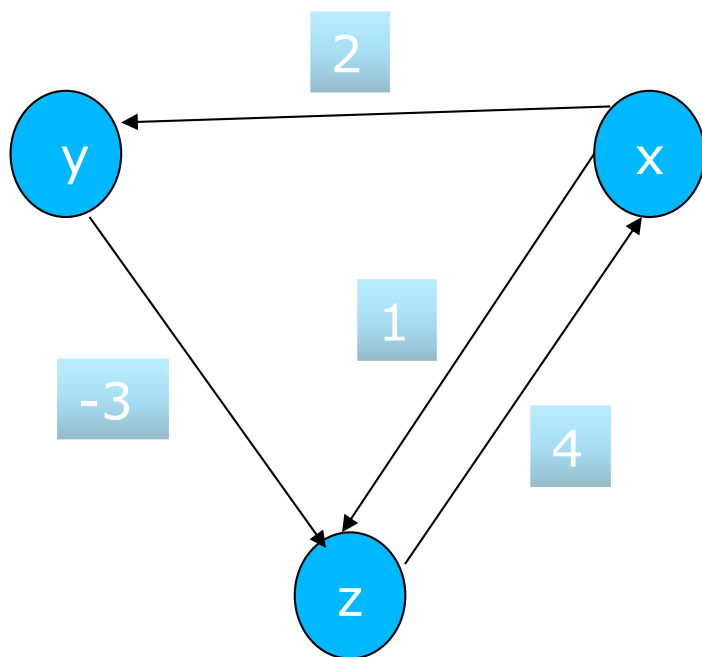


# Najkratšia cesta 2

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  $D[i, j] = \infty$

# Najkratšia cesta - Príklad



# Najkratšia cesta 3

- Špecifikácia: program má počítať maticu  $d$  tak, že invariant FP  $\Rightarrow (d = D)$   
true  $\rightarrow$  FP
- Stratégia neformálne:
  - 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)

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  $\infty$ ,

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

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

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

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

$$\neg FP \wedge (num, sum) = (m, n) \rightarrow (num, sum) < (m, n) \quad (3)$$

# 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  $\langle \forall i, j :: d[i, j] = W[i, j] \rangle$

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

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 sekvenčializáciou:

PASCAL-like program PP (sekvenčializácia P1)

```
Program 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  $\langle \parallel i, j :: d[i, j] = W[i, j] \rangle$

assign

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

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

assign

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

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

end{P4}

program P4: potrebuje  $O(N)$  krokov

# Najkratšia cesta - paralelná synchronná architektúra

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

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

Program P5

```
initially  $\langle \forall i, j :: d[i, j] = W[i, j] \rangle$ 
```

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

```
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 ASM 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:

$$d[i, j] = H[i, j, k]$$

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

$\wedge k \leq N$



# Najkratšia cesta – asynchrónna architektúra

v P4 je synchronnosť 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

$$d[i, j] \leq H[i, j, k[i, j]]$$

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

$$\wedge k[i, j] \leq N$$

# 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

$d[r, j] \leq H[r, j, r]$

# 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 zväčši s každou zmenou stavu a je ohraničená zhora.

Program P6

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

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

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

$$\langle \forall i, j :: d[i, j], r := \min(d[i, j], d[i, r] + d[r, j]), r + 1 \\ \text{if } r < N \wedge k[i, r] \geq r \wedge k[r, j] \geq r \rangle$$

end{P6}

# Readers–Writers Problem 1

## Zadanie:

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

$$\text{invariant } 0 \leq nr \leq N \wedge 0 \leq nw \leq N \quad (1)$$

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

Úloha: modifikovať priradenia programu user tak, aby platilo:

$$\text{invariant } nw \leq 1 \wedge (nr = 0 \vee nw = 0) \quad (2)$$

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

## Readers–Writers Problem 3

Riešenie:

ak  $0 \leq nr \leq N \wedge 0 \leq nw \leq N$ , tak

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

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

máme nový invariant  $t \geq 0$  (3)

modifikovaný program user:

declare  $t$ : int

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

assign

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

$$nr = k \wedge k > 0 \rightarrow nr \neq k \quad (4)$$

## 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 navyše  
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  $b = \text{false}$

assign

$nr := nr + 1$ if $t \geq 1 \wedge \neg b$	{started read}
▯ $nr := nr - 1$	{end read}
▯ $nw, nq, b := nw + 1, nq - 1, \text{false}$ 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. ( $nq > 0 \rightarrow nw = 1$ ) 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  $nq > 0 \rightarrow nw = 1$ :

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

- aplikujeme indukciu

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

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

## Večerajúci filozofi



# Večerajúci filozofi 1

- 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:

$$u.t \equiv (u.dine = t) \quad \textit{thinking}$$

$$u.h \equiv (u.dine = h) \quad \textit{hungry}$$

$$u.e \equiv (u.dine = e) \quad \textit{eating}$$

## Večerajúci filozofi 2

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

$$t \rightarrow h \rightarrow e \rightarrow t \rightarrow \dots$$

realizovanie zmien stavu premennej *u.dine*:

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

program os:  $h \rightarrow e$

Úloha: navrhnuť program os tak, aby os  $\parallel$  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

### 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)
- podmienená vlastnosť pre user: (udn4)  
 $\langle \forall (u, v) :: \neg(u.e \wedge v.e) \rangle /$   
 $\langle \forall u :: u.e \rightarrow \neg u.e \rangle$

### Špecifikácia pre useros:

- invariant  $\neg(u.e \wedge v.e \wedge E[u, v])$  in useros (dn1)
- $u.h \rightarrow u.e$  in useros (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 $\parallel$ os:

- $u.t$  unless  $u.h$  in user $\parallel$ os (dn3)
- $u.h$  unless  $u.e$  in user $\parallel$ os (dn4)
- $u.e$  unless  $u.t$  in user $\parallel$ 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

Spec1:

(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 spíňa (dn1), (dn2)
- os spíňa 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  $user \parallel os$  z predchádzajúceho a z  $(odn3)$  v  $os$
- $\langle \forall u :: u.e \rightarrow \neg u.e \rangle$  platí v  $user \parallel os$  lebo toto je dôsledok  $(udn4)$  a hypotézy  $(udn4) - (dn1)$  platí v  $user \parallel os$
- $\langle \forall u :: u.h \rightarrow u.e \rangle$  platí v  $user \parallel os$ , lebo je to dôsledok  $(odn4)$  a predpoklad  $(odn4)$  platí
- $(dn1)$  platí v  $user \parallel os$  z predchádzajúceho

## Večerajúci filozofi: Spec2

- Problém s implementovaním Spec1:

- hladný je (eats), ak nejedia susedia – spĺňa to (odn3)
- problém je zabezpečiť (odn4), pretože jeden proces môže stále opakovať  $t \rightarrow h \rightarrow e \rightarrow t \rightarrow h \rightarrow e \rightarrow \dots$

Riešenie:

zavedieme asymetriu medzi procesmi – usporiadanie

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

čiasť 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.  $u \rightarrow v$ )

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

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

# Večerajúci filozofi: Spec2

## Formálny popis: Spec2

(odn1)–(odn3), (odn5)–(odn8), kde

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

$(\text{prior}[u, v] = v)$  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)

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

čo znamená:

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

# Večerajúci filozofi: Spec3

## Problém s implementovaním Spec2:

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

Ďalšie zjemnenie - Spec3:

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  $u.e \wedge E[u, v] \Rightarrow \text{fork}[u, v] = u$  (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

Ďalšie zlepšenie:

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

Distribúovaná implementácia priorít: 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 nepoše preč – vtedy sa aj vyčistí (odn12, zodpovedá odn6)
- *clean forks* držia len *hungry* (odn13)

## Večerajúci filozofi: Spec4

nech  $clean[u, v]$ : boolean

$clean[u, v] = true \equiv$  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: navrhne mechanizmus, ako  $v$  informuje suseda  $u$ , že je *hungry*

- zavedieme *request-token* pre každú 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 vidličky (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  $u$  posielala *request-token* procesu  $v$ , je označená  $sendreq[u, v]$
- podmienka, za ktorej  $u$  posielala vidličku procesu  $v$ , je označená  $sendfork[u, v]$

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

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

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

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

$sendfork[u, v] \equiv$

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

# 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. očísľujeme 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

- $u.g = \text{true}$  ak  $u$  čaká
- $x.co = \text{true}$  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
- $x.g = \text{true}$  ak všetci členovia  $x$  čakajú, t.j  
 $x.g \equiv \langle \forall u: u \in x.mem :: u.g \rangle$
- $x^*.co \equiv x.co \vee \langle \exists y: E[x, y] :: y.co \rangle$



## Koordinácia schôdzí 5

Daný je program prof

Ideme navrhnúť program coord(**inator**) tak aby program

Sync(**hronizator**) = prof ◻ coord

mal nasledovné vlastnosti:

Špecifikácia sync

$\top$  x.co unless x.g                    sy1

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

x.g  $\rightarrow$  x\*.co                        sy3

## Koordinácia schôdzí 5

Špecifikácia prof

u.g unless  $\langle \exists x: u \in x.\text{mem} :: x.\text{co} \rangle$

pr1

$\neg$ x.co je stable

pr2

(t.j. schôdzu nemôže začať prof)

## Koordinácia schôdzí 6

$\langle \forall x: x.co \rightarrow \top x.co \rangle$

pr3

Odvođená 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 constant	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

$\langle \Box x :: x.g = \langle \forall u: u \in x.mem :: u.g \rangle \rangle$

$\Box x^*.co \equiv x.co \vee \langle \exists y: E[x,y] :: y.co \rangle \rangle$

initially

$\langle \Box x :: x.co = false \rangle$

assign

$\langle \Box x :: x.co = true \text{ if } x.g \wedge \neg x^*.co \rangle$

## Koordinácia schôdzí 10

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

sy1 (⊢ x.co unless x.g)

⊢ x.co unless x.g v coord1

⊢ 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 –  $x.co \Rightarrow x.e$

# Koordinácia schôdzí 13

-----**sync**-----

-----**user**-----

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

-----**coord**-----

sync = prof  $\parallel$  middle  $\parallel$  os

user a os sú z večerajúcich filozofov

usr = prof  $\parallel$  middle

coord = middle  $\parallel$  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)  $t \rightarrow h$
- 2)  $e \rightarrow t$

## Koordinácia schôdzí 15

- ak je komisia v stave  $t$  tak sa stane  $h$  iba ak všetci jej členovia čakajú
- ak prof  $\square$  middle bude spíňať 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

$\langle \Box x :: x.g = \langle \forall u: u \in x.mem :: u.g \rangle \rangle$

initially

$\langle \Box x :: x.co = false \rangle$

assign

$\langle \Box x ::$

$x.dine := h \text{ if } x.t \wedge x.g$

$\Box$

$x.dine := t \text{ if } x.e \wedge \neg x.co \rangle$

## 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:

$middle \sqsubseteq os1$

splňa cd1-cd3

$prof \sqsubseteq middle$

splňa požiadavky na user

$sync1 = prof \sqsubseteq middle \sqsubseteq os1$  splňa „sync“

## Koordinácia schôdzí 18

Lema 1.  $x.co \Rightarrow x.e$  v sync1

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

Dôkaz.

Na začiatku  $x.co = \text{false}$  a teda  $x.co \Rightarrow x.g$  platí. Keďže

$x.co \Rightarrow x.e \equiv \neg x.co \vee x.e$ , stačí ukázať, že

$\neg x.co \vee x.e$  je stabilné v prof, middle a os1.

Jediný zaujímavý prípad je os1.  $\neg x.co$  unless  $x.e$  ale  $x.e$  je stabilné v os1 a z toho  $\neg x.co \vee x.e$  je stabilné v os1

## Koordinácia schôdzí 19

middle  $\sqcap$  os1

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

z prechádzajúcej lemy vieme, že  $\lceil x.e \Rightarrow \lceil x.co$   
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  $\sqcap$  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  $\sqcap$  middle  $\sqcap$  os1 (= sync)

z podmienok pre filozofov vieme:

$\langle \forall x, y: E[x, y] :: \neg(x.e \wedge y.e) \rangle$  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: klúdná, smädná, pijúca.  
stavy mení v poradí:

klúdná  $\rightarrow$  smädná  $\rightarrow$  pijúca  $\rightarrow$  klúdná

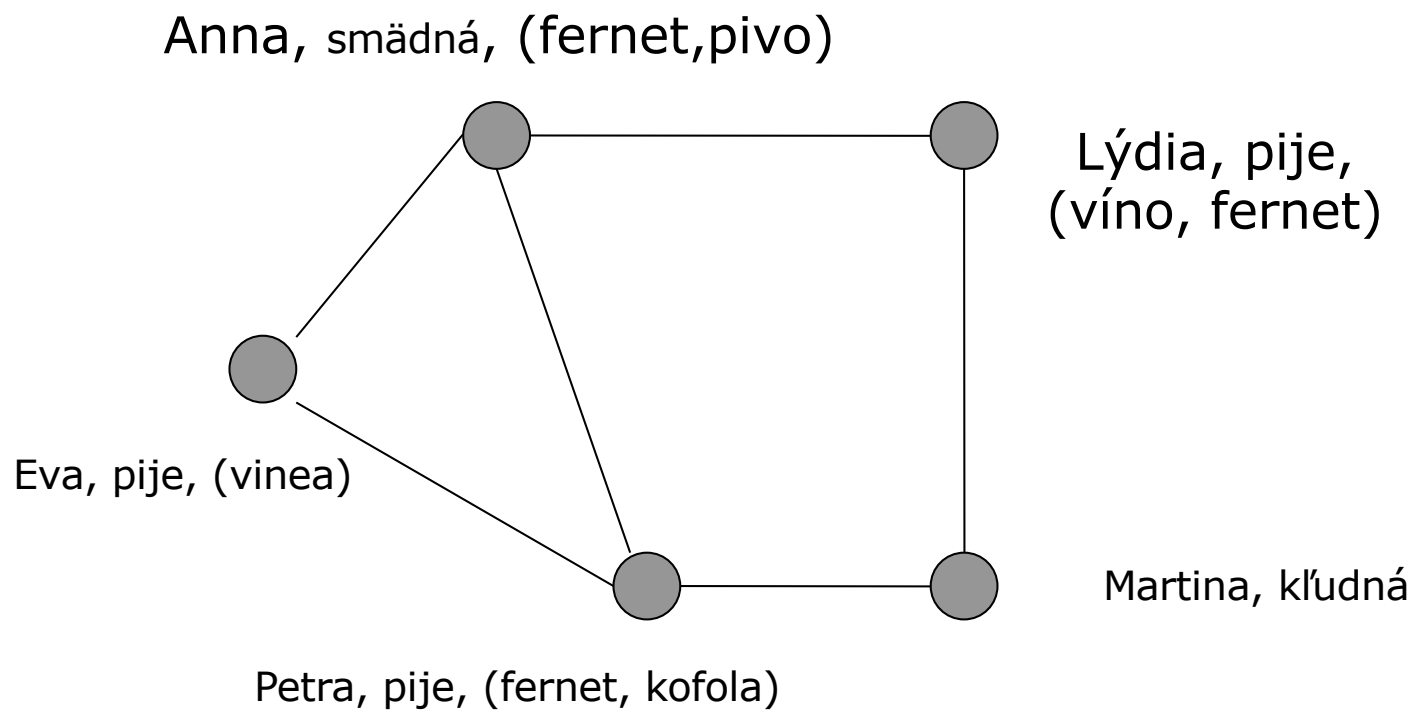
Každej smädnej alebo pijúcej (=neklúdnej)  
filozofke je priradená nejaká množina nápojov (v okamihu  
prechodu z klúdná do smädná), ktorá sa pokiaľ je  
filozofka neklúdná nemení.

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:

klúdna → smädna → pijúca → klúdna  
thinking → hungry → eating → thinking

stav bude dvojica (stav\_večerajúci, stav\_pijúca)

## Pijúce filozofky 4

Nemôžeme ovplyvniť prechody:

$(x, \text{klúdná}) \rightarrow (x, \text{smädná})$

$(x, \text{pijúca}) \rightarrow (x, \text{klúdná})$  pre ľubovoľný „večerajúci stav“  $x$

Predpokladáme, že:

1.  $(\text{thinking}, \text{smädná})$  sa stane hungry alebo pijúca
2. hladná bude raz jesť (ak nikto neje večne)
3. konzumujúca a smädná sa stane medituujúcou a pijúcou, konzumujúca a nesmädná sa stane medituujúcou alebo smädnou

## Pijúce filozofky 5

ideme navrhnuť 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  $u.nap = \emptyset \equiv u.k$  udr5
- $\forall u : u.p \rightarrow \neg u.p$  udr6

udr4 - ak je  $u$  neklúdná, množina nápojov sa nemení a tá je prázdna ak je klúdná udr5

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



# Pijúce filozofky 7

Špecifikácia bar

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

dr1

dr2

Obmedzenia pre osdrink

$u.k, u.nap = N$  sú constant  
 $u.p$  je stable

odr1

odr2

Odvođené vlastnosti pre osdrink

$\neg u.s$  je stable

odr3

$u.s$  unless  $u.p$

odr4

## Pijúce filozofky 8

### Odvozené 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 $\rightarrow \neg$ 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 constatatnt	mdr1
u.p je stable	mdr2
u.t unless u.h	mdr3
u.h je stable	mdr4
u.e unless u.t	mdr5
invariant $\lceil(u.p \wedge v.p \wedge E[u,v] \wedge (u.nap \cap v.nap \neq \emptyset))$	mdr6
$(u.t \wedge u.s)$ ensures $\lceil(u.t \wedge u.s)$	mdr7
$(u.e \wedge \lceil u.s)$ ensures $\lceil(u.e \wedge \lceil u.s)$	mdr8
 dr3-7, dr8, $\langle \forall u : u.p \rightarrow \lceil u.p \rangle$	 mdr9
-----	
$\langle \forall u : u.e \wedge u.s \rightarrow u.t \wedge u.p \rangle$	

# 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

$u.\text{maydrink} \equiv$

$\langle \forall n, v: n \in u.\text{nap} \wedge E[u, v] :: \text{bot}[u, v, n] = u \wedge$   
 $(\text{rb}[u, v, n] = v \vee \text{fork}[u, v] = u) \rangle$

$\text{sendbot}[u, v, n] \equiv$

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

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

# Pijúce filozofky 18

## špecifikácia middle

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

# Pijúce filozofky 19

Program middle

initially

$\langle \forall u :: u.dine, u.drink = t, k \rangle$

assign

$\langle \forall u :: u.dine := h \text{ if } u.t \wedge u.s$   
 $\quad \quad \quad \sim t \text{ if } u.e \wedge \neg u.s \rangle$

$\square \langle \forall u: \quad u.drink := p \text{ if } u.s \wedge u.maydrink$   
 $\quad \quad \quad || u.dine := t \text{ if } u.s \wedge u.maydrink \wedge u.e \rangle$

$\square \langle \forall u, v, n: E[u,v] ::$   
 $\quad bot[u,v,n] := v \text{ if } sendbot[u, v, n]$   
 $\square rb[u, v, n] := v \text{ if } sendreq[u,v,n] \rangle$

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:

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

-----

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

# Mutual Exclusion

Úlohou je transformovať program user  
na program

$$\text{mutex} = \text{user}' \parallel G$$

kde  $\text{user}'$  získame z  $\text{user}$  len pridaním príkazov,  
ktoré sa vykonajú súčasne s príkazom z  $\text{user}$  a  $G$  sú  
nové príkazy.

V  $G$  musí byť  $u.e$  stabilné a  $u.t$  konštantné. Jediné  
premenená, ktoré sa využívajú z  $\text{user}$  je  $u.dine$



# Mutual Exclusion

Špecifikácia pre mutex:

*Invariant*

$\neg( u.e \wedge v.e \wedge u \neq v )$

MX1

$u.h \rightarrow u.e$

MX2

# Mutual Exclusion

## Odvozené 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 len 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

$\langle \forall u :: u.dine, u.b = t, false \rangle$

transform - ak u.dine sa stane t tak sa vykoná

$u.b := false$

add

$\langle \forall u ::$

$u.b, turn := true, u$       if  $\neg u.b \wedge u.h$

$\exists u.dine := e$       if  $u.b \wedge \neg(u'.b \wedge turn = u)$

$\rangle$

end

# Two Process Mutual Exclusion

Dôkaz správnosti

Invariant

$u.e \Rightarrow [u.b \wedge \neg(u'.b \wedge \text{turn} = u)]$  in mutex2

Z tohoto plynie MX1

Progres podmienka MX2

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

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

Inak (ak platí  $(u'.b \wedge \text{turn} = u)$ ) 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

$\langle \text{u} :: \text{u.dine}, \text{u.b}, \text{u.r} = \text{t}, \text{false}, \text{false} \rangle$

transform - ak u.dine sa stane t tak sa vykoná

$\text{u.b} := \text{false}$

add

$\langle \text{u} ::$

$\text{u.b}, \text{u.r} := \text{true}, \text{true}$       if  $\neg \text{u.b} \wedge \text{u.h}$

$\text{u.r}, \text{turn} := \text{false}, \text{u}$       if  $\text{u.r}$

$\text{u.dine} := \text{e}$       if  $\text{u.b} \wedge \neg \text{u.r} \wedge \neg(\text{u'.b} \wedge \neg \text{u'.r} \wedge \text{turn} = \text{u})$

$\rangle$

end

# N- Processes Mutual Exclusion

Rozdelíme procesy do dvoch neprázdnych 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  $\oplus$  1

add

<⊢u ::

u.num, issue := issue, issue  $\oplus$  1                      if u.num = N  $\wedge$  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  $u.num \neq v.num$  (aj ranky sú teda rôzne).

Ak je proces  $u$  hungry a  $u.num \neq 0$  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.

$u \leq v$  ak  $(u.num, u)$  je lexikograficky menej alebo rovné ako  $(v.num, v)$

# Mutual Exclusion – Bakery Program

Program Bakery 2

always

$\langle \square u :: u.\text{rank} = \langle + v: (v.\text{num} = 0) \vee (u \leq v) :: 1 \rangle$

initially

$\langle \square u :: u.\text{dine}, u.\text{num} = t, 0 \rangle$

transform - ak u.dine sa stane t tak sa vykoná

u.num:=0

add

$\langle \square u ::$

    u.num:=  $\langle \max v :: v.\text{num} \rangle + 1$       if u.num = 0  $\wedge$  u.h

$\square$  u.dine := e                              if u.num  $\neq$  0  $\wedge$  u.h  $\wedge$  u.rank=N

$\rangle$

end

# Triedenie 1

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

Úloha: nájsť pole  $y[1..N]$  také, že

$y$  je permutáciou  $x \wedge \langle \forall i: 1 \leq i < N :: y[i] < y[i + 1] \rangle$  (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.)

- zavedme metriku

$$M = \langle +i, j: 0 < i < j \leq N: y[i] > y[j]:: 1 \rangle$$

- stratégia je formálne definovaná nasledujúcim invariantom, FP a progress podmienkou:

$$\text{invariant } y \text{ je permutáciou } x \quad (2)$$

$$\text{FP} \equiv (M = 0) \quad (3)$$

$$\langle \forall k: k > 0:: M = k \rightarrow M < k \rangle \quad (4)$$

- Dôkaz správnosti: stačí ukázať, že platí

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

a že (1) platí v každom FP.



# Triedenie 3

## 2. Utriedenie časti poľa

budeme používať premennú  $m$ ,  $1 \leq m \leq N$  a v každom bode výpočtu bude platiť

(a)  $y[m + 1 .. N]$  je utriedené

t.j.  $\langle \wedge i, j: m < i < j \leq N :: y[i] < y[j] \rangle$

(b) všetky prvky  $y[1 .. m]$  sú menšie než prvky  $y[m + 1 .. N]$

t.j.  $\langle \wedge i, j: 1 \leq i \leq m < j \leq N :: y[i] < y[j] \rangle$

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

$\langle \wedge i, j: 1 \leq i < j \leq N \wedge m < j :: y[i] < y[j] \rangle$

Na začiatku  $m = N$  to zaručuje a ak chceme znížiť  $m$ , tak musíme permutovať  $y[1 .. m]$  tak, aby pre nejaké  $k$ ,  $1 < k \leq m$ , 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  $x$

$$\wedge 1 \leq m \leq N$$

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

$$FP \equiv (m \leq 1) \quad (6)$$

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

# Triedenie – Jednoduché riešenia 1

budeme uvádzať len assign sekciu a predpokladať, že  
initially  $\langle \parallel i: 1 \leq i \leq N :: y[i] = x[i] \rangle$

## 1. Reduckia počtu zle usporiadaných dvojíc

Program P1

assign

$\langle \parallel i: 1 \leq i < N ::$

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

$\parallel y[i + 1] := \max(y[i], y[i + 1]) \rangle$

end{P1}

## Triedenie – Jednoduché riešenia 2

Program P1'

assign

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

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.  $\text{min}(7, 5, 1976) = 5$ ,  $\text{max}(7, 5, 1976) = 1976$ ,  $\text{sort2}(7, 5) = (5, 7)$

# 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  $O(N^2)$  krokov

P2: podobne

V nasledujúcom ukážeme lepšiu realizáciu P1 a P2 na SA.

Z P2 odvodíme heapsort, zložitosť  $O(N \cdot \log N)$  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

strom:  $y[i]$  má synov  $y[2i]$  a  $y[2i + 1]$  (a opačne *byť\_synom*)

potomok: nereflexívny tranzitívny uzáver relácie *byť\_synom*

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

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

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

formálne:

invariant

$\langle \forall i, j: 1 \leq i \leq m \wedge j \leq m \wedge j \text{ je potomok } i \wedge top[i] :: y[i] > y[j] \rangle$  (8)

teda

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



## 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  $y[i] := \max(y[2i], y[2i + 1])$ .

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

# Triedenie – Sekvenčné architektúry 5

Program P3 {nedeterministický heapsort}

declare

l: array [1 .. N] of integer

top: array [1 .. N] of boolean

always

$\langle \forall i: 1 \leq i \leq N ::$

$l[i] = \begin{array}{ll} 2i + 1 & \text{if } y[2i + 1] > y[2i] \wedge (2i + 1 \leq m) \sim \\ 2i & \text{if } \neg(y[2i + 1] > y[2i] \wedge (2i + 1 \leq m)) \end{array} \rangle$

initially

$\langle \forall i: 1 \leq i \leq N :: top[i] = (2i > N) \parallel y[i] = x[i] \rangle \parallel m = N$

assign

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

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

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

$\langle \forall i: 1 \leq i \leq N/2 ::$

$y[l[i]], y[i], top[i], top[l[i]] := sort2(y[l[i]], y[i]), true, (2.l[i] > m)$   
 $\text{if } (2i \leq m) \wedge \neg top[i] \wedge top[2i] \wedge top[2i + 1] \rangle$

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):  $\langle \forall k: k > 1 :: m = k \rightarrow m < k \rangle$

Ukážeme, že platí

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

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

$$m = k \wedge k > 1 \wedge \text{top}[1] \rightarrow m < k \quad (12)$$

Potom  $m = k \rightarrow (m = k \wedge \text{top}[1]) \vee m < k$  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):  $\text{true} \rightarrow \text{top}[1]$

Zavedieme metriku

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

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

Teda máme

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

avšak hodnota  $d$  je ohraničená, teda

$\text{true} \rightarrow \text{top}[1]$

Dôkaz (6):  $\text{FP} \equiv (m \leq 1)$

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

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

# Triedenie – Paralelné architektúry (synchronne) 1

zistíme, čo sa dá vykonať paralelne

Program P4

assign

```
    < || i: 1 ≤ i < N ∧ even(i):: y[i], y[i + 1] := sort2(y[i], y[i + 1]) >  
▯ < || i: 1 ≤ i < N ∧ odd(i):: y[i], y[i + 1] := sort2(y[i], y[i + 1]) >  
end{P4}
```

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

# Triedenie – Paralelné architektúry (synchronne) 2

Program P5

declare  $m$ : integer

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

assign

$\langle \parallel i: 1 \leq i \leq N :: y[i] := \min(y[i], y[i + 1])$  if  $(i = k \bmod 2)$   
 $\sim \max(y[i], y[i - 1])$  if  $(i \neq k \bmod 2) \rangle$

$\parallel k := k + 1$  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
```

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

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

```
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  $mr \subseteq ms$

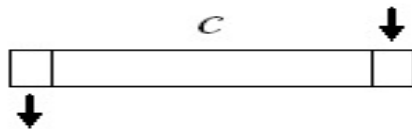
$|mr| = n \rightarrow |mr| = n + 1$

## Faulty channels 2

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

$c, ms := c; \text{head}(ms), \text{tail}(ms)$  {sender}  
▮  $c, mr := \text{tail}(c), mr; \text{head}(c)$  if  $c \neq \text{null}$  {receiver}

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



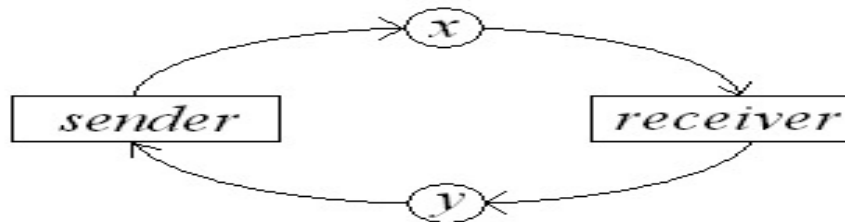
## Faulty channels 3

Budeme riešiť problém, ak  $c$  je chybné – správa môže byť stratená alebo duplikovaná, ale len *konečne veľa* rás

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 stratená, ak *sender* zapíše do  $x$  prv, než to *receiver* prečítal
- správa je zduplikovaná, 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 očísľujeme a namiesto správ posielame len ich čísla)
- nech:
  - $ks$  – posledné číslo poslané *senderom*
  - $kr$  – posledné číslo prijaté *receiverom*
- 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 navyše 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 priradovať len *sender*
- do *y* vie priradovať len *receiver*

## Faulty channels 6

zjemníme progress podmienku:

$$kr = n \rightarrow ks = n + 1 \rightarrow kr = n + 1$$

to sa zabezpečí príkazom

$$ks := kr + 1 \parallel kr := ks$$

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}  
   $\parallel$   $ks := y + 1$             {sender}  
   $\parallel$   $x := ks$                  {sender}  
   $\parallel$   $kr := x$                 {receiver}  
end

## Faulty channels 7

$x$  je chybový kanál, cez ktorý sa posiela  $ks$

$y$  je chybový kanál, cez ktorý sa posiela potvrdenie

$$\text{invariant } y \leq kr \leq x \leq ks \leq y + 1 \quad (\text{I1})$$

*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:
- $y = n \wedge ks \neq n + 1 \Rightarrow y = kr = x = ks \quad \{z (I1)\}$
- $\{y = n\} ks := y + 1 \{ks = n + 1\}$
- $\{y = n \wedge ks \neq n + 1\} s \{y = n \vee ks = n + 1\}$



# Faulty channels 9

## Prenos ľubovoľnej postupnosti dát

$ms$  – nekonečná postupnosť (lokálna *senderu*)

$mr$  – nekonečná postupnosť (lokálna *receiveru*)

$ms[j]$ ,  $j > 0$  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}

$kr, mr := x.dex, mr$ ;  $x.val$  if  $kr \neq x.dex$  {receiver}

end{P2}

# Faulty channels 10

Správnosť P2:

invariant  $mr \subseteq ms$

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

Pozorovanie: (I1) platí aj pre P2, kde miesto  $x$  je  $x.dex$

označenie  $x.val \in ms$  ak  $\exists j : x.val = ms[j]$

platí nasledovný

invariant  $x.val \in ms \wedge mr \subseteq ms \wedge |mr| = kr$  (I2)

(Hint: z (I1)  $kr \neq x.dex \Rightarrow kr + 1 = x.dex$ )

Progress podmienka má podobný dôkaz ako v P1

Poznámka. Z (I1) a hintu  $kr := x.dex$  if  $kr \neq x.dex$  možno v P2 nahradiť  $kr := kr + 1$  if  $kr \neq x.dex$  a zároveň možno  $kr := y + 1$  nahradiť  $ks := ks + 1$  if  $ks = y$

# Faulty channels 11

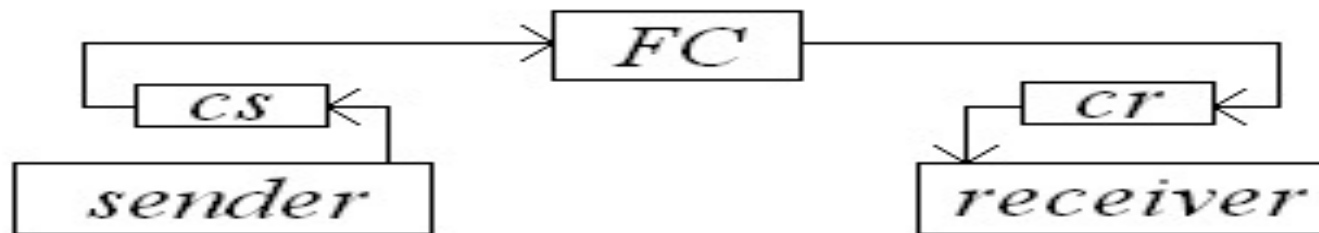
V predchádzajúcich programoch duplikovanie a strata správy boli „pod kontrolou“ *sendera a receivera*

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  $cr$

- $cs$ : celá „história“  $cs$
- $cs$  –  $cs$ : prefix  $cs$  bez  $cs$

$u$  loss  $v$ :  $v$  je možný výstup z chybného kanála, za predpokladu, že vstup je  $u$

*(null 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 ( $cs$  -  $cs$ ) loss  $cr$

# 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  $cs$ )

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:

$$p.m \wedge p.n \Rightarrow m = n$$

$$p.m \wedge cs = null \text{ unless } \neg p.m \vee cs = \langle\langle m \rangle\rangle$$

$$p.m \wedge (cs = cs^0) \wedge (cs \neq null) \text{ unless } \neg p.m \vee cs = \text{tail}(cs^0) \vee cs = cs^0; m$$

$$p.m \wedge |cs| = k \rightarrow \neg p.m \vee |cs| > k$$

Záver:  $p.m \rightarrow \neg p.m \vee (m \in cr)$

# Faulty channels 15

Program simulujúci FC

strata správy

$cs := \text{tail}(cs)$  if  $cs \neq \text{null}$

duplikácia správy

$cr := cr; \text{head}(cs)$  if  $cs \neq \text{null}$

správny transfér

$cs, cr := \text{tail}(cs), cr; \text{head}(cs)$  if  $cs \neq \text{null}$

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  $b = \text{false}$

assign

$cs := \text{tail}(cs)$  if  $cs \neq \text{null} \wedge \neg b$

$cr := cr; \text{head}(cs)$  if  $cs \neq \text{null} \wedge \neg b$

$b, cs, cr := \text{false}, \text{tail}(cs), cr; \text{head}(cs)$  if  $cs \neq \text{null}$

$b := \text{true}$

end



## Faulty channels 17

Protokol FC pre komunikáciu s dvoma chybnými kanálmi (pre  $cr$ ,  $cs$  a pre  $ackr$ ,  $acks$ ) má spĺňať 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

$ks, kr = 1, 0$

▯  $cs, cr, acks, ackr = null, null, null, null$

▯  $mr = null$

assign

$ackr := ackr; kr$

▯  $ks := ks + 1$  if  $acks \neq null \wedge ks = \text{head}(acks)$

▯  $acks := \text{tail}(acks)$  if  $acks \neq null$

▯  $cs := cs; (ks, ms[ks])$

▯  $kr, mr := kr + 1, mr; \text{head}(cr).val$  if  $cr \neq null \wedge kr \neq \text{head}(cr).dex$

▯  $cr := \text{tail}(cr)$

end

## Faulty channels 19

Pri dokazovaní správnosti protokolu použijeme upravené invarianty (I1), (I2)

invariant  $y \leq kr \leq x.\text{dex} \leq ks \leq y + 1$  (I1)

invariant  $x.\text{val} \in ms \wedge mr \subseteq ms \wedge |mr| = kr$  (I2)

(I1) sa dá prepísať ( $x, y$  môžu byť nedefinované, ak zodpovedajúce kanály sú *null*) nasledovným spôsobom:

$\langle \forall y: y \in acks \vee y \in ackr :: y \leq kr \wedge ks \leq y + 1 \rangle \wedge$

$\langle \forall x: x \in cs \vee x \in cr :: kr \leq x.\text{dex} \leq ks \rangle \wedge$

$kr \leq ks \leq kr + 1$

podobne sa dá prepísať aj (I2)

potrebujeme ešte jeden invariant:

invariant  $cs.\text{dex}, cr.\text{dex}, acks, ackr$  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  $kr \leq head(cr).dex \leq kr + 1$

teda

$kr \neq head(cr).dex \equiv (kr \bmod 2 \neq [head(cr).dex] \bmod 2)$

$ks = head(acks) \equiv (ks \bmod 2 = head(acks) \bmod 2)$

P3 môže byť zjednodušené tak, že nahradíme  $kr$ ,  $ks$   
nasledujúcimi:  $kr \bmod 2$ ,  $ks \bmod 2$

dostávame Alternating Bit Protocol

# Global snapshots 1

Úloha: modifikovať daný program tak, aby sme mohli zaznamenať stav jeho výpočtu

Toto je problémom pri asynchrónnych a distribuovaných systémoch

Príklad:

Program P

```
declare x, y, z: integer
```

```
initially x, y, z = 0, 0, 0
```

```
assign x := z + 1 || y := x || z := y
```

```
end
```

## Global snapshots 2

stavom P je trojica hodnôt  $x, y, z$

ľahko vidno, že jediná možná postupnosť stavov P je  
 $(0, 0, 0) \rightarrow (1, 0, 0) \rightarrow (1, 1, 0) \rightarrow (1, 1, 1) \rightarrow (2, 1, 1) \rightarrow$   
 $(2, 2, 1) \rightarrow \dots$

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

- nezaobráame 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* – okamžitý 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  $\langle \forall x :: x.rec \rangle$
- stav *cur* je daný momentálnymi hodnotami *x*

# Global snapshots 6

invariant a progress podmienka:

invariant  $\langle \bigwedge x :: x.done \rangle \Rightarrow \langle \exists u, v: [init] u [rec] \wedge [rec] v [cur] \rangle$   
 $begun \rightarrow \langle \bigwedge x :: x.done \rangle$

neformálne:

- zaznamenávanie sa skončí v konečnom čase
- $[rec]$  je na nejakej „ceste“ medzi  $[init]$  a  $[cur]$

Program P1

initially  $\langle \bigvee x :: x.done = false \rangle$

add  $\langle \bigvee x :: x.rec, x.done = x, \text{true if } begun \wedge \neg x.done \rangle$

end{P1}

P1 je nevhodný pre asynchrónne a distribuované počítače

# Global snapshots 7

Zjemnenie pôvodnej špecifikácie:

pre každý stav výpočtu (pôvodného programu)  
definujeme

$$\begin{aligned} x.\text{partial} &= & x.\text{rec} & \text{ if } x.\text{done} \\ &\sim & x & \quad \text{if } \neg x.\text{done} \end{aligned}$$

invariant a progress podmienka:

invariant  $begun \Rightarrow \langle \exists u, v: v \text{ obsahuje len zaznamenané premenné} :: [init] u [partial] \wedge [partial] v [cur] \rangle$

$begun \rightarrow \langle \wedge x :: x.\text{done} \rangle$

# 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

## Pravidlo $R$ :

- 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  $u$  vezmeme  $u;t$
  - za  $v$  vezmeme  $v$
- nech všetky premenné v  $t$  sú zaznamenané; potom *partial* sa nemení;
  - za  $u$  vezmeme  $u$
  - za  $v$  vezmeme  $v;t$

# 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 príkaz *t*. Nech *cur'* je stav pred vykonaním *t* a *cur* je stav po vykonaní *t*.

Podobne pre *partial*.

Z invariantu máme:

$$[init] u [partial'] \wedge [partial'] v [cur']$$

dálej

$$[cur'] t [cur]$$

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

*Prípád 1.*

$t$  obsahuje nezaznamenané premenné.

Potom ľahko vidno

$[partial'] t [partial] \quad (**)$

a

$[init] u; t [partial]$   
 $[partial'] v; t [cur] \quad (*)$

$t$  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

*Prípád 2.*

$t$  obsahuje len zaznamenané premenné.

Z definícií

$x.\text{partial} = x.\text{rec}$  if  $x.\text{done} \sim x.\text{cur}$  if  $\neg x.\text{done}$

$x.\text{partial}' = x.\text{rec}$  if  $x.\text{done} \sim x.\text{cur}'$  if  $\neg x.\text{done}$

teda  $\text{partial} = \text{partial}'$

$[\text{init}]$  u  $[\text{partial}]$

d'alej z  $[\text{partial}']$  v  $[\text{cur}']$  máme

$[\text{partial}]$  v;  $t$   $[\text{cur}]$

## 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 spíňa  $R$

# Global snapshots 17

## Distribuovaná architektúra:

- využijeme proces *central*
- *central* pošle všetkým procesom požiadavku zastaviť výpočet
- každý proces po obdržaní tejto správy pošle potvrdenie *centralu* a zastaví sa
- keď *central* dostane potvrdenie od každého procesu, všetkým pošle príkaz na zaznamenanie stavu
- každý proces zaznamená stav a pošle ho *centralu*
- keď *central* dostane od každého jeho stav, pošle všetkým pokyn na pokračovanie
- keď procesy tento pokyn na pokračovanie dostanú, pošlú *centralu* 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 *minus* 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

#### Asynchrónna shared-memory:

- 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  $t$ :

$\langle \ ||x: t \text{ obsahuje } x \wedge \neg x.done ::$

$x.rec, x.done = x, \text{ true if } \langle \exists y: t \text{ obsahuje } y :: y.done \rangle \ || t$

## Global snapshots 20

na začiatku predpokladajme, že zaznamenávanie je iniciované zaznamenávaním jednej špecifickej premennej *first*

$first.rec, first.done := first, \text{true if } begun \wedge \neg first.done$

je isté, že v konečnom čase po zaznamenaní *first* bude zaznamenaná každá premenná, ktorá sa vyskytuje v nejakom príkaze *t* spolu s *first*

Definujme:

$x \text{ je\_spolu s } y \equiv \langle \exists \text{ príkaz } t :: t \text{ obsahuje } x \wedge t \text{ obsahuje } y \rangle$

$\text{je\_spolu}^* \equiv \text{tranzitívny uzáver je spolu}$

Platí:

$x.done \wedge x \text{ je\_spolu s } y \rightarrow y.done$

$x.done \wedge x \text{ je\_spolu}^* \text{ s } y \rightarrow y.done$



# 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  $\langle \parallel x:: x.done = false \rangle$

transform  $\forall t$

$\langle \parallel x: t$  obsahuje  $x \wedge \neg x.done::$

$x.rec, x.done = x, \text{true if } \langle \exists y: t$  obsahuje  $y:: y.done \rangle \rangle$

$\parallel t$

add

$first.rec, first.done = first, \text{true if } begun \wedge \neg first.done \rangle$

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  $v$  prijme 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  $\Rightarrow W$

$W \rightarrow$  claim

## Detekovanie stabilných vlastností 3

- $rec$  .....zaznamenaný stav
- $W(s)$  hodnota  $W$  v stave  $s$

Definujme:

$$claim \equiv W(rec)$$

(predpokladajme, že počiatočná hodnota  $rec$  je taká, že  $\neg W(rec)$  )

## Detekovanie stabilných vlastností 4

Riešenie:

$\text{rec} := \text{globálny stav výpočtu if } \neg W(\text{rec})$

t.j. z času na čas sa zaznamenáva rec až kým neplatí  $W(\text{rec})$

## Detekovanie stabilných vlastností 5

Dôkaz správnosti stratégie, t.j. toho, že platí:

$W(\text{rec})$  detects  $W$

( $W$ - hodnota  $W(\text{st})$  pre momentálny stav  $\text{st}$ )

Dôkaz invariantu (invariant  $W(\text{rec}) \Rightarrow W$ )

každý stav  $\text{st}$  po zaznamenaní je dosiahnuteľný z  $\text{rec}$  i.e. existuje  $v$

$[\text{rec}] v [\text{st}]$



## Detekovanie stabilných vlastností 6

Keďže  $W$  je stable tak pre každé dva stavy  $s_1, s_2$  také, že  $s_2$  je dosiahnuteľné z  $s_1$  platí:

$$W(s_1) \Rightarrow W(s_2)$$

a odtiaľ

$$W(\text{rec}) \Rightarrow W(s)$$

pre všetky stavy  $s$ , ktoré sa vyskytujú po ukončení zaznamenávania

# Detekovanie stabilných vlastností 7

Dôkaz progress podmienky ( $W \rightarrow W(\text{rec})$ )

ak zaznamenávanie je iniciované v stave keď  $W$  platí, tak  $W$  bude platiť aj v stave  $\text{rec}$  (keďže je  $W$  stabilné)

Platí  $W$  ensures  $W(\text{rec})$

keďže

$\{W \wedge \neg W(\text{rec})\}$

$\text{rec} := \text{globálny stav výpočtu if } \neg W(\text{rec})$

$\{W(\text{rec})\}$

# 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.:

$v.\text{idle} := v.\text{idle} \wedge u.\text{idle}$

aktívny proces sa môže kedykoľvek stať idle

## Detekcia terminácie

Program R0

assign

$\langle \square u, v : (u, v) \in E :: v.\text{idle} := v.\text{idle} \wedge u.\text{idle} \rangle$

$\square \langle \square u : u \in V :: u.\text{idle} := \text{true} \rangle$

end

terminácia nastane, keď všetky procesy sú idle, t.j.

$W \equiv \langle \wedge u : u \in V :: u.\text{idle} \rangle$

claim detects  $W$

## Detekcia terminácie

Program R1

initially claim = false

add

claim :=  $\langle \bigwedge u : u \in V :: u.\text{idle} \rangle$

tento program síce detekuje termináciu ale je vhodný len pre sekvenčné architektúry

# Detekcia terminácie

Asynchrónna shared-memory architektúra

Stratégia:

invariant  $d = \{u \mid u.\text{idle}\}$

claim detects  $(d=V)$

Program R2

initially claim = false  $\parallel d = \{u \mid u.\text{idle}\}$

assign

$\langle \parallel u, v : (u, v) \in E :: v.\text{idle} := v.\text{idle} \wedge u.\text{idle}$

$\parallel d := d - \{v\}$  if  $\neg u.\text{idle} \rangle$

$\parallel \langle \parallel u : u \in V :: u.\text{idle} := \text{true} \parallel d := d \cup \{u\} \rangle$

$\parallel \text{claim} := (d=V)$

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  $\square$   $b = \emptyset$   $\square$   $\langle \square u : u.\text{delta} = \emptyset \rangle$

assign

$\langle \square u, v : (u, v) \in E :: v.\text{idle} := v.\text{idle} \wedge u.\text{idle}$

$\parallel u.\text{delta} := u.\text{delta} \cup \{v\}$  if  $\neg u.\text{idle} \rangle$

$\square \langle \square u : u \in V :: u.\text{idle} := \text{true} \rangle$

$\square \langle \square u : b, u.\text{delta} := b \cup \{u\} - u.\text{delta}, \emptyset$  if  $u.\text{idle} \rangle$

$\square \text{claim} := (b = V)$

end



# Detekcia terminácie

## Distribučovaná 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), tail((u,v).c)  
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 možnosti ako detekovať prázdnotu vychádzajúcich kanálov:
  - proces dostane potvrdenie o prijatí každej poslanej správy
  - proces pošle marker a dostane potvrdenie o jeho prijatí
- u.e detects ....  $\langle \wedge v : (u,v) \in E :: (u,v).c = \text{null} \rangle$

## Detekcia terminácie

Program

initially claim = false  $\square$  b = empty  $\square$   $\langle \square$  u.delta = empty  $\rangle$

assign

{u pošle správu m cez kanál (u,v).c ak u je aktívny}

$\langle \square$ (u,v) :: (u,v).c, u.delta := (u,v).c;m, u.delta  $\cup$  {v} if  $\neg$ u.idle  $\rangle$

{v prijme správu cez kanál (u,v).c }

$\square$   $\langle \square$ (u,v) :: v.idle, m, (u,v).c := false, head((u,v).c),  
tail((u,v).c) if (u,v).c  $\neq$  null  $\rangle$

$\square$   $\langle \square$ u :: u.idle := true  $\rangle$

$\square$   $\langle \square$ u :: b, u.delta := b  $\cup$  {u} - u.delta, empty if u.idle  $\rangle$

$\square$  claim := (b=V)

end

## Detekcia terminácie

Nevýhodou predchádzajúceho riešenia je že každý proces  $u$  musí udržiavať  $u.\text{delta}$ , ktoré môže byť veľké

(neprázdna) množina  $J$  detekujúcich procesov - rôzna od  $V$

$j.\text{flag}$  – boolovská, pre každé  $j \in J$

množinu  $V$  si rozdelia procesy  $z \in J$

$g(v)$  – člen  $J$ , ktorý má na starosti  $v$

ak  $j.\text{flag}$  platí pre všetky  $j$ , tak všetky procesy  $v \in V$  sú idle

## Detekcia terminácie

Program

initially claim = false  $\square$  b = empty  $\triangleright$

assign

$\langle \square(u,v) :: v.\text{idle} := u.\text{idle} \wedge v.\text{idle} \quad || \quad g(u).\text{flag} := g(u).\text{flag} \wedge u.\text{idle} \rangle$   
 $\triangleright$

$\square \langle \square u :: u.\text{idle} := \text{true} \rangle$

$\square \langle \square j :: b, j.\text{flag} := \text{empty}, \text{true} \quad \text{if } \neg j.\text{flag} \rangle$

$\square \langle \square u :: b := b \cup \{u\} \quad \text{if } u.\text{idle} \rangle$

$\square \text{claim} := [(b=V) \wedge \langle \wedge j :: j.\text{flag} \rangle$

end

# Paralelná „Garbage Collection“

Daný program *mutator* a graf s pevne danou množinou vrcholov.

Hrany môžu byť pridané alebo odobrané mutatorom.

Jeden z vrcholov sa volá *root*.

*Vrcholy dosiahnuteľné od root sa volajú food.*

*Všetky vrcholy, čo nie sú food, sú garbage.*

*Vrcholy, ktoré sú garbage v čase keď mutator je inicializovaný sa volajú manure.*



## Paralelná „Garbage Collection“

Dané sú vlastnosti  $\text{add}[u,v]$  a  $\text{del}[u,v]$ .

Hrana z  $u$  do  $v$  sa pridá, ak podmienka  $\text{add}[u,v]$  platí. Ak  $\text{add}[u,v]$  platí, tak  $v$  je food.

Hrana z  $u$  do  $v$  sa odoberie, ak  $\text{del}[u,v]$  platí.

## Paralelná „Garbage Collection“

Úloha: modifikovať program mutator tak, aby označil vrcholy tak, že na konci sú všetky food vrcholy označené a všetku manure vrcholy sú neoznačené.

u je označený <---> u.m platí

Vrcholy, ktoré sa stanú garbage (non-manure) môžu byť označené alebo neoznačené.

Boolovska premenná over indikuje, či označovanie skončilo.

## Paralelná „Garbage Collection“

- 1) Ak over platí, tak všetky food vrcholy sú označené a všetky manure vrcholy sú neoznačené.
- 2) Raz bude platiť over.

# Paralelná „Garbage Collection“

Program mutator

always

$R = E^*$  {R je reflexívny a tranzitívny uzáver E}

$\square \langle \square u :: u.\text{food} = R[\text{root}, u] \square u.\text{garbage} = \neg u.\text{food} \rangle$

initially  $\langle \square u :: u.\text{manure} = u.\text{garbage} \rangle$

assign

$\langle \square u, v, ::$

$E[u, v] := \text{true}$  if add[u, v]

$\square E[u, v] := \text{false}$  if dell[u, v]

$\rangle$

end

# Paralelná „Garbage Collection“

Transformovaný mutator musí splňat následovně:

(sp1):

over  $\Rightarrow$   $\langle \forall u : u.\text{manure} :: \neg u.m \rangle \wedge \langle \forall u : u.\text{food} :: u.m \rangle$

(sp2):

true  $\rightarrow$  over

# Paralelná „Garbage Collection“

Vlastnosti mutátoru:

food dosahuje len food (mu4-5)

Žiadna nová cesta sa nevytvorí ku garbage (mu6)

garbage ostáva garbage (mu7)

manure je garbage (mu8)

manure sa dá dosiahnúť len od manure (mu9)

# Paralelná „Garbage Collection“

$\neg E[u,v]$  unless  $\neg E[u,v] \wedge v.\text{food}$  (mu1)

initially  $u.\text{manure} = u.\text{garbage}$  (mu2)

constant  $u.\text{manure}$  (mu3)

Odvozené vlastnosti:

invariant  $R[u,v] \wedge u.\text{food} \Rightarrow v.\text{food}$  (mu4)

invariant  $R[u,v] \wedge v.\text{garbage} \Rightarrow u.\text{garbage}$  (mu5)

$\neg R[u,v]$  unless  $\neg R[u,v] \wedge v.\text{food}$  (mu6)

stable  $v.\text{garbage}$  (mu7)

invariant  $v.\text{manure} \Rightarrow v.\text{garbage}$  (mu8)

invariant  $R[u,v] \wedge v.\text{manure} \Rightarrow u.\text{manure}$  (mu9)

# Paralelná „Garbage Collection“

Jedným riešením by bolo zaznamenať globálny stav (E) a potom vypočítať jeho vlastnosti (R, ...). To chce však veľa pamäte a tú ideme naopak šetriť.

Najprv mutator transformujeme na propagator (udeluje marks).

Pravidlo T:

1) root je označený, 2) všetky vrcholy spojené z označených sú označené

$T = \text{root.m} \wedge \langle \forall u,v : u.m \wedge E[u,v] :: v.m \rangle$

invariant  $v.\text{manure} \Rightarrow \neg v.m$  (pr1)

stable T (pr2)

true  $\rightarrow$  T (pr3)



## Paralelná „Garbage Collection“

Potom propagator transformujeme na marker (zistuje, či to predchádzajúce už skončilo).

over detects T

Ľahko sa ukáže, že marker spíňa (sp1) a (sp2)

# Paralelná „Garbage Collection“

Jednoduché riešenie pomocou algoritmu na zistenie dosiahnuteľnosti nefunguje:

Nech sú na začiatku  $u, v, w$  food a existuje hrana z  $v$  do  $w$  ale nie z  $u$  do  $w$ .

V cykle sa bude robiť:

Dvojica  $(u, w)$  sa skontroluje ale nič sa nezmení, lebo taká hrana nie je.

mutator pridá hranu z  $u$  do  $w$

mutator vymaže hranu z  $v$  do  $w$

Dvojica  $(v, w)$  sa skontroluje ale nič sa nezmení, lebo taká hrana nie je.

mutator pridá hranu z  $v$  do  $w$

mutator vymaže hranu z  $u$  do  $w$

# Paralelná „Garbage Collection“

## Vlastnosti propagatora

- 1) Ak sa pridá hrana z u do v tak v sa označí (pr4).
- 2) Neoznačený vrchol v sa označí ak je food alebo ak existuje označený u z ktorého vedie hrana do v (pr5).
- 3) Označené vrcholy ostávajú označené (pr6).
- 4) Ak je hrana z označeného u do neoznačeného v, tak v sa raz stane označený alebo sa hrana odstráni (pr7).
- 5) Na začiatku je len root označený (pr8).

# Paralelná „Garbage Collection“

$\neg E[u,v]$  unless  $v.m$  (pr4)

$\neg v.m$  unless  $\neg v.m \wedge (v.food \vee < \exists u :: u.m \wedge E[u,v]>)$  (pr5)

stable  $v.m$  (pr6)

$true \rightarrow (u.m \wedge E[u,v] \Rightarrow v.m)$  (pr7)

initially  $u.m = (u = \text{root})$  (pr8)

Vlastnosti (pr4-8) a vlastnosti mutatora implikujú (pr1-3).

# Paralelná „Garbage Collection“

Program propagator (transformácia mutator)

always

$R = E^*$  {R je reflexívny a tranzitívny uzáver E}

□  $\langle \square u :: u.\text{food} = R[\text{root}, u] \square u.\text{garbage} = \neg u.\text{food} \rangle$

initially  $\langle \square u :: u.\text{manure} = u.\text{garbage} \square u.m = (u = \text{root}) \rangle$

assign

< □  $u, v, ::$

□  $E[u, v], v.m := \text{true}, \text{true}$  if add[u, v]

□  $E[u, v] := \text{false}$  if dell[u, v]

□  $v.m := v.m \vee (u.m \wedge E[u, v])$

>

end

# Paralelná „Garbage Collection“

Ideme detekovať termináciu markovania

$b$ -množina párov  $(u,v)$ ,  $ma.flag$ ,  $mu.flag$  (marking a mutator flag – budú označovať kto to mohol pokaziť (false))

invariant

$ma.flag \wedge mu.flag \Rightarrow$

$\langle \forall u,v: (u,v) \in b :: u.m \wedge E[u,v] \Rightarrow v.m \rangle$  (ma2)

$T \rightarrow ma.flag \wedge mu.flag \wedge (b = \forall V)$  (ma3)

$(ma.flag \wedge mu.flag \wedge b = \forall V)$  detects  $T$

over detects  $(ma.flag \wedge mu.flag \wedge b = \forall V)$  (ma4)

# Paralelná „Garbage Collection“

1. Na začiatku je  $b$  prázdne a  $over$  je `false`.
2. Pridá sa  $(u,v)$  do  $b$  ak  $u.m$  a  $E[u,v]$  implikujú  $v.m$ .
3.  $b$  sa vyprázdni a obidve flags sa nastaví `true` ak nejaká flag je `false`.
4. Nastaví sa flag na `false`, ak neoznačený vrchol je označený.
5. Premenná  $over$  sa nastaví `true`, ak sú také obidve flags a  $b$  je  $VV$ .

# Paralelná „Garbage Collection“

Program marker

always

$R = E^*$  {R je reflexívny a tranzitívny uzáver E}  
□  $\langle \text{u} :: \text{u.food} = R[\text{root}, \text{u}] \mid \text{u.garbage} = \neg \text{u.food} \rangle$

initially  $\langle \text{u} :: \text{u.manure} = \text{u.garbage} \mid \text{u.m} = (\text{u} = \text{root}) \rangle$

□  $b = \text{empty}$  {ma.flag, mu.flag sú ľubovoľné}

assign

< □  $\text{u}, \text{v}, ::$

□  $E[\text{u}, \text{v}], \text{v.m} := \text{true}, \text{true}$  if add[u,v]  
□  $\text{mu.flag} := \text{false}$  if add[u,v]  $\wedge \neg \text{v.m}$

□  $E[\text{u}, \text{v}] := \text{false}$  if dell[u,v]  
□  $\text{v.m} := \text{v.m} \vee (\text{u.m} \wedge E[\text{u}, \text{v}])$   
□  $\text{ma.flag} := \text{false}$  if  $\neg \text{v.m} \wedge (\text{u.m} \wedge E[\text{u}, \text{v}])$   
□  $b := b \cup \{(\text{u}, \text{v})\}$  if  $\text{u.m} \wedge E[\text{u}, \text{v}] \Rightarrow \text{v.m}$

>

□  $b, \text{ma.flag}, \text{mu.flag} := \text{empty}, \text{true}, \text{true}$  if  $\neg \text{ma.flag} \vee \neg \text{mu.flag}$

□  $\text{over} := (b = \text{VV}) \wedge \text{ma.flag} \wedge \text{mu.flag}$

end



# Kombinatorické hľadania

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 posledný 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  $\text{sum} = \langle + i : i \in \text{seq} :: A[i] \rangle + A[v]$

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

$$\begin{aligned} \text{FP} &\equiv (\text{sum} = M) \\ &\equiv (\langle + i: i \in \text{seq} :: A[i] \rangle + A[v]) = M \end{aligned}$$

Ak  $v \neq N$  pre pevný bod, tak  $\text{seq};v$  je riešenie

Problém je NP úplný.



Úloha:

Vynásobiť matice  $M_1, M_2, \dots, M_N, N > 0$

tak aby sme minimalizovali počet skalárnych operácií.

Násobenie je asociatívne -  $M_1 \times (M_2 \times M_3) = (M_1 \times M_2) \times M_3$

Ak násobíme matice s dimenziou  $u,v$  a  $v,w$

tak potrebujem  $u \times v \times w$  skalárnych operácií.

Výsledný počet operácií môže závisieť od poradia násobenia.

Riešenie hrubou silou – vyskúšanie všetkých možných postupností násobenia.  
(exponenciálne riešenie)

Lepšie riešenie:

Pre  $N=1$  je riešenie triviálne – počet skalárnych operácií je rovný 0.

Pre  $N > 1$ , posledné maticné násobenie je medzi A a B, kde A je výsledok násobenia  $M_1 \times \dots \times M_k$  a B je výsledok násobenia  $M_{k+1} \times \dots \times M_N$ .

Nech matica  $M_i$  má  $r[i-1]$  riadkov a  $r[i]$  stĺpcov.

Potom A má dimenziu  $r[0] \times r[k]$  a B má dimenziu  $r[k] \times r[N]$ .

Celkový počet potrebných operácií je  $r[0] \times r[k] \times r[N]$  plus počet operácií 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  $M_i \times \dots \times M_j$ .

Program

always

<  $i : 1 \leq i \leq N :: g[i,i] = 0$  >

□

<  $i,j : 1 \leq i < j \leq N :: g[i,j] =$

$\langle \min k : i \leq k < j :: g[i,k] + g[k+1,j] + r[i-1] \times$   
 $r[k] \times r[j] \rangle$

>

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

$\langle \forall i : 1 \leq i \leq N :: g[i,i] = 0 \rangle$

□

$\langle \forall t : 1 \leq t < N$

$\langle \forall i, j : 1 \leq i < j \leq N \wedge j = i+t ::$

$g[i,j] = \langle \min k : i \leq k < j :: g[i,k] + g[k+1,j] + r[i-1] \times r[k] \times r[j] \rangle$

$\rangle$

$\rangle$

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  $u$  -  $\text{dist}(u)$  – je počet hrán na ceste od 0 po  $u$  ak cesta existuje, inak  $\infty$ .

$\text{dist}(0)=0$

Úloha: nájsť  $\text{dist}(u)$  pre všetky vrcholy.

Prehľadávanie do šírky

Vrchol so vzdialenosťou  $k+1$  ma predchodcu so vzdialenosťou  $k$ .

Program

k: integer

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 nasledovníka, odstránime ho a skúsime predposledný 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  $x$ ,  $0 \leq x \leq N$

$(p[x] < 0 \Rightarrow x$  nikdy nebolo pridané do cesty)

$\wedge$

$(0 \leq p[x] < N \Rightarrow x$  je na ceste a

ak  $x \neq 0$ ,  $p[x]$  je predchodca  $x$  na ceste)

$\wedge$

$(N \leq p[x] \Rightarrow x$  bolo odstránené z cesty)

## Program

declare

$u, v$  : integer

$p$ : array[0..N] of integer

always

$v = \langle \min j : 0 \leq j \leq N \wedge E[u, j] \wedge p[j] < 0 :: j \rangle$

initially

$\langle \Box j : 0 \leq j < N :: E[j, N] = \text{true} \rangle$

$\Box u=0 \Box p[0] = 1$                       {pre  $N > 1$  vrchol 0 je na ceste}

$\Box \langle \Box j : 0 < j \leq N :: p[j] = -1 \rangle$

assign

$u, p[v] := v, u$                       if  $v \neq N \wedge u \neq \text{dest}$                       {rozšírenie cesty}

$\Box u, p[u] := p[u], N$                       if  $v = N \wedge u \neq 0 \wedge u \neq \text{dest}$                       {backtrack}

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:

Navrhnuť program, ktorý keď vykonávajú spoľahlivé procesy, tak všetky si nastaví 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 matic (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)

$con^r[u, *] = \langle + x: con^r[u, x] :: 1 \rangle$

## Byzantská dohoda 5

Spec2:

$$(B1) \quad \langle \wedge u: u \neq g :: \neg d^0[u] \rangle \wedge \langle \wedge u, x :: \neg con^0[u, x] \rangle$$

$$(A1) \quad con^r[u, v] = d^{r-1}[v]$$

$$(A2) \quad con^{r-1}[u, x] \Rightarrow con^r[v, x]$$

$$(E1) \quad d^r[u] = d^{r-1}[u] \vee (con^r[u, *] \geq r \wedge con^r[u, g])$$

$$(E2) \quad byz[u] = d^{t+1}[u]$$

Správnost Spec2 (ukážeme, že Spec2  $\Rightarrow$  Spec1)



## Byzantská dohoda 6

Veta: Ak  $g$  je spoľahlivý, tak platí

$$\langle \forall r: r \geq 1 :: d^r[u] = d^0[g] \rangle$$

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  $d^1[u] = false \vee d^0[g] = d^0[g]$

## Byzantská dohoda

$r > 1$ :

$$d^r[u] = d^{r-1}[u] \vee (\text{con}^r[u, *] \geq r \wedge \text{con}^r[u, g])$$

z predpokladu vieme, že:

$$d^{r-1}[u] = d^0[g]$$

z (A1)  $\text{con}^r[u, g] = d^{r-1}[g] (= d^0[g])$

teda  $d^r[u] = d^0[g] \vee (\text{con}^r[u, *] \geq r \wedge d^0[g])$

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  $\forall v$ .

$\neg d^{r-1}[u] \wedge d^r[u]$	<i>/* výber <math>r</math> a <math>u</math> */</i>
$con^r[u, *] \geq r \wedge con^r[u, g]$	<i>/* z (E1) */</i>
$con^r[u, x] \Rightarrow con^{r+1}[v, x]$	<i>/* z (A2) */</i>
$\neg con^r[u, u]$	<i>/* z (A1), <math>\neg d^{r-1}[u]</math> */</i>
$con^{r+1}[v, u]$	<i>/* z (A1), <math>d^r[u]</math> */</i>
$con^{r+1}[v, *] > con^r[u, *]$	<i>/* z predošlých troch */</i>
$con^{r+1}[v, *] \geq r + 1$	<i>/* z predošlých */</i>
$con^{r+1}[v, g]$	<i>/* z 2.riadku a (A2) */</i>
$d^{r+1}[v]$	<i>/* z predošlých dvoch */</i>

## 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

$$\begin{aligned} & \neg d^{r-1}[w] \\ & \neg \text{con}^r[u, w] \quad /* \text{ z (A1) } */ \end{aligned}$$

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

$$\begin{aligned} & \text{con}^r[u, *] \leq t \\ & r \leq \text{con}^r[u, *] \leq t \end{aligned}$$

# 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  $d^r[y]$ )

Spec3: (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

Správnosť Spec3 (ukážeme, že  $\text{Spec3} \Rightarrow \text{Spec2}$ )

Z (A3) vieme, že

$$(D1) \quad \langle \exists w :: \text{obs}^r[w, x] \rangle \vee \text{sum}^r[u, x] \leq t$$

$$(D2) \quad \langle \forall w :: \text{obs}^r[w, x] \rangle \Rightarrow \text{sum}^r[u, x] > 2.t$$

$$(D3) \quad \text{sum}^r[u, x] > 2.t \Rightarrow \text{sum}^r[v, x] > t$$

Dôkaz D3:

$$\text{sum}^r[u, x] > 2.t \Rightarrow \langle +w: \text{obs}^r[w, x] :: 1 \rangle > t$$

$$\text{sum}^r[v, x] \geq \langle +w: \text{obs}^r[w, x] :: 1 \rangle > t$$

# Byzantská dohoda 12

Lema 1:  $\langle \forall r: r \geq 0 :: obs^{r+1}[u, v] = d^r[v] \rangle$

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:  $\langle \forall r: r \geq 0:: con^r[u, v] = obs^r[u, v] \rangle$

Dôkaz: indukciou cez  $r$

$r = 0$ :

$$\begin{aligned} sum^0[u, v] &\leq t \\ \neg con^0[u, v] \end{aligned}$$

/\* z (B2) \*/

/\* 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  $\uparrow$ \*/

$$obs^r[u, v] = sum^r[u, v] > 2.t$$

/\* z  $\uparrow$  a (D1), (D2) \*/

$$obs^r[u, v] = con^r[u, v]$$

/\* z (E5) \*/

# Byzantská dohoda 14

Veta:  $\langle \forall r: r \geq 1 :: con^r[u, v] = d^{r-1}[v] \rangle$

Dôkaz: z liem 1 a 2

Veta:  $\langle \forall r: r \geq 0 :: con^r[u, x] \Rightarrow con^{r+1}[v, x] \rangle$

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

Spec4: (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  $\langle +w: obs^r[w, x]:: 1 \rangle$  o viac než  $t$

Tým je dokázaná správnosť Spec4 (lebo Spec4  $\Rightarrow$  Spec3).

# Byzantská dohoda 17

Program Byzantská dohoda  
always

- $\langle \Box y: y \neq g :: d^0[y] = \text{false} \rangle$  (B1)
- $\Box \langle \Box y, x :: \text{obs}^0[y, x] = \text{false} \rangle$  (B2)
- $\Box \langle \Box r, y: 1 \leq r \leq t + 1 ::$   
 $d^r[y] = d^{r-1}[y] \vee (\text{con}^r[y, *] \geq r \wedge \text{con}^r[y, g]) \rangle$  (E1)  
 $\{ \text{„con}^r[y, *]” \text{ je skratka za } \langle + z: \text{con}^r[y, z] :: 1 \rangle \}$
- $\Box \langle \Box y :: \text{byz}[y] = d^{t+1}[y] \rangle$  (E2)
- $\Box \langle \Box r, y, x: 0 \leq r \leq t + 1 ::$   
 $\text{obs}^{r+1}[y, x] = (\text{obs}^r[y, x] \vee \text{sum}^r[y, x] > t$   
 $\vee \text{val}^r[y, x])$  (E3)  
 $\Box \text{val}^r[y, x] = d^r[x]$  (E4)  
 $\Box \text{con}^r[y, x] = \text{sum}^r[y, x] > 2.t \rangle$  (E5)
- $\Box \langle \Box r, y, z, x: 0 \leq r \leq t + 1 :: \text{robs}^r[y, z, x] = \text{obs}^r[z, x] \rangle$  (E6)
- $\Box \langle \Box r, y, z, x: 0 \leq r \leq t + 1 ::$   
 $\text{sum}^r[y, x] = \langle + z: \text{robs}^r[y, z, x] :: 1 \rangle \rangle$  (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

```
<  $\forall y: y \neq g :: d^0[y] = \text{false}$  >  
<  $\forall y, x :: \text{obs}^0[y, x], \text{val}^0[y, x] = \text{false}, d^0[x]$  >  
<  $\forall x, y, z :: \text{robs}^0[y, z, x] = \text{obs}^0[z, x]$  >  
<  $\forall y, z, x :: \text{sum}^0[y, x] = \langle +z: \text{robs}^0[y, z, x] :: 1 \rangle$  >
```

For r = 1 to t+1

do

```
<  $\forall y, x: \text{obs}^r[y, x] = (\text{obs}^{r-1}[y, x] \vee \text{sum}^{r-1}[y, x] > t \vee \text{val}^{r-1}[y, x])$  >
```

```
<  $\forall y, z, x: \text{robs}^r[y, z, x] = \text{obs}^r[z, x]$  >
```

```
<  $\forall y, z, x: \text{sum}^r[y, x] = \langle +z: \text{robs}^r[y, z, x] :: 1 \rangle$  >
```

```
<  $\forall y, x: \text{con}^r[y, x] = \text{sum}^r[y, x] > 2.t$  >
```

```
<  $\forall y: d^r[y] = d^{r-1}[y] \vee (\text{con}^r[y, *] \geq r \wedge \text{con}^r[y, g])$  >
```

```
<  $\forall y, x: \text{val}^r[y, x] = d^r[x]$  >
```

od

```
<  $\forall y: \text{byz}[y] = d^{t+1}[y]$  >
```

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 pozostáva z:

- $S$
- $C \Rightarrow \text{false}$  (neplatí  $C$ )

Ak vykonanie ukáže, že je to nekonzistentné, tak  $C$  vyplýva z  $S$



# Logické programovanie

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]  $\vee$  rád [Mária,Ján] => šťastný[Ján]

(rád [Ján, Mária]  $\wedge$  láskavý[Ján])  $\vee$

(príjemný[Ján]  $\wedge$  silný[Ján]) => rád[Mária, Ján]

pekná[Mária] => rád [Ján, Mária]

? šťastný [Ján]

# Logické programovanie

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  $b \Rightarrow \text{false}$  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

Doteraz sme uvažovali konečný prehľadávací priestor

Uvažujeme funkcie a relácie.

Každú budem reprezentovať ako množinu, do ktorej možno len pridávať a nie uberať.

Príklad:

$$f(x,y) = (x \neq y)$$

$$\text{a nech } t(x,y,z) = (x * y = z)$$

$$\text{true} \Rightarrow f(0,1)$$

$$f(x,v) \wedge t(x+1,v,u) \Rightarrow f(x+1,u)$$

Pridáme záver  $f(n,m) \Rightarrow \text{false}$

# Logické programovanie

UNITY:

$f, t \dots F, T$  množiny

$b \Rightarrow f(x, y)$  prepíšeme ako

$F := F \cup \{(x, y)\}$  if  $b$

$b(x, y, z) \Rightarrow f(x, y)$  prepíšeme ako

$F := F \cup \langle \cup x, y, z: (x, y, z) \in B :: (x, y) \rangle$

Program môže skončiť ak pre nejaké  $m, (n, m) \in F$



## Logické programovanie

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 (distribúované počítanie)

# Program Composition Notation

Elementárny blok:

priradenie, volanie programu napísaného v PCN, C, Pascal, Fortran, ....

Zložený blok:

{; <blok><sup>1</sup>} |,

{|| <blok><sup>1</sup>} |,

{▯ <blok><sup>1</sup>} |,

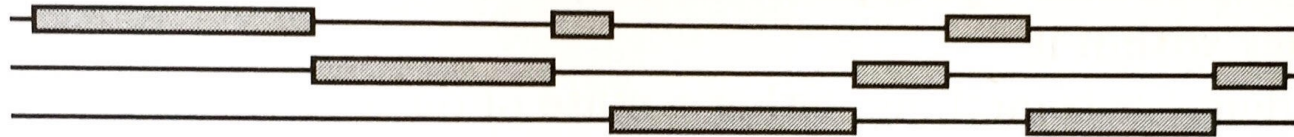
{? <guard -> blok><sup>1</sup>}

Progarm-call@location

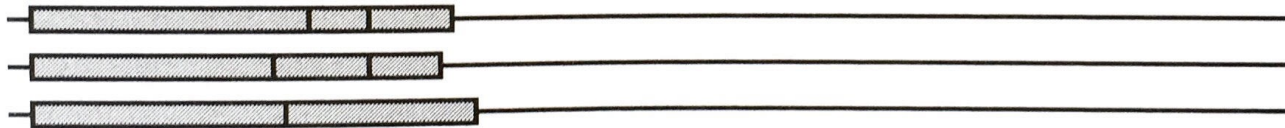
(lokácia – premenná, číslo, relatívna lokácia)

Počítače – 0, ..., n

# Hardvér



Concurrent, non-parallel execution



Concurrent, parallel execution

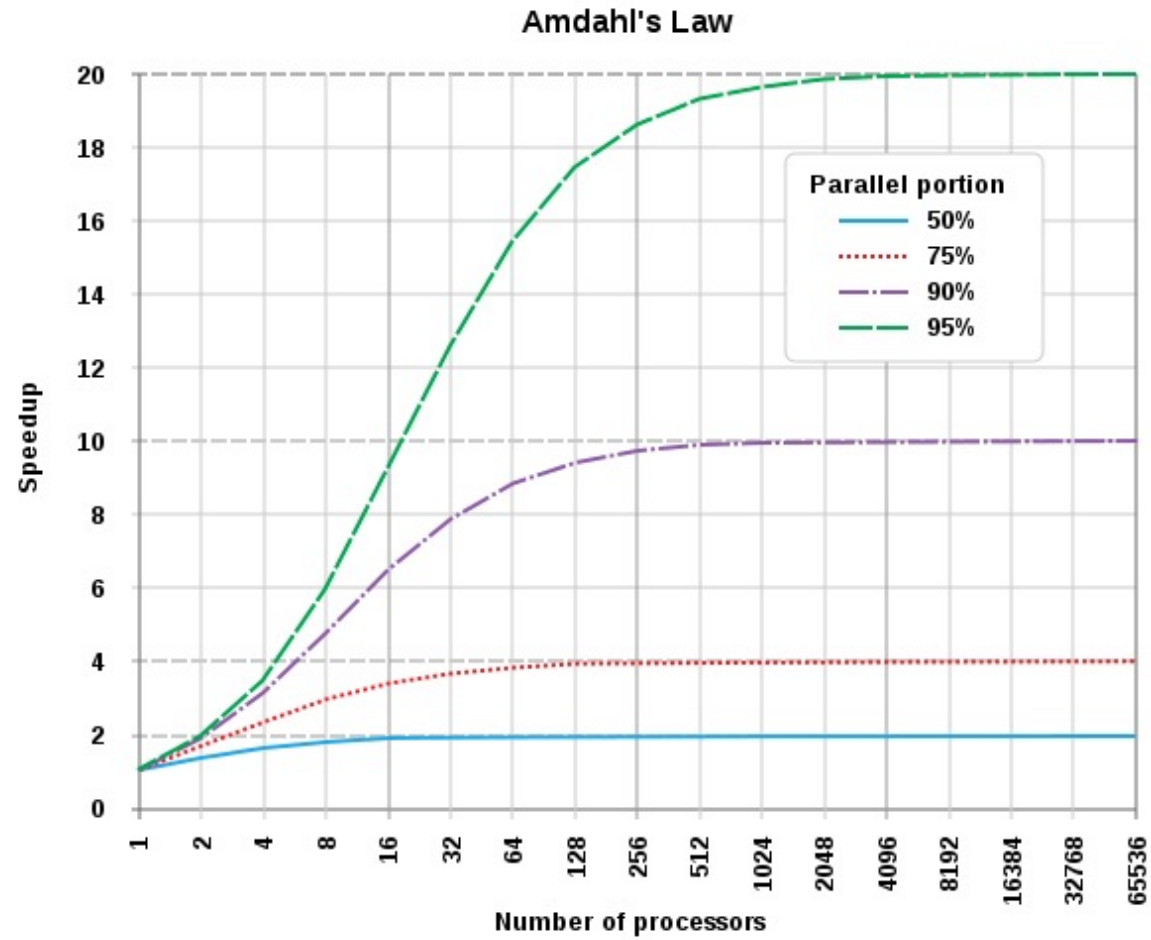
# Amdahlov zákon

Čas s jeným procesorom  $t_1$

Čas s  $p$  procesormi  $t_p$

$$t_p = \frac{t_1}{p}$$

# Amdahlov zákon



# Gustafsonov zákon

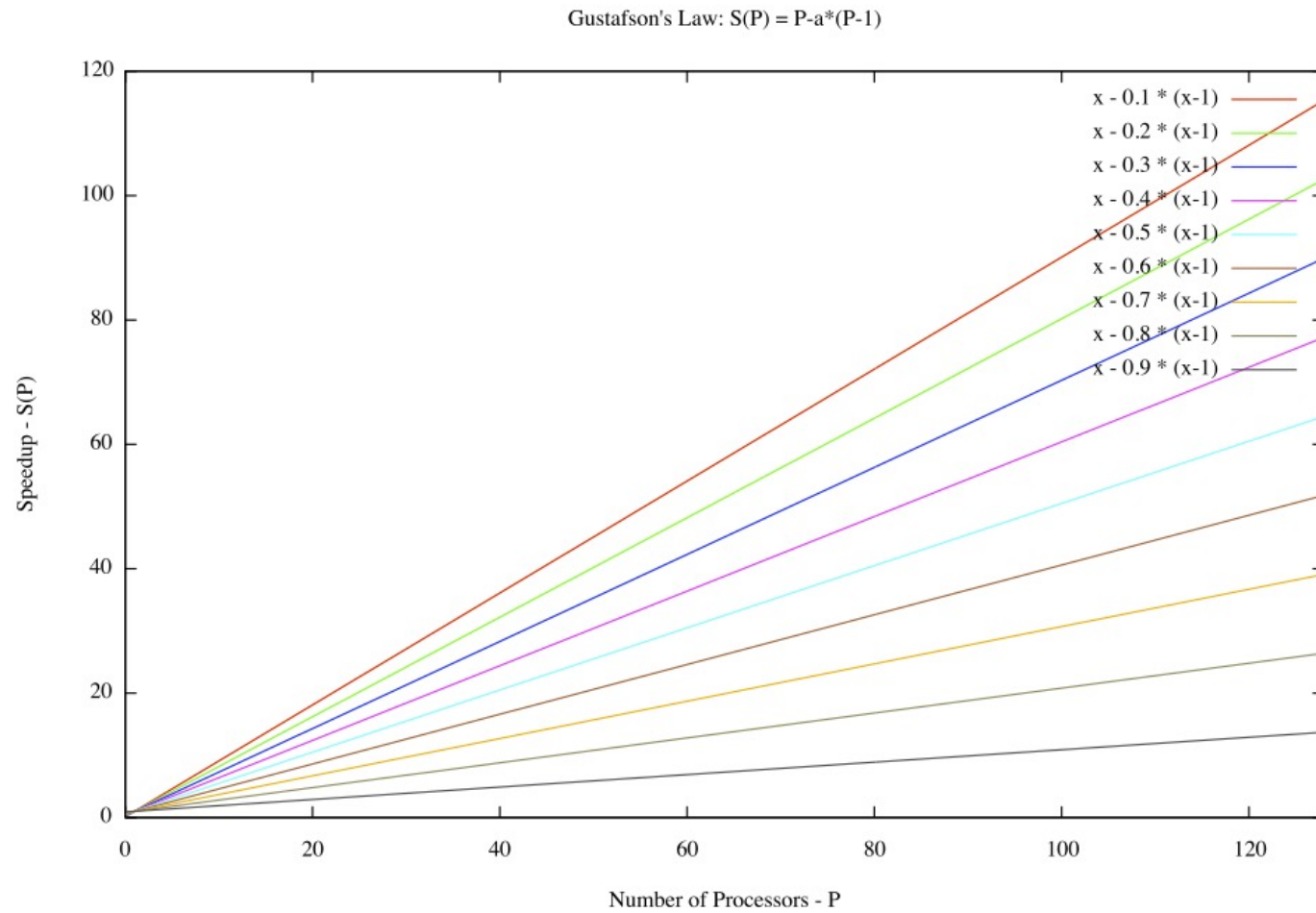
Zrýchlenie S

Získané použitím N procesorov (miesto jedného)

Pri seriovej časti s

$$S = N + (1 - N) \cdot s$$

# Gustafsonov zákon



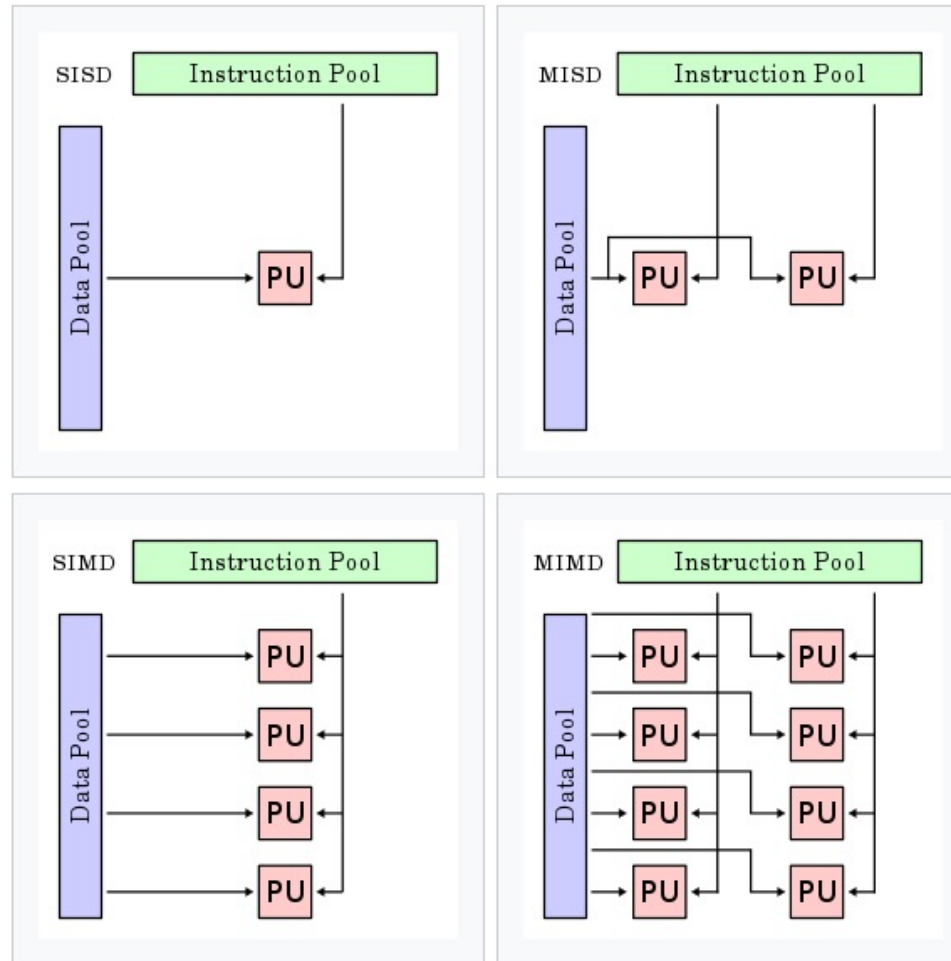


# Hardvér

## Flynnové delenie

- 1 Single instruction stream single data stream (SISD)
- 2 Single instruction stream, multiple data streams (SIMD)
- 3 Multiple instruction streams, single data stream (MISD)
- 4 Multiple instruction streams, multiple data streams (MIMD)
- 5 Single instruction, multiple threads (SIMT)

# Hardvér



# Hardvér

- Viac jadrové procesory
- Symetrický multiprocessing, SMP
- Asymetrický multiprocessing, ASMP
- Počítačový cluster (väčšina z TOP500)
- Grid computing
- General-purpose computing on graphics processing units (GPGPU)
- Vektorový procesor

# Procesové algebry

Sekvenčné komunikujúce procesy  
Synchronne posielanie správ

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...

časové, pravdepodobnostné, stochastické PA, iné komunikačné mechanizmy, lokality, ...

Jazyky odvozené z procesových algebrií alebo využívajúcich procesové algebrií

- 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 plus abstraktné 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 (rozšírené CSP, ...),
- Casper (CSP),
- mCRL2 (kombinácia viacerých PA)

# Actorové modely

actor môže robiť vlastné rozhodnutie, vytvoriť ďalších actorov, posilať správy a prijímať správy. Iných actorov môže ovplyvniť len poslaním správy.

Asynchrónne posielanie správ

Axum

Elixir (runs on the Erlang VM)

Erlang

Janus

Red

SALSA

Scala/Akka (toolkit)

Smalltalk

# Dataflow programming

Program – orientovaný graf, vrcholy reprezentujú operácia nad dátami, ktoré do nich prichádzajú, výsledok je na výstupe a je zároveň vstupom pre ďalšie operácie

CAL

E Joule (tiež distribuovaný)

LabView

Lustre

Preesm

Signal

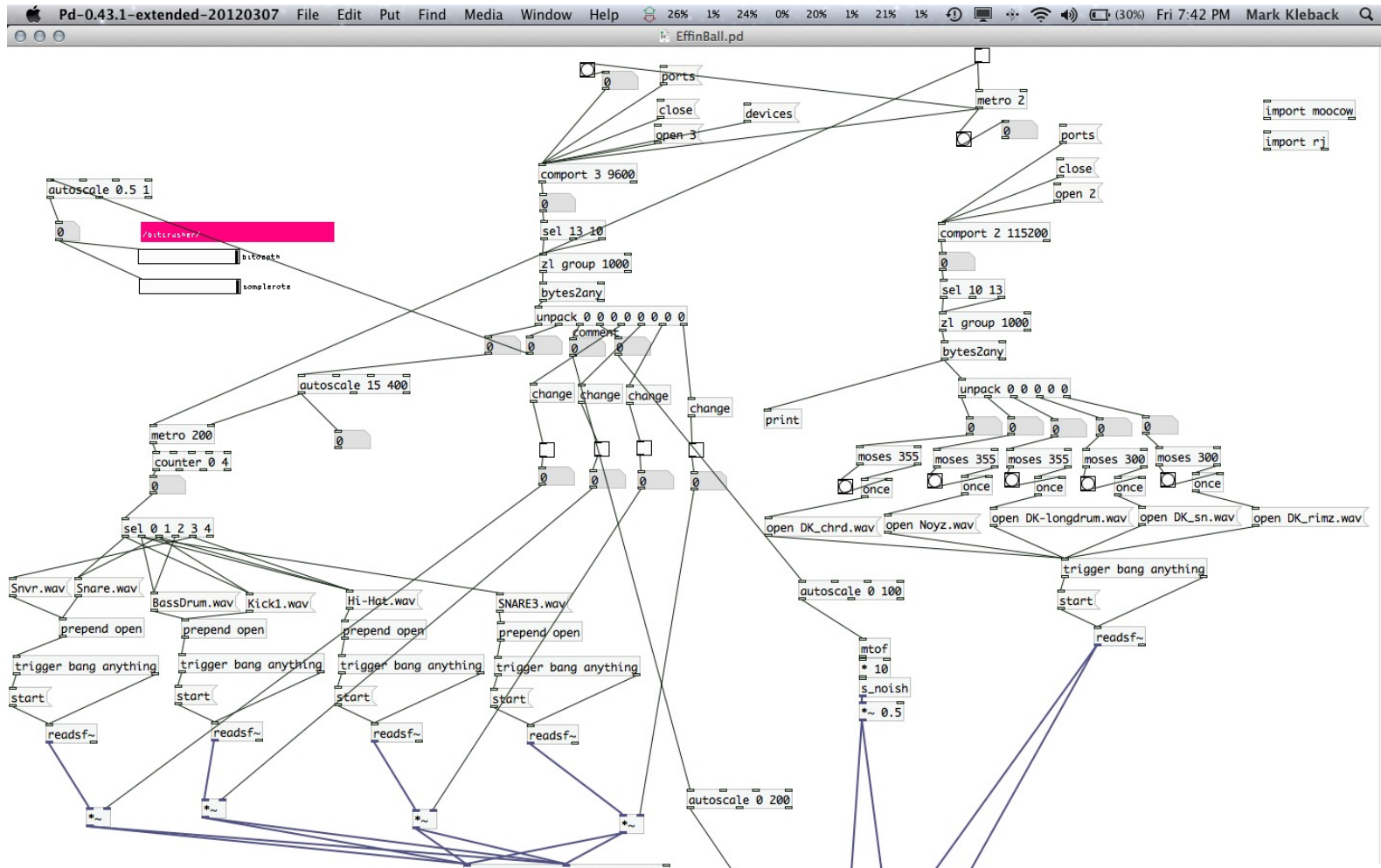
SISAL

BMDFM

Pure data



# Pure data



# Event-driven programming

Esterel

SystemC

SystemVerilog

Verilog

Verilog-AMS

VHDL

# Jazyky pre distribuované programovanie

Bloom

Hermes

Julia

Limbo

MPD

Oz Sequoia

SR

# Multi-threaded

Parallel C/C++

Cilk

Cilk Plus

C#

Clojure

Fork – programming language for the PRAM model.

Java

ParaSail

Rust

SequenceL

# APIs/frameworks

Rôzna podpora paralelizmu

Apache Hadoop

Apache Spark

Apache Flink

Apache Beam

CUDA

OpenCL

OpenHMPP

OpenMP for C, C++ a Fortran

# Partitioned global address space

globálna pamäť, ktorá je rozdelená na partície pre jednotlivé procesy

Chapel

Coarray Fortran

Fortress

High Performance Fortran

Titanium

Unified Parallel C

X10

ZPL

## Pages in category "Concurrent programming languages"

---

The following 80 pages are in this category, out of 80 total. This list may not reflect recent changes ([learn more](#)).

•

- List of concurrent and parallel programming languages

### A

- Actor-Based Concurrent Language
- ActorFoundry
- AgentSheets
- Alef (programming language)
- Algorithmic skeleton
- AmbientTalk
- Ateji PX
- Axum (programming language)

### C

- C $\omega$
- C\*
- Chapel (programming language)
- Cilk
- CMS Pipelines
- Concurrent Collections
- Concurrent Euclid
- Concurrent ML
- CS-4 (programming language)
- Curry (programming language)

### D

- DAP FORTRAN
- Dart (programming language)
- Dataflow programming

### E

- E (programming language)
- Ease (programming language)
- Erlang (programming language)

### F

- Flow-based programming
- Fortress (programming language)

### G

- Go (programming language)
- Go! (programming language)

### H

- High Performance Fortran

### I

- Intel Parallel Building Blocks

### J

- JADE (programming language)
- Janus (concurrent constraint programming language)
- Java (programming language)
- JoCaml
- Join Java
- Joule (programming language)
- Joyce (programming language)

### L

- Limbo (programming language)
- Linda (coordination language)
- Lucid (programming language)
- Lynx (programming language)

### M

- Mesa (programming language)
- MPD (programming language)
- MultiLisp

### N

- NESL
- Newsqueak
- Nim (programming language)

### O

- Occam (programming language)
- Occam- $\pi$
- Orc (programming language)
- Oz (programming language)

### P

- Parallel programming model

- ParaSail (programming language)
- Parlog
- PL/I
- PLEX (programming language)
- Portable Standard Lisp

### R

- Rust (programming language)

### S

- SALSA (programming language)
- Scala (programming language)
- SCOOP (software)
- SequenceL
- Sieve C++ Parallel Programming System
- SISAL
- SPARK (programming language)
- Split-C
- SR (programming language)
- \*Lisp
- Strand (programming language)
- SuperPascal

### T

- Transterpreter

### U

- Umple
- Unified Parallel C

### V

- Visual Prolog

### X

- X10 (programming language)
- XC (programming language)
- XMTC
- XProc

### Z

- ZPL (programming language)

## Paralelizačné nástroje:

YUCCA

Par4All

Cetus

PLUTO

Polaris compiler

Intel C++ Compiler

Intel Advisor

AutoPar

iPat/OMP

Vienna Fortran compiler(VFC)

SUIF compiler

Omni OpenMP Compiler

Timing-Architects Optimizer

TRACO

SequenceL

6OMP2MPI

OMP2HMPP

emmtrix Parallel Studio



# Temporálne logiky

proпозиčná 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  $(\mathbb{N}, <)$
- čas:
  - diskrétny
  - počiatočný okamih
  - nekonečný
- $AP$ : atomické propozície (ozn.  $P, Q, \dots$ )
- štruktúra lineárneho času: trojica  $M = (S, x, L)$ , kde
  - $S$  – množina stavov
  - $x: \mathbb{N} \rightarrow S$  – postupnosť stavov
  - $L: S \rightarrow \mathbf{P}(AP)$  – určuje pre daný stav, ktoré atomické propozície v tomto stave platia (teda ktoré  $AP$  sú true)

# Temporálne logiky

- označenie:
  - postupnosť stavov  $x = s_0, s_1, s_2, s_3, \dots$
  - definujeme  $x^i = s_i, s_{i+1}, s_{i+2}, s_{i+3}, \dots$  (teda  $x = x^0$ )
- základné temporálne operátory:
  - $\diamond p$  – eventually  $p$  (raz určite  $p$ )(textovo  $Fp$ )
  - $\square p$  – vždy  $p$  (textovo  $Gp$ )
  - $\bigcirc p$  – nasledujúci krát  $p$  (textovo  $Xp$ )
  - $p \cup q$  –  $p$  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  $p \cup q, Xp$  sú formuly

zavedieme označenia:

$$\begin{array}{ll} Fp = \text{true} \cup p & \diamond p \\ Gp = \neg F\neg p & \square p \\ F^\infty p = GFp & \text{(nekonečne veľa krát)} \\ G^\infty p = FGp & \text{(skoro všade)} \end{array}$$

# Temporálne logiky

## Sémantika:

- $x \models P$  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  $x \models p$
- $x \models (p \cup q)$  iff  $\exists j (x^j \models q$  and  $\forall k < j: x^k \models p)$
- $x \models Xp$  iff  $x^1 \models p$
- $x \models Fp$  iff  $\exists j (x^j \models p)$
- $x \models Gp$  iff  $\forall j (x^j \models p)$
- $x \models F^\infty p$  iff  $\forall k \exists j \geq k (x^j \models p)$
- $x \models G^\infty p$  iff  $\exists k \forall j > k (x^j \models p)$

## Príklady:

$p \Rightarrow Fq$

$G(p \Rightarrow Fq)$

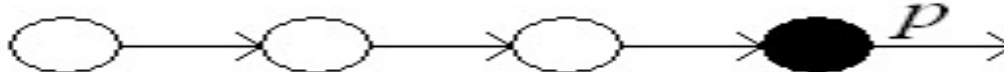
$p \wedge G(p \Rightarrow Xp) \Rightarrow Gp$

ak  $p$  platí teraz, tak raz bude platiť  $q$   
vždy keď platí  $p$ , tak raz začne platiť aj  $q$   
temporálna formulácia indukcie

# Temporálne logiky

Príklady použitia operátorov  $F$ ,  $G$ ,  $X$  a  $U$  v propozičnej lineárnej temporálnej logiky

$Fp$



$Gp$



$Xp$



$p \cup q$



# 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 nexttime) ak existuje následný stav v  $I$  tak v  
ňom platí  $p$

$X_{\exists}p$  (strong nexttime) existuje následný stav v  $I$  a v ňom platí  $p$

$X_{\exists}p \dots \neg X_{\forall} \neg p$

$X_{\forall}p \dots \neg X_{\exists} \neg p$



## Temporálne logiky

- $G-p$  vždy v minulosti platilo  $p$
- $F-p$  niekedy v minulosti platilo  $p$
- $X-p$  naposledy platilo  $p$
- $p \cup \neg q$  niekedy platilo  $q$  a odvtedy platí  $p$

# Temporálne logiky

## Branching (time) temporal logic

- temporálna štruktúra: trojica  $M = (S, R, L)$ , kde  
 $S$  – množina stavov  
 $R \subseteq S \times S$  taká, že  $\forall s \in S \exists t \in S (s, t) \in R$   
 $L: S \rightarrow \mathbf{P}(AP)$  – 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  $x = (s_0, s_1, s_2, s_3, \dots)$ : pre  $\forall i: (s_i, s_{i+1}) \in R$
  - definujeme  $x^i = (s_i, s_{i+1}, s_{i+2}, s_{i+3}, \dots)$

# 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  $p \cup q, Xp$  sú path formuly (P3)

# Temporálne logiky

CTL\* tvoria stavové formuly

CTL tvoria pravidlá (S1), (S2), (S3) a pravidlo (P0):

ak  $p, q$  sú stavové formuly, tak  $p \cup q, Xp$  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)  $M, s_0 \models Ep$  iff  $\exists x$  v  $M$  tak, že  $M, x \models p$   
 $M, s_0 \models Ap$  iff pre  $\forall x$  v  $M$  platí  $M, x \models p$

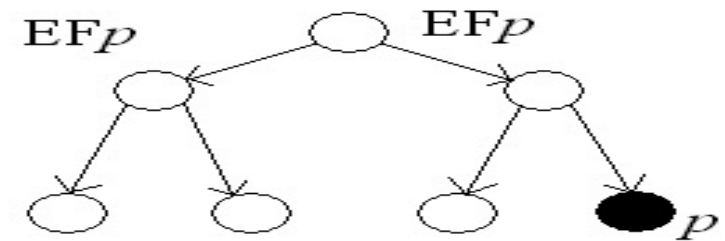
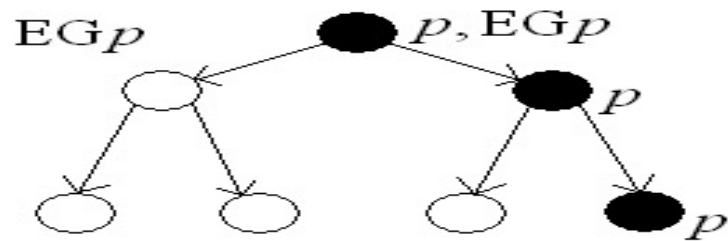
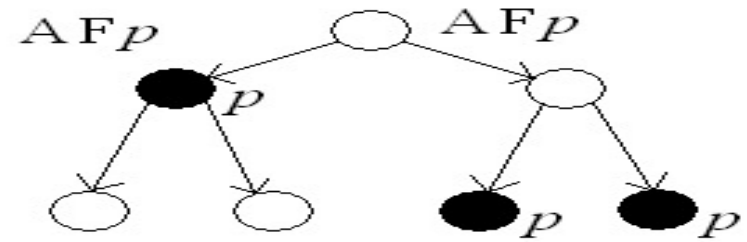
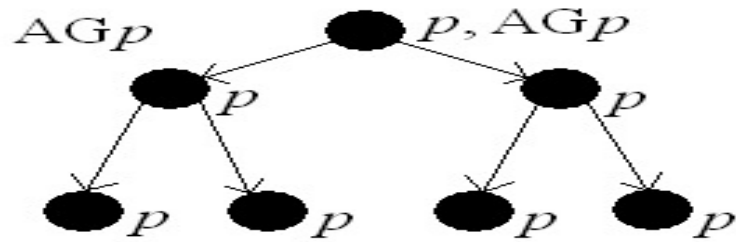
(P1)  $M, x \models p$  iff  $M, s_0 \models p$

(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)  $M, x \models (p \cup q)$  iff  $\exists j$  ( $M, x^j \models q$  a  $\forall k$  ( $k < j$  implikuje  $M, x^k \models p$ )  
 $M, x \models Xp$  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  $\vdash p$ ), 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

## Schémy axióm:

tautológie propozičnej logiky	(Ax1)
$EFp \equiv E(\text{true} \cup p)$	(Ax2)
$AGp \equiv \neg EF\neg p$	(Ax2')
$AFp \equiv A(\text{true} \cup p)$	(Ax3)
$EGp \equiv \neg AF\neg p$	(Ax3')
$EX(p \vee q) \equiv EXp \vee EXq$	(Ax4)
$AXp \equiv \neg EX\neg p$	(Ax5)
$E(p \cup q) \equiv q \vee (p \wedge EXE(p \cup q))$	(Ax6)
$A(p \cup q) \equiv q \vee (p \wedge AXA(p \cup q))$	(Ax7)
$EX \text{ true} \wedge AX \text{ true}$	(Ax8)
$AG(r \Rightarrow (\neg q \wedge EXr)) \Rightarrow (r \Rightarrow \neg A(p \cup q))$	(Ax9)
$AG(r \Rightarrow (\neg q \wedge EXr)) \Rightarrow (r \Rightarrow \neg AFq)$	(Ax9')
$AG(r \Rightarrow (\neg q \wedge (p \Rightarrow AXr))) \Rightarrow (r \Rightarrow \neg E(p \cup q))$	(Ax10)
$AG(r \Rightarrow (\neg q \wedge AXr)) \Rightarrow (r \Rightarrow \neg EFq)$	(Ax10')
$AG(p \Rightarrow q) \Rightarrow (EXp \Rightarrow EXq)$	(Ax11)



# Temporálne logiky

Odvodzovacie pravidlá:

ak  $\vdash p$ , tak potom  $\vdash AGp$  (R1, zovšeobecnenie)

ak  $\vdash p$  a  $\vdash p \Rightarrow q$ , tak potom  $\vdash q$  (R2, modus ponens)

Veta: Deduktívny systém s axiómami (Ax1)–(Ax11) a odvodzovacími pravidlami (R1), (R2) je sound (zdravý, korektný) and complete (úplný) pre CTL.

# Vztáh CTL a MU-kalkulus

$A (p \cup q) \dots \mu Z \ q \vee (p \wedge (\leftrightarrow tt \wedge [-]Z))$

$E (p \cup q) \dots \mu Z \ q \vee (p \wedge \leftrightarrow Z)$

$AF \ p \dots \mu Z \ p \vee (\leftrightarrow tt \wedge [-]Z)$

$EF \ p \dots \mu Z \ p \vee \leftrightarrow Z$

# Verifikácia vlastností systémov vyjadrených pomocou logických formúl

## 1.theorem proving

axiomy logiky+vlastnosti systému/programu |- theorema

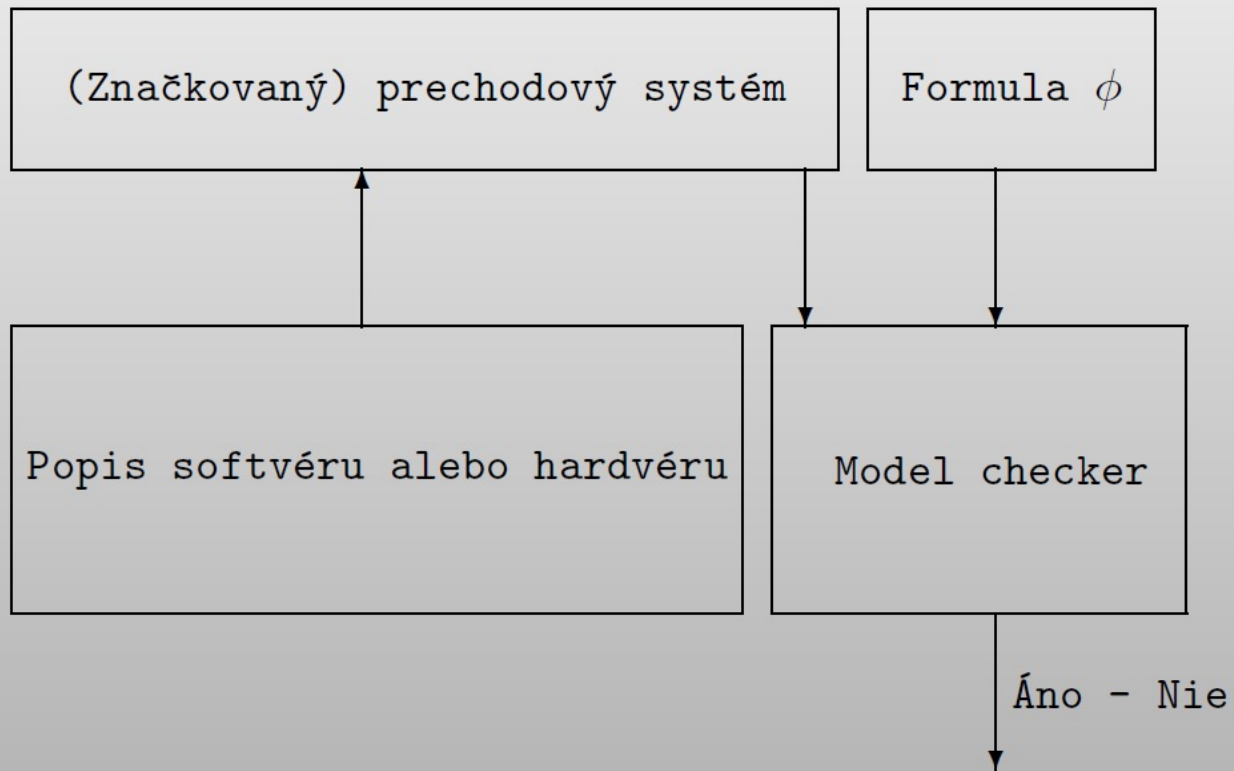
- automatický alebo poloautomatický theorem proving alebo proof verification

- v závislosti od odpovedajúcej logiky sa zložitosť riešenia pohybuje od triviálneho až po nemožné.

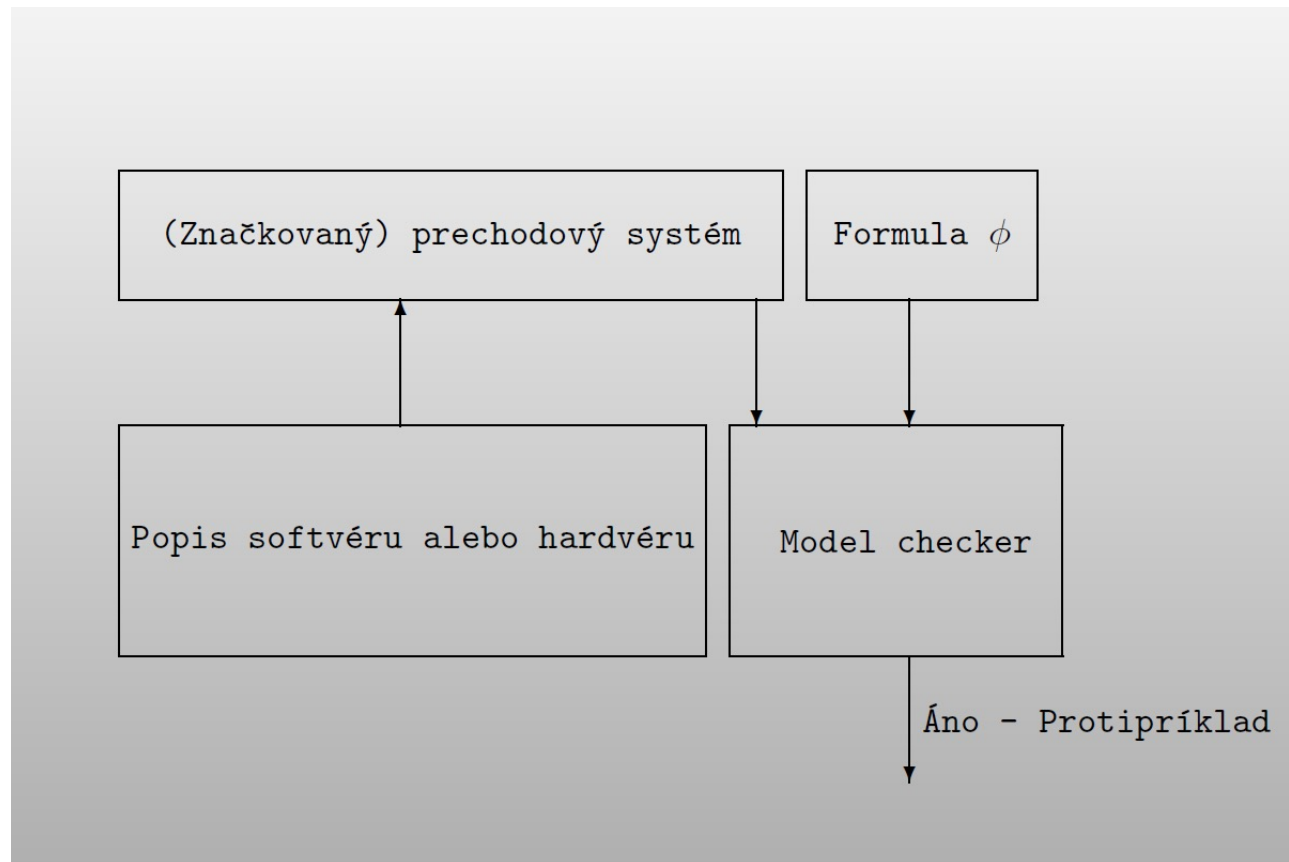
Praktické využitie: návrh a verifikácia integrovaných obvodov, (verifikácia aritmetických operácií, ....) Existuje množstvo softvérových nástrojov.

## 2. model checking

# Model Checking



# Model Checking



Úloha: daný (značkovaný) prechodový systém  $M$  (Kripkeho štruktúra) a formula temporálnej alebo modálnej logiky  $\varphi$ , 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 AFp:

1. ak je vrchol označený  $p$ , tak ho označíme aj AFp
2. (opakuje sa, kým sa dá) označ vrchol s AFp, ak všetci jeho následníci sú označení AFp
3. označ  $\neg$ AFp tie, ktoré nie sú označené AFp