Tomáš Horváth (Ed.)

# Informačné Technológie - Aplikácie a Teória

Zborník príspevkov prezentovaných na pracovnom seminári ITAT
Hotel Magura, Belianske Tatry, Slovensko, september 2012

# ITAT 2012
# Informačné technológie – Aplikácie a Teória

**Zborník príspevkov prezentovaných**
**na konferencii ITAT, september 2012**

**Editor**

Tomáš Horváth
Ústav informatiky
Prírodovedecká fakulta, Univerzita P. J. Šafárika v Košiciach
Šrobárova 2, 041 54 Košice

ISBN 978-80-971144-1-1

Vydala: Slovenská spoločnosť pre umelú inteligenciu
Počet strán: 72

# Predhovor

Dvanásty ročník konferencie **ITAT'12 Informačné Technológie – Aplikácie a Teória** sa konal v hoteli Magura v Monkovej doline Belianskych Tatier pri Ždiari v dňoch 17.–21. septembra 2012.

Konferencia ITAT je tradičným miestom stretnutia informatikov z Českej a Slovenskej republiky s dôrazom na vzájomnú výmenu informácií ako aj upevnenie kontaktov medzi účastníkmi čomu zodpovedá aj vyhradenie dostatočného priestoru pre diskusiu v odbornom aj spoločenskom programe. Príspevky sú prezentované v slovenčine alebo češtine.

Všetky príspevky boli recenzované dvoma recenzentmi. Z 25 zaslaných príspevkov 4 boli prijaté v kategórii *Pôvodné vedecké práce* a 7 ako *Work in progress*, ktoré sú publikované v tomto zborníku. Ďalších 8 článkov zaradených do kategórie *Vybrané pôvodné vedecké práce* sú publikované v CEUR ((`http://ceur-ws.org/`)) zborníku spolu s abstraktom pozvanej prednášky.

Konferenciu ITAT'12 organizovali
– Ústav informatiky Univerzity P. J. Šafárika, Košice
– Matematicko-fyzikální fakulta Univerzity Karlovy v Praze
– Ústav informatiky AV ČR Praha
– Slovenská spoločnosť pre umelú inteligenciu

Týmto sa chcem poďakovať autorom prezentovaných príspevkov, pozvanému prednášajúcemu Jiřímu Klémovi, členom programového výboru, externým recenzentom a organizačnému výboru konferencie na čele s Petrom Gurským.

Konferencia ITAT'12 bola čiastočne podporená grantom VEGA 1/0832/12 a spoločnosťou Profinit (`http://www.profinit.eu/`)

*Tomáš Horváth*

*Pre správne zobrazenie PDF verzie zborníka odporúčame použiť Adobe Reader verzie 9 a novšej.*

## Programový výbor

Tomáš Horváth, (predseda), *Univerzita P. J. Šafárika, Košice, SR*
Radim Bača, *Vysoká škola báňská – Technická univerzita Ostrava, ČR*
David Bednárek, *Univerzita Karlova v Praze, ČR*
Mária Bieliková, *Slovenská technická univerzita v Bratislave, SR*
Jiří Dokulil, *Univerzita Karlova v Praze, ČR*
Jana Dvořáková, *Univerzita Karlova v Praze, ČR*
Peter Gurský, *Univerzita P. J. Šafárika, Košice, SR*
Tomáš Holan, *Univerzita Karlova v Praze, ČR*
Martin Holeňa, *Ústav informatiky, Akademie věd České republiky, Praha, ČR*
Jozef Jirásek, *Univerzita P. J. Šafárika, Košice, SR*
Jana Katreniaková, *Univerzita Komenského v Bratislave, SR*
Rastislav Královič, *Univerzita Komenského v Bratislave, SR*
Michal Krátký, *Vysoká škola báňská – Technická univerzita Ostrava, ČR*
Věra Kůrková, *Ústav informatiky, Akademie věd České republiky, Praha, ČR*
Markéta Lopatková, *Univerzita Karlova v Praze, ČR*
Roman Neruda, *Ústav informatiky, Akademie věd České republiky, Praha, ČR*
Dana Pardubská, *Univerzita Komenského v Bratislave, SR*
Tomáš Plachetka, *Univerzita Komenského v Bratislave, SR*
Martin Plátek, *Univerzita Karlova v Praze, ČR*
Jaroslav Pokorný, *Univerzita Karlova v Praze, ČR*
Karel Richta, *Univerzita Karlova v Praze, ČR*
Gabriel Semanišin, *Univerzita P. J. Šafárika, Košice, SR*
Vojtěch Svátek, *Vysoká škola ekonomická, Praha, ČR*
Roman Špánek, *Ústav informatiky, Akademie věd České republiky, Praha, ČR*
Július Štuller, *Ústav informatiky, Akademie věd České republiky, Praha, ČR*
Peter Vojtáš, *Univerzita Karlova v Praze, ČR*
Jakub Yaghob, *Univerzita Karlova v Praze, ČR*
Filip Zavoral, *Univerzita Karlova v Praze, ČR*

## Organizačný výbor

Peter Gurský, (predseda), *Univerzita P. J. Šafárika, Košice, SR*
Hanka Bílková, *Ústav informatiky, Akademie věd České republiky, Praha, ČR*
Róbert Novotný, *Univerzita P. J. Šafárika, Košice, SR*
Martin Šumák, *Univerzita P. J. Šafárika, Košice, SR*
Lenka Pisková, *Univerzita P. J. Šafárika, Košice, SR*

## Organizácia

**ITAT 2012 – Informačné Technológie – Aplikácie a Teória organizovala**
Univerzita P. J. Šafárika v Košiciach, SR
Ústav informatiky, Akademie věd České republiky, Praha
Matematicko-fyzikální fakulta, Univerzita Karlova v Praze, ČR
Slovenská spoločnosť pre umelú inteligenciu, SR

## Obsah

**ITAT'12**

**Informačné Technológie – Aplikácie a Teória**

# PÔVODNÉ VEDECKÉ PRÁCE

# Algoritmus rýchleho vyhľadávania pohybových vektorov pre kódovanie videa v H.264

Rastislav Adamek, Gabriela Andrejková
Prírodovedecká fakulta
Univerzita Pavla Jozefa Šafárika v Košiciach
Jesenná 5, Košice 041 54, Slovakia

**Abstrakt.** *Spracovanie sekvencií obrázkov videa a ich kódovanie je časovo veľmi náročný problém, ktorého náročnosť tiež závisí od obsahu obrázkov. Článok pojednáva o blokových metódach odhadu pohybu pri kódovaní videa. Navrhnutá metóda používa bloky veľkosti 4x4 a jej základnou myšlienkou je vyhľadávanie pohybových vektorov pomocou polohy hrán pri kódovaní jednotlivých blokov videa. Metóda umožňuje nájsť pohybové vektory presnejšie a rýchlejšie ako pri klasickej známej metóde, ktorá prepočítava všetky možnosti. Vzhľadom na to, že pracujeme s blokmi veľkosti 4x4 zohľadňujeme smer hrán v 5-tich uhloch v rozpätí 90° (so zaokrúhlením). Nami predkladaná metóda je porovnávaná s klasickou štandardnou metódou pomocou vyhodnocovacej funkcie špičkového pomeru signál-šum. Porovnanie je urobené na benchmarkových dátach a výsledky nami navrhnutej metódy sú porovnateľné a v niekto-rých prípadoch lepšie.*

## 1 Úvod

Skupina expertov pre sekvencie pohyblivých obrázkov MPEG (Moving Picture Experts Group) a skupina expertov pre kódovanie videa VCEG (Video Coding Experts Group) vyvinuli nový štandard, ktorý sľuboval lepší výkon ako predchodca MPEG-4 a H.263 štandard, za predpokladu lepšej kompresie video obrázkov. Nový štandard označený AVC (Advanced Video Coding) alebo tiež MPEG-4 je v ITU (International Telecommuni-cations Union) dokumente zapísaný pod pracovným číslom H.264. V predošlých štandardoch (aj v H.264) je jasne definovaný kodek (kóder a dekóder), ale jednotlivé funkčné bloky kodeku boli vylepšené, aby sa získala úspora toku bitov. Jednotlivé vylepšenia základných funkčných elementov (predikcia, transformácia, kvantizácia a entropické kódovanie) spolu dosahujú lepšiu kvalitu pri rovnakom bitovom toku ako v predchádzajúcich štandardoch (MPEG-1, MPEG-2, MPEG-4, H.261, H.263).

Obraz štandardu H.264 môže byť rozdelený do blokov (pravidelných pravouhlých), skupín blokov (rezov) a skupín rezov. Kóder môže používať jedno alebo dve čísla predchádzajúcich kódovaných snímok ako referenciu pohybovo-kompenzačnej predikcie každého inter-kódovaného bloku alebo jeho časti. Snímka je kódovaná ako jeden alebo viac skupín blokov. Blok je vizuálne spojitý, ak hodnoty všetkých jeho pixlov sú takmer rovnaké. Naopak, ak hodnoty pixlov bloku sú rozmanité, tak blok je vizuálne nespojitý. Pojem vizuálnej spojitosti je dôležitý pri reprezentácii blokov. Ak je blok vizuálne nespojitý, tak je možné, že je v ňom hrana. Hrany v bloku majú svoje orientácie a pomocou orientácií hrán je možné reprezentovať štruktúru obrázku. Aj keď H.264 má skonštruované kódery a dekódery, hľadanie ich vylepšení je možné na základe niektorých teoretických poznatkov o spracovaní obrázkov, ktoré sú prezentované v [1-6]. V [2] je uvedený algoritmus pre testovanie zhody blokov používajúci stratégiu zladenia hrán v blokoch na video snímkach kódovaných pomocou H.264. Algoritmus využíva viacnásobné referencie a viacnásobné veľkosti blokov na odhad pohybu.

V článku sa zaoberáme konštrukciou pohybových vektorov. V druhej kapitole sú uvedené teoretické východiská pre konštrukciu pohybových vektorov, v tretej kapitole je opísaný algoritmus predikcie pohybových vektorov pomocou zladenia hrán. V ďalšej časti je uvedené kritérium vyhodnocovania kvality obrazu a závere je zhrnutý prínos a využitie vytvoreného algoritmu.

## 2 Princíp vyhľadávania pohybových vektorov pomocou polohy hrán

Snímka štandardného kodeku H.264 je rozdelená do makroblokov a potom do blokov. Štandardne sa používajú veľkosti blokov 16 × 16 bodov. Tieto bloky sa neprekrývajú. Každý blok je potom ďalej rozdelený do niekoľkých čiastkových blokov s premenlivou veľkosťou v závislosti od toku informácií, čo je dôležité pre optimalizáciu systému.

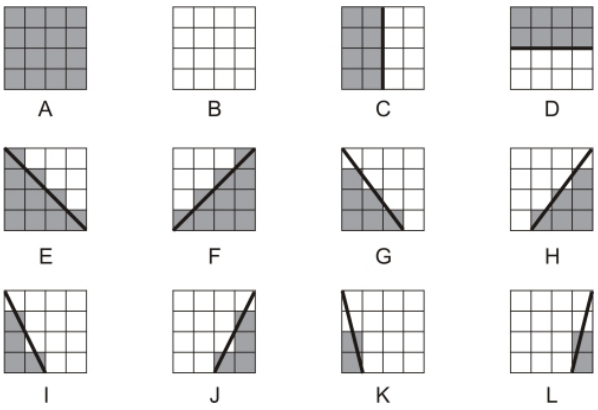
Veľkosti čiastkových blokov môžu byť 16 × 16, 8 × 16, 16 × 8, 8 × 8, 8 × 4, 4 × 8, 4 × 4. My budeme používať bloky 16 × 16, ktoré budú rozdelené do šestnástich čiastkových blokov o veľkosti 4 × 4 bodov. Pre tieto bloky budeme hľadať pohybové vektory pomocou polohy hrán v daných blokoch. Našou úlohou je nájsť najpodobnejší blok z predchádzajúcej a nasledujúcej snímky (na koľko pixlov daná zhoda súhlasí). Samozrejme, že podobnosť je hodnotená na základe určitých kritérií. Konkrétna funkcia na zisťovanie podobnosti blokov v kódovaní videa je SAD (Sum of absolute differences – Suma absolútnych rozdielov) t. j.

$$SAD_{x,y}(u,v) = \sum_{i=1}^{N-1}\sum_{j=1}^{N-1} |I_t(x+i, y+j) - I_{t-1}(x+u+i, y+v+j)| \quad (1)$$

kde $\vec{u}$ = (u, v) je posunutie a | $I_t$ (x + i, y + j) - $I_{t-1}$ (x + u + i, y + v + j) | sa nazýva výsledná snímka rozdielov. Keďže táto technika prehľadáva všetky možnosti, tak je v praxi príliš pomalá a náročná na procesor (CPU). My sa pokúsime tento proces urýchliť algoritmom vyhľadávania pohybových vektorov pomocou zladenia hrán.

Na najnižšej úrovni počítačového videnia, môžeme zo snímky získať rôzne grafické útvary, ako sú napríklad hrany bez toho, aby sme poznali obsah obrazu. My nepotrebu-

jeme poznať detailne celý objekt a nepotrebujeme poznať ani jeho obsah. Keďže obraz je rozdelený na jednotlivé neprekrývajúce sa bloky, budeme sa zaoberať práve nimi. Každý blok je kódovaný ako homogénny blok alebo hranový blok. V našom prípade budeme zohľadňovať možnosti hranových blokov.




**Obr. 1.** A) hranový blok, B) homogénny blok, C) 90° horizontálna hrana, D) 90° vertikálna hrana, E) 45° hrana, F) 45° hrana, G) 39,6° hrana, H) 39,6° hrana, I) 26,5° hrana, J) 26,5° hrana, K) 14° hrana, L) 14° hrana

Spojitý dvojrozmerný hranový model špecifikujú štyri parametre, dve hodnoty šedej farby *h1* a *h2*, vzdialenosť hrany od stredu *l*, a uhol $\theta$ určujúci orientáciu hrany bloku *B*, ktorý je uvedený na Obr. 2. Vzdialenosť hrany *l* je definovaná ako najkratšia vzdialenosť od stredu bloku po hranu v rozmedzí - 2 až + 2. Uhol $\theta$ označuje smer hrany a jeho rozsah je obmedzený 0 až 180 stupňov. Riešenie problému detekcie hrán v danom bloku je analytické, čo znamená, že proces detekcie hrán môže byť vykonaný veľmi rýchlo pre veľké databázové aplikácie bez nutnosti použitia špeciálneho hardvéru. Na Obr. 1 (G, H, I, J, K, L) sú uvedené hrany, ktoré neboli spracované v [1]. Pri odvodení vzťahov vo výpočtoch sme čerpali z literatúry [7].

Hranu v každom bloku určíme pomocou metódy na detekciu hrán [1]. X-hmotnostný moment $M_x$, y-hmotnostný moment $M_y$ a stredový hmotnostný moment $M_0$ sú definované v rámci kružnice *C* vpísanej do bloku *B* takto

$$M_x = \iint_C x\, f(x,y)\, dydx \quad (2)$$

$$M_y = \iint_C y\, f(x,y)\, dydx \quad (3)$$

$$M_0 = \iint_C f(x,y)\, dydx \quad (4)$$

kde *f(x,y)* je šedá úroveň normy vektora bodu *(x,y)* pre zobrazenie šedej snímky z farebnej snímky. Norma vektora bodu *(x,y)* z farebnej snímky sa vypočíta ako

$$f(x,y) = \sqrt{R(x,y)^2+G(x,y)^2+B(x,y)^2} \quad (5)$$

kde *(R(x,y), G(x,y), B(x,y))* označuje hodnoty farieb *(R, G, B)* bodu *(x,y)* farebnej snímky. Hodnotu $\theta$ môžeme ľahko získať z hodnôt *Mx* a *My*. Nech $(\bar{x}, \bar{y})$ sú súradnice ťažiska šedých hodnôt vo vnútri kružnice *C*. Potom

$$(\bar{x}, \bar{y}) = \left(\frac{M_x}{M_0}, \frac{M_y}{M_0}\right) \quad (6)$$

a uhol $\theta$ určujúci orientáciu hrany vypočítame takto

$$\theta = tan^{-1}\left(\frac{\bar{y}}{\bar{x}}\right) = tan^{-1}\left(\frac{M_y}{M_x}\right) \quad (7)$$

Hodnoty *h1*, *h2*, a *l* a typ hrany môže rozhodnúť o zachovaní prvých troch momentov vstupného bloku nasledujúcim spôsobom

$$m_k = p_1 h_1^k + p_2 h_2^k \quad (8)$$

kde $p_1 = A_1/A$, $A_1$ je plocha s hodnotou $h_1$ bloku *B*, *A* je celková plocha bloku *B* a je rovná 4, $p_2 = 1 - p_1$, $B=\{(x,y) : |x|\leq 1, |y|\leq 1\}$, a $m_k$ je k-tý šedý (kvadrantový) moment pôvodného bloku *B* a je vypočítaný takto

$$m_k = \frac{1}{A'} \iint_B g^k(x,y)\, dy\, dx. \quad (9)$$

Za predpokladu, že *g(x,y)* má konštantnú hodnotu v jednotlivých zoznamoch, integrál v rovnici (9) sa stáva váženým súčtom hodnôt bodov v bloku *B* a môže byť písaný ako

$$m_k = \sum_x \sum_y w_{xy}\, g^k(x,y), \quad k = 1,2,3 \quad (10)$$

kde $w_{xy}$ je váhový koeficient spojený s *(x,y)* (ak veľkosť bloku je 4 × 4, potom $w_{xy} = 1/16$).

Ak vypočítame hodnoty *h1* a *h2* dostaneme šedý blok snímky, ktorý je zložený iba z hodnôt *h1* alebo *h2* prípadne ich kombinácie. Na Obr. 2 môžeme vidieť originálnu snímku (a) a snímku zloženú zo šedých blokov (b).


a) b)

**Obr. 2.** Prvá snímka videosekvencie a) originálna snímka, b) snímka zložená zo šedých blokov.

Vzdialenosť hrany môžeme potom vypočítať podľa vzťahu

$$l = (1 - 2p_2)cos\theta \quad (11)$$

Aj keď ťažisko bloku B na Obr. 3 sú umiestnené v prvom kvadrante so súradnicami $(\bar{x}, \bar{y})$, môže nastať aj situácia keď sa ťažisko nachádza v ostatných kvadrantoch. Pomocou hodnôt $M_x$ a $M_y$ jednoducho nájdeme orientáciu hrany

$M_x > 0 \wedge M_y > 0 \Rightarrow \bar{\theta} = \theta$ I. kvadrant (12)

$M_x < 0 \wedge M_y > 0 \Rightarrow \bar{\theta} = \pi - \theta$ II. kvadrant (13)

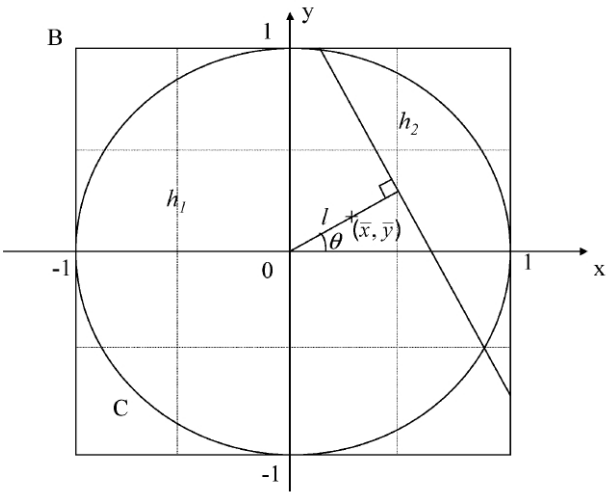
$M_x < 0 \wedge M_y < 0 \Rightarrow \bar{\theta} = \pi + \theta$ III. kvadrant (14)

$M_x > 0 \wedge M_y < 0 \Rightarrow \bar{\theta} = 2\pi - \theta$ IV. kvadrant (15)

Zladenie dvoch hranových blokov je pre nás nevyhnutné pre určenie pohybového vektora. Zladenie hrán si môžeme predstaviť ako prekrytie hrán.

V našom prípade, existujú dva prípady zladenia dvoch hranových blokov. Prvý prípad nastane, ak máme rovnakú orientáciu hrán dvoch blokov *(B, B′)*, čo znamená, že v prípade prekrytia hrany *B* s hranou *B′*, sa hrany presne zhodujú, ako je znázornené na Obr. 4. (a). Druhý prípad nastane ak máme rozdielnu orientáciu hrán. V tomto prípade najprv hranu bloku *B* preložíme a potom otočíme o uhol, ktorý je rovný rozdielu medzi vzdialenosťami hrán *B* a *B′*, aby sa prekrývali obidve hrany, viď Obr. 4 (b).

Dané dva hranové bloky *B* a *B′* sa vyznačujú orientáciou hrany a vzdialenosťou hrany *(θ, l)* a *(θ′, l′)*. Proces zladenia hranových blokov sa skladá z dvoch krokov. Prvý krok je posun a druhý krok je rotácia, viď Obr. 5.




**Obr. 3.** Blok *B* s hranou o veľkosti 4 × 4. Kruh *C* je vpísaný do bloku *B* a *(x̄, ȳ)* sú súradnice ťažiska hodnôt šedej farby vo vnútri kruhu *C*.

Najprv preložíme stred *B′* do stredu *B*, čo má za následok posunutie:

$$\overrightarrow{u_1} = (x_c - x'_c,\ y_c - y'_c) \qquad (16)$$

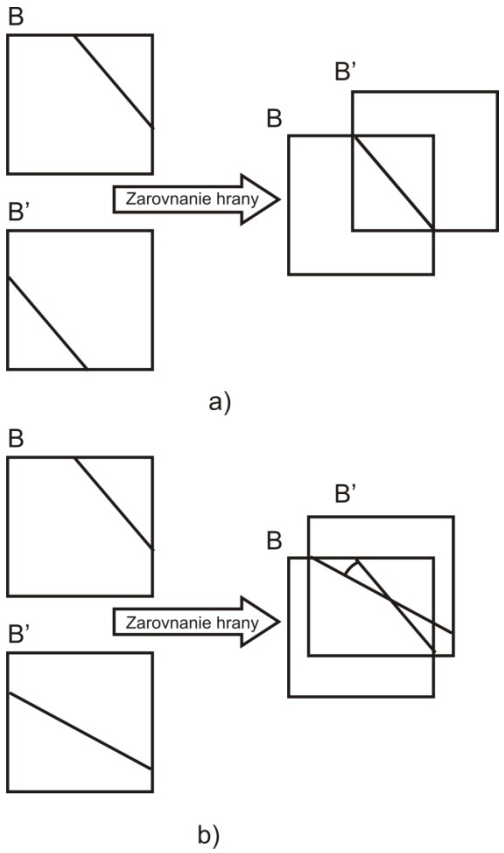
kde $(x_c, y_c)$ a $(x'_c, y'_c)$ sú stredové súradnice hranového bloku *B* a *B′*. Potom ešte preložíme blok podľa rovnice:

$$\overrightarrow{u_2} = (Nd_1 cos\theta, Nd_1 sin\theta) \qquad (17)$$

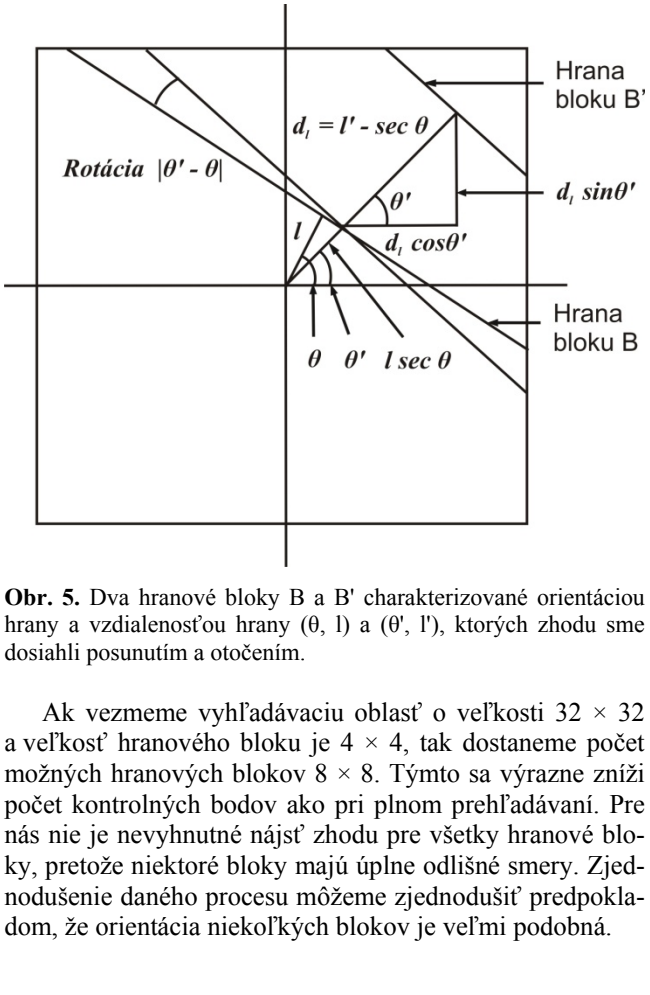
kde $d_l = |\ l' - l\ sec\theta\ |$ je vzdialenosť stredu preloženia hrán $h_B$ a $h_{B'}$, tak aby sa stredy *B* a *B′* zhodovali pričom *N* je veľkosť bloku. Kombináciou (16) a (17), dostaneme konečný pohybový vektor

$$\vec{u} = \overrightarrow{u_1} + \overrightarrow{u_2} \qquad (18)$$

Hranové bloky sú dôležité pre vyhľadávanie pohybových vektorov snímky. Preto sme hľadali hranové bloky pre aktuálnu snímku v postupnosti zľava doprava a zhora dolu. Nech *B* je hranový blok v aktuálnej snímke a *B′* je odpovedajúci hranový blok v referenčnej snímke. Našou úlohou bude nájsť zladenie medzi týmito blokmi. SAD hodnota nám pomôže posúdiť mieru zhody medzi *B* a *B′*.



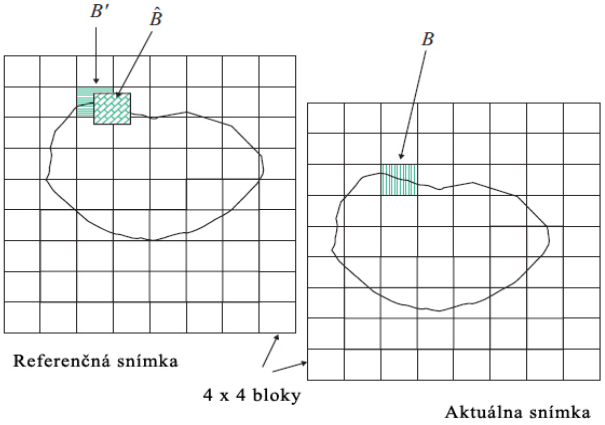

**Obr. 4.** Proces zladenia dvoch hranových blokov: a) rovnaká orientácia hrán dvoch blokov *B* a *B′*, b) rozdielna orientácia hrán dvoch blokov *B* a *B′*.




**Obr. 5.** Dva hranové bloky B a B' charakterizované orientáciou hrany a vzdialenosťou hrany (θ, l) a (θ', l'), ktorých zhodu sme dosiahli posunutím a otočením.

Ak vezmeme vyhľadávaciu oblasť o veľkosti 32 × 32 a veľkosť hranového bloku je 4 × 4, tak dostaneme počet možných hranových blokov 8 × 8. Týmto sa výrazne zníži počet kontrolných bodov ako pri plnom prehľadávaní. Pre nás nie je nevyhnutné nájsť zhodu pre všetky hranové bloky, pretože niektoré bloky majú úplne odlišné smery. Zjednodušenie daného procesu môžeme zjednodušiť predpokladom, že orientácia niekoľkých blokov je veľmi podobná.

Definujme hranový blok $B = (h_1, h_2, \theta, l)$ so stredom v $(x_B, y_B)$ v aktuálnej snímke a hranový blok $\hat{B} = (\widehat{h_1}, \widehat{h_2}, \hat{\theta}, \hat{l})$ so stredom v $(x_{\hat{B}}, y_{\hat{B}})$, ktorý je najlepším kadidátom na zhodu s blokom $B$ v referenčnej snímke. Predpokladajme, že blok $B'$ je jedným z pravidelne rozdelených blokov v referenčnej snímke v blízkosti bloku $\hat{B}$, $B' = (h_1', h_2', l', \theta')$ odkazuje na parametre bloku $B$, viď Obr. 5. Je zrejmé, že orientácia bloku $\hat{B}$ sa bude rovnať orientácii bloku $B'$, $\hat{\theta} = \theta'$.




**Obr. 6.** Hranice zhody $\hat{B}$ pre daný hranový blok $B$ v aktuálnej snímke sa nemusí časovo zhodovať s daným kódovaným blokom v referenčnej snímke.

Ďalej by pri možnej zhode malo platiť, že, $\hat{l} = l'$. Potrebujeme zistiť kde sa nachádza stred $(x_{\hat{B}}, y_{\hat{B}})$ hranového bloku $\hat{B}$. Pomocou nasledujúcej sústavy rovníc určíme umiestnenie hranového bloku $\hat{B}$.

$$l' - \frac{x_{\hat{B}} - x_{B'}}{N} cos\theta' - \frac{y_{\hat{B}} - y_{B'}}{N} cos\theta' = l \qquad (19)$$

$$y_{\hat{B}} - y_{B'} = tan\theta' \times (x_{\hat{B}} - x_{B'}) \qquad (20)$$

$$\begin{bmatrix} x_{\hat{B}} \\ y_{\hat{B}} \end{bmatrix} = \begin{bmatrix} x_{B'} + N(l' - l)cos\theta' \\ y_{B'} + N(l' - l)cos\theta' \end{bmatrix} \qquad (21)$$

Potom pomocou ďalšej rovnice dostaneme vektor pohybu

$$\vec{u} = (x_{\hat{B}} - x_B,\ y_{\hat{B}} - y_B) \qquad (22)$$

ak $\hat{B}$ je predikovaný blok bloku $B$.

Najprv musíme definovať pravidlo, podľa ktorého budeme posudzovať typ bloku. Máme dva typy blokov: blok hranový a blok homogénny.

Ak blok B je definovaný ako hranový blok, tak platí

$$|h_2 - h_1| > \tau \qquad (23)$$

kde $|\ h_2 - h_1\ |$ je kontrast bloku $B$ a $\tau$ je preddefinovaný prah. Je samozrejmé, že ak použijeme malú hodnotu $\tau$ tak budem mať veľké množstvo hranových blokov. My sme si stanovili prah $\tau$ ako polovicu priemeru kontrastu bloku z aktuálnej snímky, aby sme nedostali príliš veľa hranových blokov. Ale aj tak máme veľa homogénnych blokov, ktorých vektor pohybu nie je možné získať použitím procesu zladenia hrán. U každého homogénneho bloku $U$, môžeme vektor pohybu jednoducho predvídať podľa pohybových vektorov hranových blokov v danom makrobloku takto

$$\vec{u}_U = \sum_{B \in E} w_B \overrightarrow{u_B}, \sum_{B \in E} w_B = 1 \qquad (24)$$

kde $E$ je množina hranových blokov v makrobloku obsahujúceho $U$, $\vec{u}_B$ je pohybový vektor hranového bloku $B$, a $w_B$ je váha bloku $B$. Hranový blok, ktorý nie je ďaleko od homogénneho bloku $U$ má malý vplyv na vektor pohybu bloku $U$. To znamená, že hodnotu $w_B$ vypočítame podľa

$$w_B = a_B / \sum_{B' \in E} a_{B'}, a_B = 1 - d(B,U)/d_{max} \qquad (25)$$

kde $d(B,U)$ je Euklidovská vzdialenosť medzi $B$ a $U$, pokiaľ ide o stredové súradnice a $d_{max}$ je maximálna vzdialenosť $U$ všetkých hranových blokov vo $E$. Podobne môžeme gradientom interpolovať orientáciu homogénneho bloku $U$ podľa $\theta_U = \sum_{B \in E} w_B \theta_B$, ktorý sa ďalej používa pre jemné doladenie vektora pohybu.Výhody navrhovanej hranovej blokovej metódy sú tri:

(1) rýchlo stanovuje pohybové vektory s menšími SAD hodnotami bloku v aktuálnej snímke,

(2) pohybové vektory sa získavajú sub-pixlovou presnosťou,

(3) okrajové informácie, ktoré sú vizuálne dôležité pre ľudské vnímanie sú zachované počas kompresie videa.

Bez ujmy na všeobecnosti predpokladajme, že máme ideálny objekt jednej farby pohybujúci sa na jednotnom pozadí. Vzhľadom na blok $B$ na rozhraní objektu v aktuálnej snímke, môže vzor hrany bloku $B$ byť modelovaný ako krok hrany, ktorý triedi hodnoty obrazových bodov z $B$ do dvoch tried, jedna z nich je zastúpená $h_1^B$ a druhá je zastúpená $h_2^B$ pričom ($h_1^B < h_2^B$). Nech blok $\hat{B}$ je zodpovedajúci bloku $B$ vo vzťažnej sústave charakterizovaný dvoma hodnotami reprezentatívnych obrazových bodov $h_1^{\hat{B}}$ a $h_2^{\hat{B}}$ pričom ($h_1^{\hat{B}} < h_2^{\hat{B}}$). Potom máme $h_1^B \approx h_1^{\hat{B}}$ *a* $h_2^B \approx h_2^{\hat{B}}$. Ako je ukázané na Obr. 7 (a), hodnotu SAD pre $B$ a $\hat{B}$ ak sú ich hrany správne vyrovnané, je možné vypočítať pomocou vzťahu

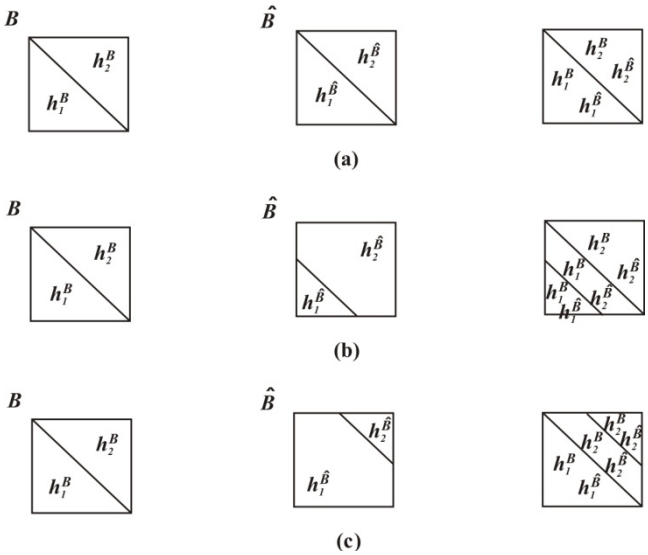
$$SAD_1 = n_1|h_1^B - h_1^{\hat{B}}| + n_2|h_2^B - h_2^{\hat{B}}| \qquad (26)$$

kde $n_1$ a $n_2$ sú počty obrazových bodov bloku $B$ zastúpené $h_1^B$ a $h_2^B$. Obr. 7 (b) a 7 (c) ukazujú dva prípady kedy hrany nie sú správne zladené a hodnoty SAD vypočítame podľa vzťahov

$$\begin{aligned} SAD_2 &= n_{11}\left|h_1^B - h_1^{\hat{B}}\right| + n_{12}\left|h_1^B - h_2^{\hat{B}}\right| \\ &\quad + n_2\left|h_2^B - h_2^{\hat{B}}\right|,\ \ n_1 = n_{11} + n_{12} \\ SAD_3 &= n_1\left|h_1^B - h_1^{\hat{B}}\right| + n_{21}\left|h_2^B - h_1^{\hat{B}}\right| \\ &\quad + n_{22}\left|h_2^B - h_2^{\hat{B}}\right|,\qquad n_2 = n_{21} + n_{22} \end{aligned} \qquad (27)$$

Porovnaním (26) a (27), je ľahké dokázať, že hodnota $SAD_1$ je menšia ako $SAD_2$ alebo $SAD_3$. To znamená, že

proces zladenia hrán vytvára pohybové vektory s nižšími hodnotami SAD. V H.264, sú pohybové vektory potrebné na dosiahnutie sub-pixlovej presnosti a to vedie k vysokej výpočtovej zložitosti. Vektory pohybu získané prostredníctvom nami navrhovaného procesu zladenia hrán spĺňajú osobitné požiadavky H.264 bez ďalších náročných výpočtov.




**Obr. 7.** Vyrovnanie hrán na dva ideálne kroky: (a) hrany blokov $B$ a $\hat{B}$ sú zladené správne, (b) a (c) sú dva prípady, kedy nie sú hrany blokov $B$ a $\hat{B}$ zladené správne.

Na Obr. 8 sú uvedené tvary hrán vychádzajúce z pôvodného Obr. 2.

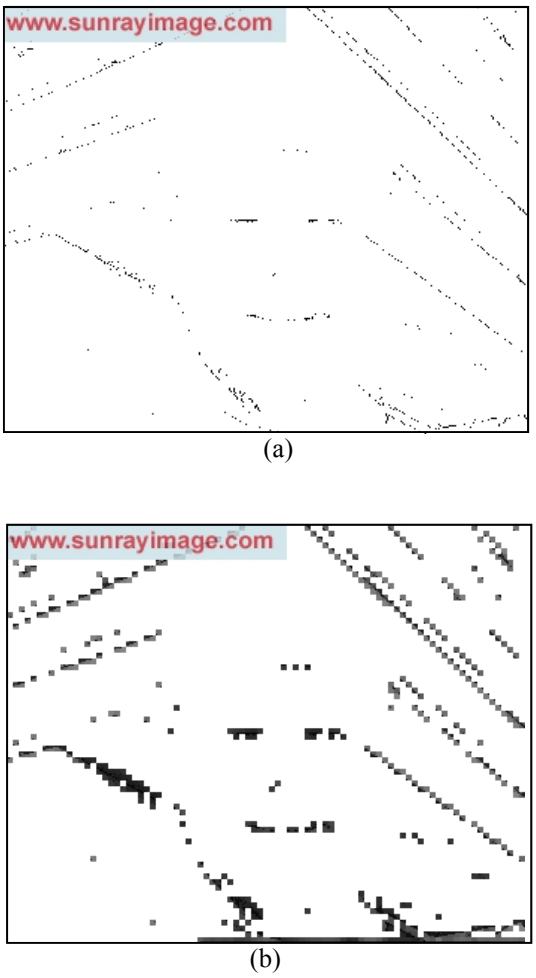
## 3 Algoritmus predikcie pohybových vektorov použitím zladenia hrán

Pomocou hrany bloku v aktuálnej snímke môžeme predikovať pohybový vektor a získať ho pomocou procesu zladenia hrán. Pre lepšie pochopenie sme proces zladenia hrán zhrnuli do nasledujúceho algoritmu.

**Algoritmus zladenia hrán pre odhad pohybu**

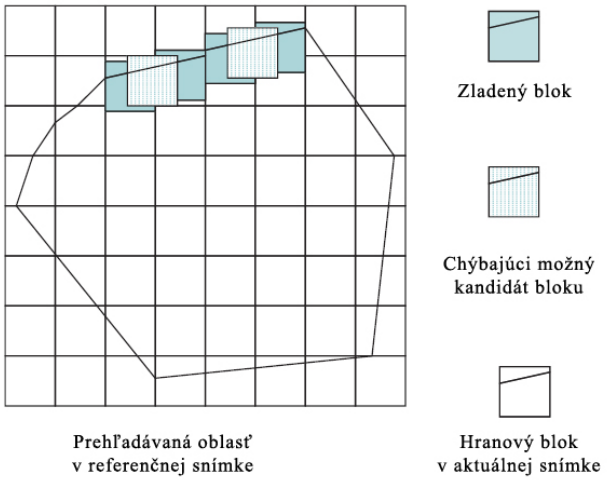
**Vstup:** 4 × 4 blok $B$.
**Výstup:** Pohybový vektor $\overrightarrow{u_B}$ bloku $B$.
**Metóda:**

1. V danom okamihu urobíme detekciu hrany spolu s detekciou orientácie $\theta$ a vzdialenosťou od hrany $l$ v bloku $B$.
2. Ak blok $B$ je hranový blok, potom
   2.1 Inicializovať $SAD_B$ ako veľmi veľké číslo
   2.2 Pre každý hranový blok $B'$ so vzdialenosťou $l'$ a orientáciou $\theta'$ vo vyhľadávanej oblasti bloku $B$ v referenčnej snímke urobíme
      2.2.1 Ak $|\theta - \theta'| < \xi$ //$\xi$ je predefinovaný prah.
      2.2.2 Použijeme (7) pre určenie polohy bloku $\hat{B}$, ktorý je alebo má byť blokom so zhodnou hranou v bloku $B$.
      2.2.3 Vypočítame SAD hodnotu $SAD_{\hat{B}}$ medzi blokom $B$ a $\hat{B}$ použitím (1).
      2.2.4 Ak $(SAD_{\hat{B}} < SAD_B)$ potom $SAD_B = SAD_{\hat{B}}$ a $\vec{u}_B = (x_{\hat{B}} - x_B, y_{\hat{B}} - y_B)$, kde $(x_B, y_B)$ a $(x_{\hat{B}}, y_{\hat{B}})$ sú stredové súradnice bloku $B$ a príslušného bloku $\hat{B}$
3. Ak $B$ je homogénny blok, urobíme interpoláciu $\overrightarrow{u_B}$ na základe pohybových vektorov hranových blokov blízko bloku $B$ v aktuálnej snímke pomocou (24).

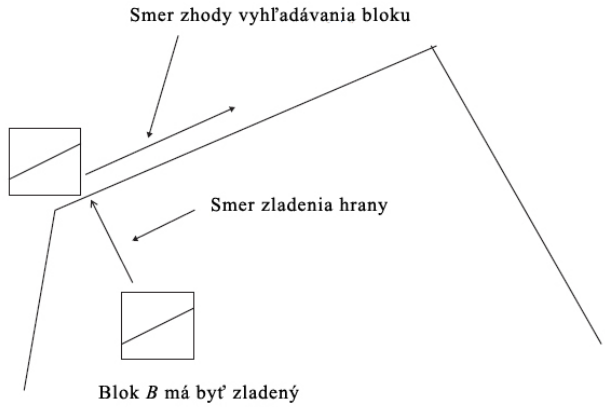



**Obr. 8.** Určenie hrán v pôvodnom Obr. 2. (a) Hrany tvorené pixelmi čiar. (b) Hrany tvorené blokmi pixelov.

V skutočnosti je algoritmus rozdelený do dvoch fáz. V prvej fáze určíme vektory pohybu pre všetky 4 × 4 hranové bloky v aktuálnej snímke. Potom, v druhej fáze sa pohybové vektory zvyšných blokov interpolujú. Tieto pohybové vektory ešte nie sú konečné pohybové vektory a budeme ich ešte ďalej spracovávať, pretože zatiaľ sme zladili iba hranové bloky kolmé na hranu bloku B Obr. 6. Niektorí kandidáti na najlepšiu zhodu by mohli chýbať, ak neuvažujeme zladenie blokov pozdĺž hrany Obr. 9. Preto pridáme proces zladenia hrán, podľa vzoru ortogonálneho vyhľadávania Obr. 10, ktorý doladí pohybový vektor bloku v aktuálnej snímke, aby sme získali lepšiu kvalitu obrazu. V tejto práci navrhujeme nový vyhľadávací model na doladenie pohybového vektora.




**Obr. 9.** Potencionálni kandidáti blokov pre najlepšiu zhodu by mohli chýbať pri tom bloku, ktorého odhad pohybu sme dostali pomocou navrhovanej stratégie zladenia hrany.




**Obr. 10.** Vyhľadávacia stratégia ortogonálneho bloku: blok *B* v aktuálnej snímke, ktorého počiatočný pohybový vektor získame použitím navrhovaného procesu zladenia hrany a potom doladíme pohybový vektor zodpovedajúci blokom v smere hrany bloku *B*.

## 4 Kvalita obrazu

Na porovnanie kvality obrazu budeme používať metódu vyhodnocovania špičkového pomeru signál – šum PSNR (Peak signal-to-noise ratio). Pomer vypočítame pomocou nasledujúceho vzťahu

$$PSNR = 10log_{10}\left\{\frac{(255)^2}{\frac{1}{256\times256}\sum_{i=1}^{256}\sum_{j=i}^{256}(X_{i,j}-\widetilde{X_{i,j}})^2}\right\} \qquad (28)$$

kde $X_{i,j}$ označuje hotnoty obrazových bodov v aktuálnej snímke a $\widetilde{X_{i,j}}$ označuje hodnoty obrazových bodov v predchádzajúcej snímke po vykonaní pohybovej kompenzácie.

## 5 Záver

Nami zvolená metóda urýchli proces vyhľadávania pohybových vektorov pri kódovaní videa v H.264. Pre tento proces vyhľadávania pohybových vektorov bol vytvorený špeciálny program, v ktorom je načítaných 300 snímok o veľkosti 352x288 zo súboru. Potom jednotlivé snímky sú kódované a následne dekódované. V tomto procese použijeme špeciálne upravenú knižnicu H.264, v ktorej bude zahrnutá aj naša metóda vyhľadávania pohybových vektorov pomocou zladenia hrán. Metóda je overovaná v praxi vo videokonferenčnom systéme EVO, pre ktorý danú metódu plánujeme využiť. Na overenie danej metódy sa používa vyhodnocovanie špičkového pomeru signál-šum. Samozrejme, že do úvahy sa berie aj čas, za ktorý prekódujeme 300 snímok a aj kompresiu dát v bitoch, aby sme vedeli posúdiť, kedy je daná metóda výhodnejšia oproti ostatným štandardným metódam.

## Poďakovanie

Tento príspevok vznikol za podpory grantovej agentúry VEGA v rámci grantového projektu 1/0479/12.

## Literatúra

1. S.-C. Cheng: *Content-based image retrieval using moment-preserving edge detection*. Image and Vision Computing, 21(9), 2003, 809-826.
2. C.-H. Chuang, C.-H. Su, S.-C. Cheng: *Fast block motion estimation with edge alignment on H.264 video coding*. Journal of Marine Science and Technology, 18(6), 2010, 883-894.
3. *Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification*, ITU-T Rec. H.264 and ISO/IEC 14 496-10 AVC, Joint Video Team 2003.
4. S. Zhu, K. K. Ma: *A new diamond search algorithm for fast blockmatching motion estimation.* IEEE Transactions on Image Processing, 9(2), 2000, 287-290.
5. C. Zhu, X. Lin, L. P. Chau: *Hexagon-based search pattern for fast block motion estimation*. IEEE Transactions on Circuits and Systems for Video Technology, 12( 5), 2002, 349-355.
6. X. Xu, Y. He: *Improvements on fast motion estimation strategy for H.264/AVC*. IEEE Transactions on Circuits and Systems for Video Technology, 18(3), 2008, 285-293.
7. J. Sekerák: *Kľúčové kompetencie žiaka vo vyučovaní analytickej geometrie na gymnáziách*. In: G – slovenský časopis pre geometriu a grafiku, Bratislava 2008, 5(9), 2008. 23 - 30. ISSN 1336-524X.

# Drsné množiny a formálny kontext

Ľubomír Antoni and Stanislav Krajči

Ústav informatiky PF UPJŠ, Jesenná 5, 040 01 Košice
lubomir.antoni@student.upjs.sk, stanislav.krajci@upjs.sk

**Abstrakt** *Analýza drsných množín pracuje s objektovo-atribútovým modelom, v ktorom modelujeme nerozlíšiteľnosť objektov využitím relácie ekvivalencie. Hlavnou výhodou presúvania postupov a výsledkov z jednej oblasti do inej je to, že v inej oblasti môžu byť tieto výsledky a postupy menej známe a je možné ich vhodne aplikovať a rozšíriť pole aplikácií. Preto sme skúmali, či pri analýze drsných množín môžeme objaviť štruktúry spĺňajúce vlastnosti Galoisovej konexie a operátora uzáveru, ktoré sú dobre známe vo formálnej konceptovej analýze. Ďalej navrhujeme spôsob ohodnotenia prvkov faktorovej množiny z tried ekvivalencí, ktoré sa používajú na aproximáciu drsných množín a analogicky spôsob ohodnotenia extentov konceptov vo formálnej konceptovej analýze cez prierez zväzov získaných z rôznych podmnožín atribútov. Tento synergizmus ilustrujeme na vhodných príkladoch.*

## 1 Úvod

Pojem drsných množín prvýkrát uviedol Pawlak [9,10] v roku 1982. Od tej doby až po súčasnosť vzrastá záujem o túto problematiku a mnohé oblasti úspešne využívajú softvérové systémy založené na aplikáciách drsných množín. V [5] je uvedený prehľad týchto aplikácií, spomenieme napríklad analýzu obrazu v medicínskych aplikáciách, ekonomiku a financie pri hodnotení rizika a firiem, sociálne vedy pri analýze konfliktov. Niektoré vzťahy teórie drsných množín s teóriou topologických priestorov, prípadne s logikou naznačuje Vlach v [11,12,13]. Vlachove výsledky nás inšpirovali k tomu, aby sme rozšírili a upresnili ďalšie súvislosti.

Základy formálnej konceptovej analýzy boli položené v práci Gantera a Willeho [3] na základe teórie úplných zväzov. Jedným z najväčších problémov formálnej konceptovej analýzy je ohodnotenie významnosti konceptu. Objavuje sa niekoľko pokusov. V [6] je prezentovaný modifikovaný Rice-ov a Siff-ov algoritmus, ktorý však slúži na ohodnotenie konceptov fuzzy formálneho konceptu, teda pracuje s viachodnotovým modelom. V [1] pracujeme s fuzzy formálnym kontextom, ale na ohodnotenie konceptov používame dolné, prípadne horné $\alpha$-rezy.

Klimushkin a kol. [4] nadväzuje na Kuznetsov index stability prezentovaný v [8], ktorý ohodnocuje koncepty na základe počtu podmnožín extentov daného konceptu, ktorých uzáver je rovný intentu tohto konceptu. V [4] je tento index stability normalizovaný zavedením pravdepodobnosti, že daná množina objektov môže byť konceptom. Index stability teda ohodnocuje koncepty vertikálne v rámci jedného konceptového zväzu. Prístup, ktorý navrhujeme v našom príspevku je horizontálny, extenty konceptov ohodnocujeme v rámci viacerých konceptových zväzov na rôznych podmnožinách atribútov.

V literatúre nachádzame aj iné prístupy, ktoré sa nezaoberajú ohodnotením konceptov, ale navrhujú metódy redukcie výsledného konceptového zväzu. V [2] je prezentovaná metóda založená na spojení podobných objektov do jedného a následnej redukcii výsledného konceptového zväzu, ale vyžaduje od užívateľa pripísať každému atribútu váhu, ktorá určuje významnosť každého atribútu vo formálnom kontexte.

Tento článok je organizovaný v siedmich kapitolách. Po úvodnej časti zavedieme základné pojmy teórie drsných množín a tieto pojmy ilustrujeme na vlastnom príklade. V tretej kapitole prispievame vlastnými dôkazmi k určeniu vlastností hornej a dolnej aproximácie a objavujeme ich súvislosť s Galoisovou konexiou, dôležitou štruktúrou vo formálnej konceptovej analýze. Vlastnú metódu na hodnotenie faktorov získaných pomocou relácie nerozlíšiteľnosti prezentujeme v štvrtej kapitole. V piatej kapitole uvádzame základné pojmy z formálnej konceptovej analýzy a navrhnutú myšlienku ohodnotenia faktorov zo štvrtej kapitoly aplikujeme na novú metódu ohodnotenia extentov konceptov. Nakoniec uvádzame záverečné poznámky s ohľadom na ďalšiu prácu v danej oblasti.

## 2 Drsné množiny

Uvažujme objektovo-atribútový model $(B, A, J)$, kde $B$ je neprázdna množina objektov, $A$ je neprázdna množina atribútov a $J$ je zobrazenie, pre ktoré platí $\mathrm{Dom}(J) = B \times A$. Na tomto modeli definujeme nerozlíšiteľnosť objektov ako reláciu:

**Definícia 1** *Nech $Y \subseteq A$. Potom $Y$-reláciou nerozlíšiteľnosti objektov nazývame reláciu*

$$R_Y = \{\langle b_1, b_2\rangle \in B \times B : (\forall a \in Y) J(b_1, a) = J(b_2, a)\}. \quad (1)$$

Táto relácia je reflexívna, symetrická, tranzitívna, teda je reláciou ekvivalencie. To nás oprávňuje celú množinu objektov rozdeliť na triedy ekvivalencie tvaru

$$[b]_{R_Y} = \{c \in B : \langle b, c\rangle \in R_Y\}. \quad (2)$$

Množinu všetkých tried ekvivalencií na množine $B$ s reláciou nerozlíšiteľnosti $R_Y$ budeme označovať $B/R_Y$.

Ilustrujme tieto pojmy v nasledujúcom príklade.

*Príklad 1.* O povýšení v zamestnaní sa rozhoduje na základe veku zamestnanca a ocenenia jeho nadriadeným. V našom prípade hodnotí nadriadený zamestnanca v stupnici od 0 – 3, kde 3 predstavuje najväčšiu spokojnosť so zamestnancom. Hodnoty pre siedmich zamestnancov firmy sú uvedené v tabuľke 1.

| zamestnanec | vekový interval v | ocenenie o | povýšenie p |
|---|---|---|---|
| 1 | 31 – 40 | 1 | NIE |
| 2 | 31 – 40 | 3 | ÁNO |
| 3 | 41 – 50 | 2 | NIE |
| 4 | 21 – 30 | 3 | ÁNO |
| 5 | 41 – 50 | 2 | NIE |
| 6 | 31 – 40 | 1 | ÁNO |
| 7 | 21 – 30 | 0 | NIE |

**Tab. 1.** Objektovo-atribútový model povýšenia v zamestnaní.

Množinu objektov tvorí $B = \{1,2,3,4,5,6,7\}$, množinu atribútov tvorí $A = \{\mathrm{v},\mathrm{o},\mathrm{p}\}$. Uvažujme reláciu nerozlíšiteľnosti objektov pre atribút vek:

$$R_{\{\mathrm{v}\}} = \{\langle 1,1\rangle,\langle 1,2\rangle,\langle 1,6\rangle,\langle 2,1\rangle,\langle 2,2\rangle,\langle 2,6\rangle,\langle 6,1\rangle,\langle 6,2\rangle,\langle 6,6\rangle\} \cup \{\langle 4,4\rangle,\langle 4,7\rangle,\langle 7,4\rangle,\langle 7,7\rangle\} \cup \cup\{\langle 3,3\rangle,\langle 3,5\rangle,\langle 5,3\rangle,\langle 5,5\rangle\}.$$

Trieda ekvivalencie, ktorá obsahuje zamestnanca 2 pre $R_{\{\mathrm{v}\}}$ je:

$$[2]_{R_{\{\mathrm{v}\}}} = \{1,2,6\}.$$

Množina všetkých tried ekvivalencií pre $R_{\{\mathrm{v}\}}$ má tvar

$$B/R_{\{\mathrm{v}\}} = \{\{1,2,6\},\{4,7\},\{3,5\}\}.$$

Môžeme uvažovať aj viac atribútov (vek a ocenenie). V takom prípade množina všetkých tried ekvivalencií pre $R_{\{\mathrm{v,o}\}}$ má tvar

$$B/R_{\{\mathrm{v,o}\}} = \{\{1,6\},\{2\},\{3,5\},\{4\},\{7\}\}.$$

Uvedený príklad ukazuje, ako je možné použitím relácie nerozlíšiteľnosti rozdeliť celú množinu objektov na disjunktné množiny tried ekvivalencií. Na každú takúto triedu ekvivalencie sa môžeme pozerať ako na tzv. základnú definovateľnú podmnožinu množiny objektov (základný stavebný blok). Zjavne však vieme nájsť aj takú podmnožinu objektov (napr. $\{1,2,3\}$), ktorá nie je triedou ekvivalencie, a teda v našom zmysle nepatrí medzi základné definovateľné podmnožiny. V takom prípade však vieme takúto podmnožinu aproximovať pomocou základných stavebných blokov zdola a zhora. Preto v ďalšom definujeme pojmy dolnej a hornej aproximácie množiny.

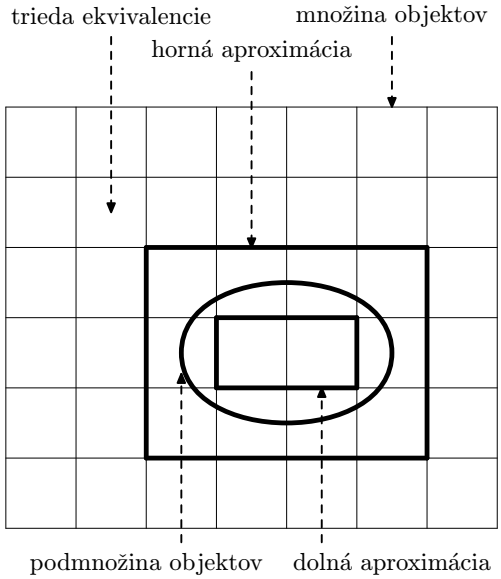
**Definícia 2** *Nech $X \subseteq B$ a $Y \subseteq A$. Dolnou aproximáciou množiny $X$ nazývame zobrazenie $\underline{R_Y}: \mathrm{P}(B) \to \mathrm{P}(B)$*

$$\underline{R_Y}(X) = \{x \in B : [x]_{R_Y} \subseteq X\}. \tag{3}$$

**Definícia 3** *Nech $X \subseteq B$ a $Y \subseteq A$.Hornou aproximáciou množiny $X$ nazývame zobrazenie $\overline{R_Y} : \mathrm{P}(B) \to \mathrm{P}(B)$*

$$\overline{R_Y}(X) = \{x \in B : [x]_{R_Y} \cap X \neq \emptyset\}. \tag{4}$$

Dvojicu $\left(\underline{R_Y}(X), \overline{R_Y}(X)\right)$ nazveme drsnou množinou pre $X \subseteq B$ a $Y \subseteq A$. Definované pojmy ilustrujeme na nasledujúcom obrázku.




**Obr. 1.** Grafické znázornenie hornej a dolnej aproximácie drsnej množiny.

Každú podmnožinu množiny objektov vieme následne aj numericky charakterizovať pomocou koeficientu aproximácie.

**Definícia 4** *Nech $X \subseteq B$ a $Y \subseteq A$. Koeficientom presnosti aproximácie nazývame zobrazenie* $\alpha_Y : \mathrm{P}(B) \to [0,1]$

$$\alpha_Y(X) = \frac{|\underline{R_Y}(X)|}{|\overline{R_Y}(X)|}. \tag{5}$$

*Ak $\alpha_Y(X) = 1$, tak množinu $X$ nazývame základná $R_Y$-definovateľná množina.*

*Príklad 2.* Aproximujme skupinu zamestnancov, ktorí boli povýšení v zamestnaní. To znamená, že z tabuľky 1 vyberieme objekty $X = \{2,4,6\}$. Množina všetkých tried ekvivalencií pre reláciu nerozlíšiteľnosti, ktorá zohľadňuje vek a ocenenie má tvar

$$B/R_{\{\mathrm{v,o}\}} = \{\{1,6\},\{2\},\{3,5\},\{4\},\{7\}\}.$$

Dolnú aproximáciu nájdeme podľa vzťahu (3) tak, že vyberieme všetky triedy ekvivalencie, ktoré sú celé podmnožinou $X$:

$$\underline{R_{\{\mathrm{v,o}\}}}(X) = \{2,4\}.$$

Hornú aproximáciu nájdeme podľa vzťahu (4) tak, že vyberieme všetky triedy ekvivalencie, ktorých prienik s množinou $X$ je neprázdny:

$$\overline{R_{\{\mathrm{v,o}\}}}(X) = \{1,2,4,6\}.$$

Koeficient presnosti aproximácie je potom

$$\alpha_{\{\mathrm{v,o}\}}(X) = \frac{|\underline{R_{\{\mathrm{v,o}\}}}(X)|}{|\overline{R_{\{\mathrm{v,o}\}}}(X)|} = \frac{1}{2}.$$

Získané výsledky je možné interpretovať nasledovne. Vek a ocenenie zamestnancov, ktorí sa nachádzajú v dolnej aproximácii, jednoznačne predurčuje ich povýšenie v zamestnaní. U zamestnancov, ktorí sa nachádzajú v hornej aproximácii, ale do dolnej aproximácie nepatria, nevieme na základe ich veku a ocenenia jednoznačne predikovať, či dosiahnu povýšenie. Vek a ocenenie zamestnancov, ktorí sa nenachádzajú v hornej aproximácii, jednoznačne predurčuje to, že povýšenie nenastane. Z tabuľky 1 môžeme napríklad vidieť, že vek 31 – 40 a ocenenie hodnotou 1 môže, ale nemusí viesť k povýšeniu. Zamestnanci s týmto vekom a ocenením sa síce objavili v hornej aproximácii, no v dolnej chýbajú.

## 3 Aproximácie ako Galoisova konexia

V druhej kapitole sme zaviedli operátory horného a dolného ohraničenia množín. Našou úlohou bude ukázať niektoré vlastnosti horných a dolných aproximácií, ktorými potvrdíme prítomnosť Galoisovej konexie v súvislosti s drsnými množinami. Pripájame vlastné dôkazy týchto vlastností a zdôvodňujeme získané výsledky.

Najskôr ukážeme prirodzenú vlastnosť umiestnenia množiny medzi jej dolnou a hornou aproximáciou a následne distributívnosť hornej aproximácie nad zjednotením a dolnej aproximácie nad prienikom.

**Lema 1** *Nech $X, X_1, X_2 \subseteq B$ a $Y \subseteq A$. Potom platí:*

*a)* $\underline{R_Y}(X) \subseteq X \subseteq \overline{R_Y}(X)$,
*b)* $\overline{R_Y}(X_1 \cup X_2) = \overline{R_Y}(X_1) \cup \overline{R_Y}(X_2)$,
*c)* $\underline{R_Y}(X_1 \cap X_2) = \underline{R_Y}(X_1) \cap \underline{R_Y}(X_2)$.

*Dôkaz.*

a) $x \in \underline{R_Y}(X)$,
akk[1] $[x]_{R_Y} \subseteq X$ podľa definície 2,
ztv.[2] $x \in X$ lebo $x \in [x]_{R_Y}$,
ztv. $[x]_{R_Y} \cap X \neq \emptyset$ lebo $x \in [x]_{R_Y}$,
akk $x \in \overline{R_Y}(X)$ podľa definície 3.

b) $x \in \overline{R_Y}(X_1 \cup X_2)$,
akk $[x]_{R_Y} \cap (X_1 \cup X_2) \neq \emptyset$,
akk $([x]_{R_Y} \cap X_1) \cup ([x]_{R_Y} \cap X_2) \neq \emptyset$,
akk $([x]_{R_Y} \cap X_1 \neq \emptyset) \vee ([x]_{R_Y} \cap X_2 \neq \emptyset)$,
akk $x \in \overline{R_Y}(X_1) \vee x \in \overline{R_Y}(X_2)$,
akk $x \in \overline{R_Y}(X_1) \cup \overline{R_Y}(X_2)$.

c) $x \in \underline{R_Y}(X_1 \cap X_2)$,
akk $[x]_{R_Y} \subseteq (X_1 \cap X_2)$,
akk $[x]_{R_Y} \subseteq X_1 \wedge [x]_{R_Y} \subseteq X_2$,
akk $x \in \underline{R_Y}(X_1) \wedge x \in \underline{R_Y}(X_2)$,
akk $x \in \underline{R_Y}(X_1) \cap \underline{R_Y}(X_2)$.□

Je potrebné si uvedomiť, že distributívnosť hornej aproximácie nad prienikom neplatí. Pre reláciu nerozlíšiteľnosti $R_{\{\mathrm{v}\}}$ z príkladu 1 a množiny $X_1 = \{1,2,4,6\}$, $X_2 = \{1,2,6,7\}$ dostávame

$$\overline{R_{\{\mathrm{v}\}}}(X_1 \cap X_2) = \{1,2,6\},$$
$$\overline{R_{\{\mathrm{v}\}}}(X_1) \cap \overline{R_{\{\mathrm{v}\}}}(X_2) = \{1,2,4,6,7\}\ .$$

Takisto dolná aproximácia nad zjednotením neplatí. Stačí si zobrať tú istú reláciu nerozlíšiteľnosti a množiny $X_1 = \{1,2,4,6\}$ a $X_2 = \{1,2,6,7\}$. Pre tieto množiny

$$\underline{R_{\{\mathrm{v}\}}}(X_1 \cup X_2) = \{1,2,4,6,7\},$$
$$\underline{R_{\{\mathrm{v}\}}}(X_1) \cup \underline{R_{\{\mathrm{v}\}}}(X_2) = \{1,2,6\}\ .$$

Zaoberajme sa ďalej monotónnosťou a kompozíciou dolnej a hornej aproximácie.

[1] práve vtedy ak (skratka pre ekvivalenciu medzi tvrdeniami, analógia anglického iff)
[2] z toho vyplýva (skratka pre implikáciu medzi tvrdeniami)

**Lema 2** *Nech $X, X_1, X_2 \subseteq B$ a $Y \subseteq A$.*

*a) Ak $X_1 \subseteq X_2$, tak $\underline{\overline{R_Y}}(X_1) \subseteq \underline{\overline{R_Y}}(X_2)$.*
*b) Ak $X_1 \subseteq X_2$, tak $\overline{\underline{R_Y}}(X_1) \subseteq \overline{\underline{R_Y}}(X_2)$.*
*c) $\underline{R_Y}(\overline{R_Y}(X)) = \overline{R_Y}(X)$.*
*d) $\overline{\underline{R_Y}}(\underline{R_Y}(X)) = \underline{R_Y}(X)$.*
*e) $X \subseteq \underline{R_Y}(\overline{R_Y}(X))$.*
*f) $\overline{R_Y}(\underline{R_Y}(X)) \subseteq X$.*
*g) $\overline{R_Y}(X) = B \setminus \underline{R_Y}(B \setminus X)$.*
*h) $\underline{R_Y}(X) = B \setminus \overline{R_Y}(B \setminus X)$.*

*Dôkaz.*

a) $X_1 \subseteq X_2$,
akk $X_1 \cap X_2 = X_1$,
ztv. $\underline{R_Y}(X_1 \cap X_2) = \underline{R_Y}(X_1)$,
akk $\underline{R_Y}(X_1) \cap \underline{R_Y}(X_2) = \underline{R_Y}(X_1)$
podľa časti c) lemy 1,
ztv. $\underline{R_Y}(X_1) \subseteq \underline{R_Y}(X_2)$.

b) $X_1 \subseteq X_2$,
akk $X_1 \cup X_2 = X_2$,
ztv. $\overline{R_Y}(X_1 \cup X_2) = \overline{R_Y}(X_2)$,
akk $\overline{R_Y}(X_1) \cup \overline{R_Y}(X_2) = \overline{R_Y}(X_2)$
podľa časti b) lemy 1,
ztv. $\overline{R_Y}(X_1) \subseteq \overline{R_Y}(X_2)$.

c) $\subseteq$ Podľa časti a) lemy 1.
$\supseteq$ $x \in \overline{R_Y}(X)$,
akk pre všetky $z \in [x]_{R_Y}$ platí
$[z]_{R_Y} \cap X \neq \emptyset$ (lebo $[z]_{R_Y} = [x]_{R_Y}$),
akk pre všetky $z \in [x]_{R_Y}$ platí $z \in \overline{R_Y}(X)$,
ztv. $[x]_{R_Y} \subseteq \overline{R_Y}(X)$,
akk $x \in \underline{R_Y}(\overline{R_Y}(X))$.

d) $\subseteq$ $x \in \overline{R_Y}(\underline{R_Y}(X))$,
akk $[x]_{R_Y} \cap \underline{R_Y}(X) \neq \emptyset$,
ztv. existuje $z \in [x]_{R_Y}$, že $z \in \underline{R_Y}(X)$,
ztv. $[z]_{R_Y} \subseteq X$,
akk $[x]_{R_Y} \subseteq X$ (lebo $[z]_{R_Y} = [x]_{R_Y}$),
akk $x \in \underline{R_Y}(X)$.
$\supseteq$ Podľa časti a) lemy 1.

e) Z $X \subseteq \overline{R_Y}(X)$ podľa časti a) lemy 1
a z $\underline{R_Y}(\overline{R_Y}(X)) = \overline{R_Y}(X)$ podľa časti c) dostávame tvrdenie.

f) Z $\overline{R_Y}(\underline{R_Y}(X)) = \underline{R_Y}(X)$ podľa časti d)
a z $\underline{R_Y}(X) \subseteq X$ podľa časti a) lemy 1 dostávame tvrdenie.

g) $x \in B \setminus \underline{R_Y}(B \setminus X)$,
akk $x \in B \wedge \neg([x]_{R_Y} \subseteq B \setminus X)$,
akk $x \in B \wedge \neg([x]_{R_Y} \cap X = \emptyset)$,
akk $x \in B \wedge [x]_{R_Y} \cap X \neq \emptyset$,
akk $x \in \overline{R_Y}(X)$.

h) $x \in B \setminus \overline{R_Y}(B \setminus X)$,
akk $x \in B \wedge \neg([x]_{R_Y} \cap B \setminus X \neq \emptyset)$,
akk $x \in B \wedge [x]_{R_Y} \cap B \setminus X = \emptyset$,
akk $x \in B \wedge [x]_{R_Y} \subseteq X$,
akk $x \in \underline{R_Y}(X)$. □

Zadefinujeme pojem antitónnej a izotónnej Galoisovej konexie.

**Definícia 5** *Izotónnou Galoisovou konexiou medzi množinami $K$ a $L$ nazývame dvojicu zobrazení $(f, g)$, ak pre každú $I \subseteq K$ a $J \subseteq L$ platí*

$$I \subseteq g(J) \text{ akk } f(I) \subseteq J. \tag{6}$$

*Antitónnou Galoisovou konexiou medzi množinami $K$ a $L$ nazývame dvojicu zobrazení $(f, g)$, ak pre každú $I \subseteq K$ a $J \subseteq L$ platí*

$$I \subseteq g(J) \text{ akk } f(I) \supseteq J. \tag{7}$$

Nasledujúca veta už hovorí o tom, že dvojica zobrazení $\left(\overline{R_Y}(X), \underline{R_Y}(X)\right)$ tvorí izotónnu Galoisovu konexiu medzi tou istou množinou objektov $B$.

**Veta 1** *Nech $X_1, X_2 \subseteq B$ a $Y \subseteq A$. Potom*

$$X_1 \subseteq \underline{R_Y}(X_2) \text{ akk } \overline{R_Y}(X_1) \subseteq X_2.$$

*Dôkaz.*

$\subseteq$: $X_1 \subseteq \underline{R_Y}(X_2)$,
ztv. $\overline{\underline{R_Y}}(X_1) \subseteq \overline{R_Y}(\underline{R_Y}(X_2))$
podľa časti b) lemy 2,
akk $\overline{R_Y}(X_1) \subseteq \underline{R_Y}(X_2)$
podľa časti d) lemy 2,
ztv. $\overline{R_Y}(X_1) \subseteq X_2$
podľa časti a) lemy 1.

$\supseteq$: $\overline{R_Y}(X_1) \subseteq X_2$,
ztv. $\underline{R_Y}(\overline{R_Y}(X_1)) \subseteq \underline{R_Y}(X_2)$
podľa časti a) lemy 2,
akk $\overline{R_Y}(X_1) \subseteq \underline{R_Y}(X_2)$
podľa časti c) lemy 2,
ztv. $X_1 \subseteq \underline{R_Y}(X_2)$
podľa časti a) lemy 1. □

Ďalej definujme zobrazenie $\mathrm{cl}_{R_Y} : \mathrm{P}(B) \to \mathrm{P}(B)$ ako zloženie zobrazení $\overline{R_Y}$ a $\underline{R_Y}$. Ukážeme, že takto získané zobrazenie spĺňa tri podmienky pre operátor uzáveru na množine objektov $B$.

**Lema 3** *Nech $X, X_1, X_2 \subseteq B$, $Y \subseteq A$ a nech $\mathrm{cl}_{R_Y}(X) = \underline{R_Y}(\overline{R_Y}(X))$. Potom platia nasledujúce podmienky:*

*a) $X \subseteq \mathrm{cl}_{R_Y}(X)$,*
*b) ak $X_1 \subseteq X_2$, tak $\mathrm{cl}_{R_Y}(X_1) \subseteq \mathrm{cl}_{R_Y}(X_2)$,*
*c) $\mathrm{cl}_{R_Y}(X) = \mathrm{cl}_{R_Y}(\mathrm{cl}_{R_Y}(X))$.*

*Dôkaz.*

a) Je to časť e) lemy 2.

b) $X_1 \subseteq X_2$,
ztv. $\overline{R_Y}(X_1) \subseteq \overline{R_Y}(X_2)$
podľa časti b) lemy 2,
ztv. $\underline{R_Y}(\overline{R_Y}(X_1)) \subseteq \underline{R_Y}(\overline{R_Y}(X_2)$
podľa časti a) lemy 2.

c) $\underline{R_Y}(\overline{R_Y}(\underline{R_Y}(\overline{R_Y}(X)))) = \underline{R_Y}(\overline{R_Y}(\underline{R_Y}(X))) =$
$= \underline{R_Y}(\overline{R_Y}(X))$
podľa častí c),d) lemy 2. □

Ak by sme poradie zloženia zobrazení $\overline{R_Y}$ a $\underline{R_Y}$ vymenili, tak výsledné zobrazenie je operátor vnútra na množine objektov $B$.

## 4 Ohodnotenie faktorov

Doteraz sme sa zaoberali spôsobom, ako modelovať reláciu nerozlíšiteľnosti, ako vytvoriť systém tried ekvivalencií, ako pracovať s horným a dolným ohraničením a aké majú tieto ohraničenia vlastnosti. V tejto časti navrhujeme a prispievame vlastnou metódou na ohodnotenie faktorov faktorovej množiny, ktorú definujeme pomocou relácie nerozlíšiteľnosti z kapitoly 2 nasledovne:

**Definícia 6** *Nech $X \subseteq B$ a $Y \subseteq A$. Potom definujeme faktorovú množinu $\Sigma/R_Y$ takto:*

$$X \subseteq \Sigma/R_Y \text{ akk existuje indexová množina } I$$
$$a\ \{x_i : i \in I\} \subseteq B \text{ také, že } X = \bigcup_{i\in I}[x_i]_{R_Y}.$$

To znamená, že faktorovú množinu vytvoríme tak, že množinu $B/R_Y$ uzavrieme na zjednotenie. Keďže je zrejmé, že faktorová množina $\Sigma/R_Y$ je uzavretá aj na komplement, tak je to Booleova algebra. V nasledujúcom príklade ilustrujeme vytvorenie faktorovej množiny na základe jednej možnej konkrétnej relácie nerozlíšiteľnosti pre sedem objektov.

*Príklad 3.* Pre množinu siedmich objektov z príkladu 1 a pre reláciu nerozlíšiteľnosti

$$B/R_{\{\mathrm{v}\}} = \{\{1,2,6\},\{4,7\},\{3,5\}\}$$

má faktorová množina tvar

$$\Sigma/R_{\{\mathrm{v}\}} = \{\emptyset,\{1,2,6\},\{4,7\},\{3,5\},\{1,2,4,6,7\},$$
$$\{1,2,3,5,6\},\{3,4,5,7\},\{1,2,3,4,5,6,7\}\}.$$

Vytvorením faktorovej množiny sme získali všetky podmnožiny objektov (faktory), ktoré sa použitím hornej alebo dolnej aproximácie nezmenia. Sú to akési základné jednotky, ktorých prvky sú v istom zmysle medzi sebou podobné. Treba si však uvedomiť, že pre každú podmnožinu atribútov dostávame inú faktorovú množinu. Ktoré faktory sú najdôležitejšie, sa snažíme zistiť ohodnotením faktorov a ich výskytu v rôznych faktorových množinách.

**Definícia 7** *Nech $X \subseteq B$ a $Y \subseteq A$. Definujme zobrazenie* $\mathrm{ND} : \mathrm{P}(B) \to \mathrm{P}(\mathrm{P}(A))$ *nasledovne:*

$$\mathrm{ND}(X) = \{Y \subseteq A : X \in \Sigma/R_Y\} \tag{8}$$

To znamená, že každej podmnožine objektov priradíme všetky podmnožiny atribútov, ktorých faktorová množina túto podmnožinu objektov obsahuje, a teda nerozlišuje jej prvky. Nakoniec vypočítame ohodnotenie pre každú podmnožinu a definujeme zobrazenie $\mathrm{FE} : \mathrm{P}(B) \to [0,1]$ nasledovne:

**Definícia 8** *Nech $X \subseteq B$, $n = |A|$. Ohodnotením faktora $X$ nazývame hodnotu*

$$\mathrm{FE}(X) = \frac{|\mathrm{ND}(X)|}{2^n} \tag{9}$$

Inými slovami určíme, v koľkých faktorových množinách z celkového počtu $2^n$ faktorových množín sa vyskytla konkrétna podmnožina objektov.

*Príklad 4.* Uvažujme objektovo-atribútový model povýšenia v zamestnaní z príkladu 1. Potom zo vzťahu (8) dostávame

$$\mathrm{ND}(\{2,4,6\}) = \{\{\mathrm{p}\},\{\mathrm{v},\mathrm{p}\},\{\mathrm{o},\mathrm{p}\},\{\mathrm{v},\mathrm{o},\mathrm{p}\}\}$$

a ohodnotenie tejto podmnožiny objektov je zo vzťahu (9) rovné hodnote:

$$\mathrm{FE}(\{2,4,6\}) = \frac{|\mathrm{ND}(\{2,4,6\})|}{2^3} = \frac{1}{2}.$$

V tabuľke 2 sú uvedené niektoré faktory s najvyššími hodnotami.

| $X$ | $\mathrm{FE}(X)$ |
|---|---|
| $\{3,5\}$ | $\frac{3}{4}$ |
| $\{7\}$<br>$\{3,5,7\}$<br>$\{1,2,4,6\}$<br>$\{1,2,4,6,7\}$<br>$\{1,2,3,4,5,6\}$ | $\frac{5}{8}$ |
| $\{1,6\}$ | $\frac{1}{2}$ |
| ... | ... |

**Tab. 2.** Ohodnotenie faktorov pre zamestnancov.

Takto usporiadané podmnožiny objektov na základe ohodnotenia faktorov môžeme interpretovať ako najpodobnejšie skupiny zamestnancov. Prázdna množina a množina všetkých objektov sa vyskytuje v každej faktorovej množine, keďže tá je uzavretá na prienik a zjednotenie a ich hodnotenie bude rovné jednej. Tieto množiny však pre nás nie sú z hľadiska uvažovania skupín zamestnancov významné.

## 5 Formálna konceptová analýza

V tejto časti pripomenieme pojmy formálneho kontextu, formálneho konceptu, konceptového zväzu a ukážeme vlastnosti duálne adjungovaného páru zobrazení.

**Definícia 9** *Nech $B, A$ sú neprázdne množiny a nech $I \subseteq B \times A$. Potom trojicu $(B, A, I)$ nazveme formálny kontext (prípadne objektovo-atribútový model alebo len kontext), prvky množiny $B$ objekty, prvky množiny $A$ atribúty a reláciu $I$ budeme volať incidenčná.*

Tento kontext môžeme znázorniť ako objektovo-atribútovú tabuľku a definovať nasledujúce zobrazenia:

**Definícia 10** *Definujme zobrazenie $\uparrow\colon \mathrm{P}(B) \to \mathrm{P}(A)$ takto: Ak $X \subseteq B$, tak*

$$\uparrow(X) = X^{\uparrow} = \{y \in A : (\forall x \in X)(x, y) \in I\}. \quad (10)$$

**Definícia 11** *Analogicky definujeme zobrazenie: $\downarrow\colon \mathrm{P}(A) \to \mathrm{P}(B)$ takto: Ak $Y \subseteq A$, tak*

$$\downarrow(Y) = Y^{\downarrow} = \{x \in B : (\forall y \in Y)(x, y) \in I\}. \quad (11)$$

Tieto zobrazenia $\uparrow$ a $\downarrow$ majú podľa [6] nasledujúce vlastnosti:

**Lema 4** *Nech $X, X_1, X_2 \subseteq B$ a $Y \subseteq A$. Potom platí:*

*a) Ak $X_1 \subseteq X_2$, tak $X_1^{\uparrow} \supseteq X_2^{\uparrow}$,*
*b) Ak $Y_1 \subseteq Y_2$, tak $X_1^{\downarrow} \supseteq X_2^{\downarrow}$,*
*c) $X \subseteq X^{\uparrow\downarrow}$,*
*d) $Y \subseteq Y^{\downarrow\uparrow}$.*

Podmienky a) – d) v leme 4 pre dvojicu $(\uparrow, \downarrow)$ sú ekvivalentné tvrdeniu, že pre každé $X \subseteq B$ a $Y \subseteq A$ platí

$$X \subseteq Y^{\downarrow} \text{ akk } Y \subseteq X^{\uparrow}.$$

Táto tzv. duálne adjungovaná dvojica $(\uparrow, \downarrow)$ teda tvorí tzv. antitónnu Galoisovu konexiu medzi množinou objektov $B$ a množinou atribútov $A$.

Zaoberajme sa teraz zložením týchto dvoch zobrazení. Zobrazenie $\mathrm{cl} : \mathrm{P}(B) \to \mathrm{P}(B)$, ktoré vznikne ako zloženie zobrazení $\uparrow$ a $\downarrow$, má nasledujúce vlastnosti:

**Lema 5** *Nech $X, X_1, X_2 \subseteq B$. Potom platí:*

*a) $X \subseteq \mathrm{cl}(X)$,*
*b) Ak $X_1 \subseteq X_2$, tak $\mathrm{cl}(X_1) \subseteq \mathrm{cl}(X_2)$,*
*c) $\mathrm{cl}(X) = \mathrm{cl}(\mathrm{cl}(X))$.*

Z vlastností získaných v leme 5 dostávame, že zobrazenie cl je operátor uzáveru na množine B. Ak by sme poradie zloženia zobrazení $\uparrow$ a $\downarrow$ vymenili, výsledné zobrazenie je operátor uzáveru na množine atribútov A.

Pre niektoré množiny $X$ vo vzťahu $X \subseteq \mathrm{cl}(X)$ nastáva rovnosť, pre niektoré nie. V prvom prípade dostávame tzv. pevné body (fixpointy) zobrazenia cl:

**Definícia 12** *Dvojicu $\langle X, Y\rangle$ nazývame formálny koncept, ak zároveň platí $X^{\uparrow} = Y$ a $Y^{\downarrow} = X$. Množinu objektov $X$ nazývame extentom a množinu atribútov $Y$ intentom tohto konceptu. Množinu všetkých konceptov kontextu $(B, A, I)$ označíme $\mathrm{C}(B, A, I)$.*

Zrejme je $\langle X, Y\rangle$ koncept práve vtedy, keď $X = \mathrm{cl}(X)$ a $Y^{\downarrow} = X$. Koncepty možno prirodzeným spôsobom usporiadať:

**Definícia 13** *Na množine $\mathrm{C}(B, A, I)$ definujme usporiadanie $\leq$ takto: Ak $\langle X_1, Y_1\rangle, \langle X_2, Y_2\rangle \in \mathrm{C}(B, A, I)$, tak položme $\langle X_1, Y_1\rangle \leq \langle X_2, Y_2\rangle$ vtedy, keď $X_1 \subseteq X_2$ (alebo ekvivalentne $Y_1 \supseteq Y_2$). Usporiadanú množinu $(\mathrm{C}(B, A, I), \leq)$ budeme označovať $\mathrm{CL}(B, A, I)$ a nazývať konceptový zväz kontextu $(B, A, I)$.*

Takto definovaný konceptový zväz $\mathrm{CL}(B, A, I)$ je úplným zväzom.

## 6 Ohodnotenie extentov

V tejto časti navrhujeme ohodnotenie extentov konceptov vo formálnej konceptovej analýze. Využijeme náš prístup popísaný v kapitole 4 a uplatníme myšlienku z teórie drsných množín, kde pri modelovaní relácie nerozlíšiteľnosti pre rôzne podmnožiny atribútov môžu byť rôzne aj relácie nerozlíšiteľnosti. Veríme, že tento prístup aspoň z časti pomôže odstrániť problém formálnej konceptovej analýzy, v ktorej získavame veľké množstvo konceptov už pri relatívne malom kontexte.

Zatiaľ sme si povedali, čo to koncept je, ako vytvoríme pomocou usporiadania konceptov konceptový zväz. Pre ďalšie potreby zavedieme označenie $\mathrm{ExtCL}(B, A, I)$ pre množinu všetkých extentov konceptového zväzu $\mathrm{CL}(B, A, I)$.

Ak budeme uvažovať formálny kontext pre ľubovoľnú podmnožinu atribútov, dostaneme $2^n$ rôznych kontextov a k nim prislúchajúcich konceptových zväzov. Skúmajme, ktoré extenty sa tam najčastejšie vyskytli a ohodnoťme túto ich účasť číselne.

**Definícia 14** *Nech $X \subseteq B$ a $Y \subseteq A$. Definujme zobrazenie $\mathrm{NDCL} : \mathrm{P}(B) \to \mathrm{P}(\mathrm{P}(A))$ nasledovne:*

$$\mathrm{NDCL}(X) = \{Y \subseteq A : X \in \mathrm{ExtCL}(B, A, I)\} \quad (12)$$

To znamená, že každej podmnožine objektov priradíme tie podmnožiny atribútov, z ktorých kontextov sme dostali konceptový zväz obsahujúci uvažovanú podmnožinu objektov.

Nakoniec vypočítame hodnotu pre každú podmnožinu a teda definujeme zobrazenie $\mathrm{CE} : \mathrm{P}(B) \to [0,1]$ nasledovne:

**Definícia 15** *Nech $X \subseteq B$, $n = |A|$. Ohodnotením extentu $X$ nazývame hodnotu:*

$$\mathrm{CE}(X) = \frac{|\mathrm{NDCL}(X)|}{2^n} \tag{13}$$

Ilustrujme ohodnotenie extentov na nasledujúcom príklade.

*Príklad 5.* Uvažujme nasledujúci formálny kontext s piatimi objektmi a piatimi atribútmi, na základe ktorého ohodnotíme vzniknuté extenty.

| | **a** | **b** | **c** | **d** | **e** |
|---|---|---|---|---|---|
| **1** | x | x | | | x |
| **2** | | x | | x | |
| **3** | x | | x | | x |
| **4** | x | x | | x | |
| **5** | x | | | | x |

**Tab. 3.** Formálny kontext.

Na základe vzťahov (12) a (13) dostávame napríklad pre množinu $X = \{1,4\}$:

$$\mathrm{NDCL}(\{1,4\}) = \{\{\mathrm{a},\mathrm{b}\},\{\mathrm{a},\mathrm{b},\mathrm{c}\},\{\mathrm{a},\mathrm{b},\mathrm{d}\},\{\mathrm{a},\mathrm{b},\mathrm{e}\},$$
$$\{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d}\},\{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{e}\},\{\mathrm{a},\mathrm{b},\mathrm{d},\mathrm{e}\},\{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d},\mathrm{e}\}\}$$

a ohodnotenie tejto podmnožiny objektov je

$$\mathrm{CE}(\{1,4\}) = \frac{|\mathrm{NDCL}(\{1,4\})|}{2^5} = \frac{1}{4}.$$

V tabuľke 4 uvádzame všetky podmnožiny objektov, ktorých ohodnotenie je nenulové.

Tabuľka 4 obsahuje všetky extenty konceptov (je ich osem, prázdnu množinu a množinu všetkých objektov opäť vynechávame), ktoré sa postupne objavili v rôznych konceptových zväzoch.

Výsledky môžeme interpretovať nasledovne. Koncept, ktorý obsahuje objekty $\{1,3,5\}$ je významnejší ako koncept s objektami $\{1,4\}$. Ak by sme potrebovali vybrať z ôsmich konceptov len najdôležitejšie, bolo by to práve prvých päť. Navyše jednoprvkové koncepty môžeme z našich úvah vynechať pre ich nízku výpovednú hodnotu a tým sa počet najvýznamnejších konceptov opäť zredukuje. Pri interpretácii výsledkov

| $X$ | $\mathrm{CE}(X)$ |
|---|---|
| $\{3\}$<br>$\{2,4\}$<br>$\{1,2,4\}$<br>$\{1,3,5\}$<br>$\{1,3,4,5\}$ | $\frac{1}{2}$ |
| $\{1\}$<br>$\{4\}$<br>$\{1,4\}$ | $\frac{1}{4}$ |

**Tab. 4.** Ohodnotenie extentov.

v našom príklade je potrebné uvedomiť si, že ohodnotenie ilustrujeme na kontexte s pomerne malým počtom objektov a atribútov, a preto aj počet konceptov nie je vysoký. Pri kontextoch s vyšším počtom objektov a atribútov môže byť aj vyšší počet konceptov, v špeciálnom prípade počet konceptov závisí od počtu objektov, resp. atribútov exponenciálne. V takýchto prípadoch väčších kontextov je výber niekoľkých najvýznamnejších konceptov nevyhnutný.

Navyše po hlbšej analýze, ktorú tu neuvádzame, v špeciálnom prípade formálneho kontextu (napríklad kontext uvedený v tabuľke 3), ktorý je bez prázdneho stĺpca, plného riadku, resp. rovnakých riadkov nadobúda ohodnotenie extentov len hodnoty 0,5 a 0,25 ako v tabuľke 4.

## 7 Záverečné poznámky

Ohodnotenie konceptov, resp. ich extentov, je dôležitá úloha aj pre praktické problémy, keďže z veľkého množstva konceptov často potrebujeme vybrať iba niektoré z nich, väčšinou tie najvýznamnejšie. Preto navrhujeme prístup v rámci všetkých kombinácií konceptových zväzov, ktoré získame z rôznych podmnožín atribútov pre formálne kontexty.

Galoisova konexia a operátor uzáveru sú algebraickými štruktúrami, ktoré tvoria základné piliere formálnej konceptovej analýzy. Otázka využitia aproximácií z analýzy drsných množín vo formálnej konceptovej analýze je predmetom ďalšieho štúdia, keďže sme v tejto práci zistili, že v teórii drsných množín vieme nájsť súvislosť so spomínanými štruktúrami ako je Galoisova konexia a operátor uzáveru. V tomto smere chceme prispieť k výsledkom Vlacha [11,12,13], ktoré sa zaoberajú drsnými množinami a ich vzťahom s inými oblasťami.

Vhodným praktickým prínosom získaných výsledkov sa pre nás javí využitie drsných množín pri spracovaní a analýze obrazu vhodnou fuzzifikáciou pri dolných a horných aproximáciách, kde nebudeme uvažovať to, či objekt patrí alebo nepatrí dolnej, resp. hornej aproximácii, ale v akej miere jej patrí.

## Reference


1. Ľ.Antoni, S. Krajči: *Quality measure of fuzzy formal concepts*. Abstracts of 11th International Conference on Fuzzy Set Theory and Applications FSTA 2012, Liptovský Ján, Slovak Republic, 2012, 18.
2. S. M. Dias, N. J. Vieira: *Reducing the size of concept lattices: The JBOS Approach*. In Proceedings of the 8ht International Conference on Concept Lattices and Their Applications, CLA 2010, 80–91.
3. B. Ganter, R. Wille: *Formal concept analysis, mathematical foundations*. Springer Verlag, 1999.
4. M. Klimushkin, S. Obiedkov, C. Roth: *Approaches to the selection of relevant concepts in the case of noisy data*. In: L. Kwuida, B. Sertkaya (Eds), Formal Concept Analysis, volume 5986 of Lecture Notes in Computer Science, chapter 18, Springer Berlin, Heidelberg, Berlin, Heidelberg, 2010, 255–266.
5. J. Komorowski, Z. Pawlak, L. Polkowski, A. Skowron: *Rough sets: a tutorial*. In: S. K. Pal, A. Skowron (Eds), Rough-Neural Computing: Techniques for Computing with Words, Cognitive Technologies, Springer-Verlag, Heidelberg, 2004, 3–98.
6. S. Krajči: *Cluster based efficient generation of fuzzy concepts*. Neural Network World 13(5), 2003, 521–530.
7. S. Krajči, J. Krajčiová: *Social network and one-sided fuzzy concept lattices*. In: FUZZ-IEEE 2007, The IEEE International Conference on Fuzzy Systems July 23-26, London, 2007, 222–227.
8. S. O. Kuznetsov: *On stability of a formal concept*. Annals of Mathematics and Artificial Intelligence 49, 2007, 101–115.
9. Z. Pawlak: *Rough sets Theoretical Aspects of Reasoning about Data*. Kluwer Academic Publishers, Dordrecht, 1991.
10. Z. Pawlak: *Rough sets*. International Journal of Computer and Infomation Sciences 11, 1982, 341–356.
11. M. Vlach: *Topologies of approximation spaces of rough set theory*. Advances in Soft Computing 46, 2008, 176–186, Springer-Verlag, Berlin Heidelberg, Germany. (ISSN 1615-3871).
12. M. Vlach: *Mathematical structures of rough sets*. Proceedings of the 27th International Conference Mathematical Methods in Economics, Kostelec, Czech Republic, September 2009, 335–340.
13. M. Vlach: *Links between extended topologies and approximations of rough set theory*. Multicriteria Decision Making and Fuzzy Systems, Shaker Verlag, Aachen, Germany, 2006, 13–22. (ISBN 3-8322-5540-0).

# Task scheduling in hybrid CPU-GPU systems

Martin Kruliš, Zbyněk Falt, David Bednárek, and Jakub Yaghob *

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 25, 118 00 Prague, Czech Republic
{krulis,falt,bednarek,yaghob}@ksi.mff.cuni.cz

**Abstract.** *The distribution of workload among available computational units is an essential problem for every parallel system. It has been attended thoroughly from many perspectives, such as thread scheduling in operating systems, task scheduling in frameworks for parallel computations, or constrained scheduling in real-time systems. However, each system has unique properties and requirements, thus we cannot design a universal scheduler which would accommodate all of them. In this paper, we propose methods for task scheduling in parallel frameworks that process highly heterogeneous tasks that consists of both computational and CPU-blocking operations. Furthermore, we extend the idea of heterogeneous tasks to design combined scheduler for multi-core CPUs and many-core GPUs. Our methods significantly increase the utilization of hardware resources, which leads to improvement of speedup and throughput.*

## 1 Introduction

Task scheduling is one of the most important issues in any parallel system. If approached from wrong direction, it has the ability to significantly hurt overall performance. It may have also direct impact on other system attributes, such as CPU or memory utilization, thus having serious influence on the power consumption of the whole system.

From the general point of view, there are many quality aspects of a scheduler, like:

- *latency*
- *throughput*
- *fairness*
- *overhead*
- *scalability*
- *hardware utilization* (and power consumption)

The importance of the aspects differs with each system. For instance, the thread/process scheduler of an operating system is most concerned with latency and fairness, while the tasks scheduler of a parallel framework for high performance computations requires especially high throughput and small overhead.

As this topic is too broad to be overviewed within one paper, we will focus on computational parallel frameworks designed to process large datasets. Therefore, we will not consider fairness nor latency, as we expect that our framework is always processing one problem (which consists of many tasks) and all hardware resources (CPUs, GPUs, ...) are allocated solely for the framework at the time. As we have restricted the domain of the problem, we can discuss some generic attributes of task schedulers.

The scheduler should focus mainly on *throughput* and *scalability*. The throughput defines how large data could be processed at a unit of time or how fast a problem of fixed size could be solved. Scalability reflects the overhead and limits of the parallel processing. It helps us predict the future and determine, how would the same application work on newer hardware with more processing units. We will also monitor hardware utilization closely since idle computational units usually suggest some room for improvement.

The scheduler should be *non-preemptive*. Preemptive schedulers are able to pause running tasks, suspend them and switch them for another task, thus allowing multiple tasks to run on the same processing unit in quasi-parallel manner. Preemptive execution is very important for latency and fairness of the scheduling. However, since these two aspects are no concern to us and preemptive scheduling (which requires interrupt and context switching) is bound with nontrivial overhead, we can dismiss it. There seems to be no point of swapping running tasks on the processing units when the system needs all the tasks to terminate anyway.

Load balancing is very important especially in non-preemptive systems. We can choose either static or dynamic load balancing. Static load balancing usually requires at least some assumptions about the executed tasks. Since we have no such assumptions and we want to deal with all kinds of tasks (including blocking, or GPU bound tasks), we will use dynamic load balancing in combination with *oversubscription* technique. The oversubscription expects that the application will generate many small tasks rather than few large ones, so the number of tasks significantly exceeds the number of availabile processing units. On the other hand, tasks should be large enough so that the scheduling overhead remains still negligible.

* This work was supported by the Grant Agency of Charles University (GAUK) project no. 277911 and by the specific research grant SVV-2012-265312.

The problem of uniform task scheduling has been investigated thoroughly as we show in Section 2. Several frameworks are available and there is little that can be done to improve them. We will focus mainly on the problem of nonuniform tasks (e.g., combining computational and I/O task) and hybrid CPU-GPU platforms. Section 3 describes our CPU scheduler for Bobox framework which is ready to deal with heterogenous tasks.

In past few years, common GPU cards evolved enough to encompass general purpose computations in addition to their graphic-related capabilities. GPUs are currently used as computational coprocessors, that can take some workload and ease the overloaded CPUs. However, GPUs are designed for quite special tasks and they are incapable of running complex systems, thus the presence of CPU is still required. We have to deal with a hybrid system, that needs to schedule tasks for both CPUs and GPUs efficiently. Task scheduling and dispatching is much more complicated in such systems as the task dependencies may create periods of time when CPU waits for GPU or vice versa. Details of GPU systems and of our proposed scheduler are described in Section 4.

We present several practical applications of our schedulers to demonstrate benefits of our approaches in Section 5. Section 6 concludes the paper.

## 2 Related work

### 2.1 Task scheduling on CPUs

The task scheduler is an essential part of any framework for parallel applications such as Intel Threading Building Blocks [14] or OpenMP [5]. Therefore, many of the existing works focus on its effective implementation [12, 6, 3].

The common idea of these works is to have a *thread pool*, which contains so called *worker threads* (or *workers*). The workers wait dormant until a task appears. When it does, a worker from the pool is selected by the scheduler and the task is dispatched to it. The thread pool is prefered to other solutions since waking and blocking existing threads has significantly lower overhead than creating and destroying them. The thread pool usually contains as many workers as there are CPU cores available, so in ideal case, each worker occupies exactly one core.

The schedulers chose different strategies for dispatching tasks to the worker threads. In the remainder of this section, we describe the scheduling strategies of two the most popular parallel frameworks.

**Threading building blocks.** The scheduler is thoroughly described in the TBB Reference Manual [1]. Basically, it keeps a pool of worker threads, where each worker thread has its own double ended task queue. When a task is spawned, it is inserted to the end of the queue of a thread that spawned it. When a thread terminates current task and looks for another one to execute, it takes the newest spawned task (i.e., the task in the end of its queue). If the local queue of the thread is empty it attempts to steal the oldest task (from the beginning of the queue) of another thread. These rules increase data locality. The newest tasks have the biggest probability to have their data still hot in the caches.

The scheduler is non-preemptive – the evaluation of a task cannot be interrupted and the blocking operations called from the tasks are not recommended. Additionally, the TBB scheduler lacks support for NUMA[1] systems.

**OpenMP.** The OpenMP task scheduler is very similar to TBB scheduler, thus we focus only on differences. The most important one, is that the evaluation of tasks may be interrupted in so called *synchronization points* under certain circumstances. These points might be stated explicitly by the programmer or inserted automatically by the compiler. Interruptions are used for evaluation of child tasks, for instance. However, the scheduling is still non-preemptive and it does not support blocking operations.

The second largest difference is that there are modifications of the OpenMP scheduler that include support for NUMA systems and shared caches [13]. This increases scalability of OpenMP on modern systems.

### 2.2 Task scheduling in hybrid systems

The first attempt for hybrid tasks scheduling was incorporated into the OpenCL framework. The OpenCL specification defines out-of-order command queues in which the commands are enqueued and the framework executes them concurrently if possible. Unfortunately, this interface is too crude and most implementations do not cope well with the scheduling problem.

To our best knowledge, there is only one project that attends the problem of hybrid scheduling seriously. The StarPU [2] is an unified platform for scheduling tasks on heterogeneous multi-core architectures. Their implementation is built on the top of OpenCL as well as ours. However, their goal is to automatize as many operations as possible and offer simpler parallel model to the programmer, so it can be used by scientists from other fields (physics, biology, ...) for high performance computations. The cost

[1] Non-uniform memory access/architecture.

for this generalization is releasing control of some crucial operations to the framework, which can lead to suboptimal performance in some cases. Our main objective is to provide a framework that will also simplify the design of GPU applications, but we have chosen a lower level approach that allows the programmer to remain in control of every aspect of the application.

## 3 Bobox scheduler

The Bobox [7, 4] is a highly parallel framework designed for data processing applications. It concurrently evaluates so called *execution plans* on available CPUs. These plans consist of boxes connected together by directed edges, which determine the data flow among them. When a box receives some data, it becomes active and active boxes are planed on available threads by Bobox scheduler. The threads execute internal methods of the boxes to process the input data, produce the output data, or alter the internal state of the box.

The scheduling strategy of the Bobox system is described in standalone paper [8]. The scheduler takes many various factors into account:

1. It attempts to increase data locality by taking data flow into account. The data are produced and consumed in the same thread if possible.
2. It is aware of shared caches. In case of task stealing, the thief-thread prefers threads with which it shares cache as victims.
3. The scheduler is NUMA aware. The evaluation of an execution plan is kept on one NUMA node if possible. However, the scheduler tries to keep the workload of nodes well balanced and performs a migration of the execution from the overloaded NUMA node to another if needed. Therefore, when multiple plans are executed, each plan runs on a single node, thus the data locality is increased. On the other hand, if there are only a few running plans, they may spread to other nodes to exploit the full potential of all available CPUs.

Since the main objective of the Bobox framework is data processing, it provides support for I/O operations. An I/O operation is usually blocking, which means that the calling thread is suspended by operating system, until the requested data are read from or written to a storage device. However, during this operation another box may process its data or perform some computations.

The Bobox scheduler provides functions `detach` and `attach`, which may be called from the internal procedure of the box. The `detach` function informs the scheduler, that the box is about to perform a blocking call. At that moment, the scheduler activates a *backup thread*, which performs the blocking call, and the original thread takes another active box for processing. If there are no backup threads available, the blocking operation is enqueued and it will be processed as soon as one of the blocking threads becomes available.

After the blocking call terminates, the box invokes `attach` method, which informs the scheduler that the blocking call has finished, so the scheduler can eventually suspend the backup thread again and the rest of the internal procedure is finished by a regular worker.

If every blocking operation is correctly enclosed by these functions, the number of active computing threads remains constant, thus the parallelism is not restricted. However, this approach has some technical problems with which needs to be attended:

1. The scheduler has to keep two thread pools (the pool of worker threads and the pool of backup threads) to implement the `detach` and `attach` operations efficiently without the overhead of creation and destruction of the backup threads.
2. The I/O operations usually interfere with themselves. For instance, a parallel access to a storage device may be even slower than a sequential access, as it may lead to large number of seek operations. Therefore, the size of the pool of backup threads has to be reasonably limited.

The backup threads significantly increase the parallel potential of the application in many scenarios. However, our experiments indicate that the scheduler also needs to be informed of the type of the blocking operation, since the optimal number of backup threads depends on it heavily (see Section 5).

## 4 Hybrid CPU-GPU scheduler

In the following section, we attend the problems of hybrid computational systems. First, we review the GPU computing principles and API from the perspective of task scheduling. This environment brings us many challenges, both performance and technical. In order to identify them properly, we present a few model use cases that reflect our experience in GPU algorithms [11, 10, 9]. We describe our scheduler and advocate its design process at the end of this section.

### 4.1 GPU computing and OpenCL API

The GPU cards are quite independent on the host system. They have their own memory and the GPU chip is capable of executing pieces of code on its own. On the other hand, GPU processors are much more simple

than CPUs and GPU memory does not employ any access protection mechanisms such as paging or segmentation, so the GPU cannot run independent operating system. All computational tasks and all host-device memory transfers are issued from the code running on CPU. This have several advantages:

- The GPU code can focus solely on computations (no interrupt handling, I/O management, ...).
- The CPU can perform other tasks while the GPU is computing.

But also several disadvantages:

- The input data must be transferred to the GPU memory and the results must be transferred back. These transfers are performed over PCI-Express bus which is rather slow in comparison with internal GPU busses or CPU-memory bus.
- The CPU code must explicitly issue every command to the GPU.

The GPU programming holds many challenges, but most of them are no concern to the task scheduling. Henceforth, we denote a *GPU task* as the following sequence of operations:

- A transfer of the input data from the host memory to the GPU device memory.
- An execution of a GPU method (called *kernel*) or of a sequence of such methods.
- A transfer of the results from the GPU device back to the host memory.

The data transfers may also include some nontrivial work for CPU, such as gathering/scattering data or converting data to different format that is more suitable for GPU memory architecture.

There might be more complex situations that would require more complex model of a GPU task. However, our case study shows that this model is sufficient to design efficient hybrid scheduler.

**OpenCL framework.** OpenCL is a framework for parallel computations designed not only for GPU computing, but for other parallel devices as well. We have chosen this framework over the alternatives[2] as this is currently the only framework that supports both NVIDIA and AMD GPUs and runs on multiple platforms (Windows, Linux, and OS X). Furthermore, the framework is designed to work also with other parallel devices, thus we can expect that our scheduler could be extended to other hybrid systems.

The API that OpenCL provides is quite simple and easy to use. The host program can detect parallel devices that support OpenCL and create *command queues* attached to these devices. A command queue is then used to issue commands to that device, most importantly copying memory buffers and executing kernels on the device.

The procedures (*kernels*) that can be executed on the device are provided as source code (usually in text form) and compiled by the OpenCL. This way the code is fully optimized for the target device that is detected at runtime. Even though there are some minor technical issues concerning the kernel compilation and management, we simply assume that all required kernels are compiled at the application startup and available for every device in the system.

Since the device have memory of its own, the memory allocation must be handled by the framework aswell. The OpenCL provides *memory buffer* objects that encapsulate memory blocks allocated on GPUs. Writing and reading data to and from these buffers can be issued via command queues.

### 4.2 Identifying problems on model cases

Each of the following models describe a possible use case of the GPU framework. These models do not cover every conceivable scenario, but rather help us understand and identify problems of the hybrid scheduling.

- The most *simple case* is using a single GPU task. This scenario is possible only if all input, intermediate, and output data fit in the GPU memory. Multiple simple tasks may be executed concurrently, but they solve different problems, thus they are completely independent from the scheduling point of view.
- If the data of the GPU task do not fit the GPU memory, or they are being streamed, multiple GPU tasks are required. In this *iterative case*, the data are divided into blocks and all the blocks are processed by GPU tasks that perform the same algorithm (kernel).
- The GPU algorithm might require some data structures to be always present in the GPU memory (e.g., a precomputed read-only lookup table or intermediate results that are updated by GPU tasks). We designate this case the *incremental case*.

We have implemented all models in a naive way and test them in various combinations and settings using profiling techniques. Following problems has been identified as a result of our experiments:

- There is a fragile balance between CPU and GPU workloads. In many situations the CPU was waiting for the GPU and vice versa.
- The data transfers are especially problematic as they take considerable time and stall both CPU

[2] NVIDIA CUDA and DirectX Compute

and GPU. The data transfers need to overlap with computations, in order to reduce their effect on the performance.
- The data transfers are most efficient if aggregated in only a few bulk transactions. Unfortunately, the data gather operations that compacts the input data in one block and the scatter operation that processes the results of GPU task take approximately the same time as the transfers themselves.
- Allocation and deallocation of GPU memory is also bound with nontrivial overhead. It might be beneficial to reuse allocated buffers, especially in the iterative case.

We must also consider all model cases from the perspective of the multi-GPU systems. The simple case scenario is best suited for single-GPU system, however, we can still benefit from having multiple GPUs if there are more subproblems to be solved by separate simple tasks. Furthermore, we can divide simple GPU task in multiple tasks and use the iterative method if the task is truly data parallel.

The iterative case scales almost ideally with the number of GPU devices available. We only assume that there are more GPU tasks than GPUs. If not, the size of the data blocks must be reduced so that more tasks are spawned. Finally, the incremental case has to use data replication so that the required data structures are copied to every GPU. This can be done only during initialization stage in case of read-only lookup tables, or it must be performed on regular basis, if the data are mutable.

### 4.3 Scheduling GPU tasks

We have designed our GPU scheduler as a flexible module which can be combined with various parallel frameworks for multicore CPUs. We describe the possibilities of interoperation with them in Section 4.4.

The scheduler has two parts – a *GPU wrapper* that provides more suitable access to OpenCL API and *feeding thread pool* of CPU threads. The overall design is depicted in Figure 1.

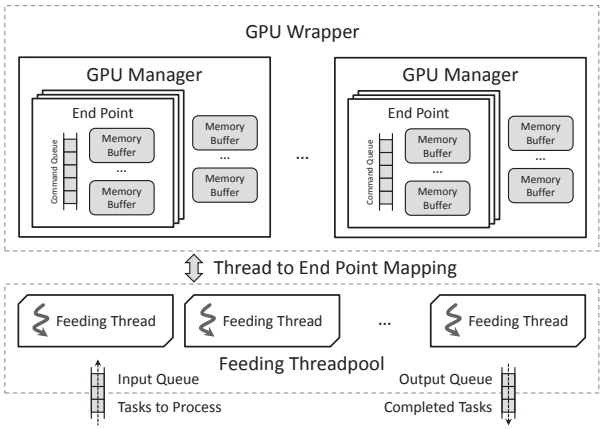



**Fig. 1.** The design of the GPU scheduler.

**The GPU wrapper** is an object oriented API built on the top of the OpenCL framework. It manages devices, memory buffers, and kernels. The wrapper is represented by singleton object, which is also a container for device managers. Each detected device in the system is handled by a *device manager* object which provides the API to work with the device and manages necessary OpenCL structures (handles, context, etc.).

Each device manager is equiped with one or more *end points*. An end point is a logical structure that encapsulates one command queue. Multiple end points may be attached to one device so that we can achieve concurrent operations running on the device – especially, overlap of the kernel execution and the data transfers.

The end points are also responsible for managing memory buffers. We recognize three types of buffers:

- *Anonymous buffers*, which are allocated and deallocated by the GPU task itself and they belong locally to the end point. They are most suitable for simple cases, when the buffer is used only once.
- *Replicated buffers*, which are allocated and registered under a name before the GPU tasks are dispatched to the scheduler. They are allocated through device manager, which ensures that each end point allocates its own buffer, thus when the GPU tasks are scheduled, they may use any end point with the same functionality. These buffers are designed to be used in iterative cases.
- *Shared buffers* are similar to replicated buffers, but only one instance is allocated by the device manager and it is shared by the end points. They are designed for incremental cases. Note that shared buffers do not ensure data replication/sharing between multiple devices. Every iterative case we have examined so far uses different approach to replication and we have not found a replication technique that would be both universal and efficient.

**The feeding thread pool.** It was created in order to reduce the amount of time the GPU spends waiting for the input data from the CPU. The threads are handling the CPU part of the GPU task, which usually consists of gather/scatter routines and data transfers. The tasks are dispatched to the pool by one input queue. This queue is thread-safe and blocking, thus it easily synchronizes the threads. We did not used other techniques like task stealing for several reasons. This design is much simpler, the GPU tasks do not spawn

another tasks, and the locking overhead of the queue is negligible in comparison with the execution time of the GPU tasks. Completed tasks are stored into output queue, which can be accessed by the remaining parts of the system.

Mapping between feeding threads and GPU end points is not fixed, so we can chose different approaches for different scenarios. One end point is always mapped to exactly one feeding thread, but each thread may have multiple end points assigned. If so, the thread dispatches GPU tasks to the end points in round-robin manner. In common cases, we have used thread-per-end-point and thread-per-device mappings.

This concept was designed under the assumption that there are always more CPU cores than feeding threads. However, we have observed that there is no reason to have more than two end points per GPU in normal situations, thus a four-GPU system will require at most 8 feeding threads. Mainstream CPUs have 12 logical cores at present time and multiprocessor servers are also quite common. For these reasons we advocate that our assumption holds for the current mainstream hardware.

### 4.4 Hybrid scheduling

If the majority of the work is performed in the GPU tasks, the GPU scheduler can be used with only single main thread. The main thread may influence the scheduling up to certain level by issuing the tasks to the input queue and waiting for them to complete. This way it may control the dependencies between tasks or their priorities.

Since the input and output queues are thread safe, the GPU scheduler may be used in combination with any other parallel framework such as TBB or OpenMP. Parallel frameworks usually create as many threads as there are CPU cores available. Since our scheduler has a thread pool of its own, there will be more threads than CPUs, thus they will be planed on available cores by the scheduler of the operating system. This approach is not ideal; however, since we cannot easily modify schedulers of existing frameworks, this solution is the best available. We can improve the situation by increasing thread priority of the feeding threads if the operating system supports it.

If we combine out GPU scheduler with Bobox system, we may use the same approach as if we are dealing with blocking operations. When a GPU task executes its kernel on the GPU and waits for it to terminate, it may use the `detach` and subsequently the `attach` method to regulate the number of currently available threads in the Bobox thread pool. This way the number of active computing threads remains the same as the number of available CPUs, thus the feeding threads are not suppressed by computing threads and waiting operations do not reduce parallelism.

Our current implementation of the GPU scheduler operates with one input and one output task queue. It will be trivial to modify it to use multiple input/output queue pairs, so that that multiple independent parts of the system may easily interoperate with our GPU scheduler.

## 5 Experiments and applications

In this section, we demonstrate effectiveness of our methods on several examples. Providing a full-scale testing of various scenarios is beyond the scope of this paper, thus the following experiments are considered to be a proof of concept that our scheduling approach improves the performance in common situations.

### 5.1 Hardware and methodology

The following experiments are oriented on performance, so the system real-time clock was used to measure the time required to complete each test. We realize that these times are strongly dependant on the hardware, used compiler, or implementation details. However, we have tried to maintain the same conditions for corresponding tests and we are mainly interested in relative speedup rather than absolute time values. All measured values should be perceived in such context.

The Bobox experiments were performed on a Dell server with two Xeon E5310 running at 1.6 GHz with 8 GB RAM and with two local 73 GB HDDs with 15 000 rpm connected in RAID 1.

The GPU experiments were performed on a server with special motherboard (FT72-B7015) designed to embrace up to 8 GPU cards. The server was equipped with Xeon E5645 processor that contains 6 physical (12 logical) cores running at 2.4 GHz, 96 GB of DDR3-1333 RAM, and 4 NVIDIA Tesla M2090 GPU cards based on Fermi architecture. Each GPU chip consists of 512 cores (32 cores per 16 SMPs) and 6 GB of memory. We also tested the implementation on commodity PC with two NVIDIA GTX 580 which have also 512 cores, but only 1.5 GB of memory. We have found that the GTX 580 cards have similar performance as the Teslas, thus we do not provide detailed comparison.

### 5.2 Blocking operations in Bobox

As we mentioned in Section 3, the most significant difference between the Bobox system and other libraries is the support of blocking operations. Therefore, both experiments in this section focus on them.

The first experiment compares how the TBB library and the Bobox deals with I/O operations in the tasks. For the experiment, we have generated a 4 GB file of random data. Then we ran 100 requests in parallel, where each request reads 32 MB (of 32b integers) from a pseudorandom position in the file and sorts them using `std::sort`. The filesystem cache was flushed before each experiment and for each experiment the same sequence of random positions was used.

The TBB experiment used `parallel_for` template to execute the requests in parallel. The Bobox framework was tested under multiple settings. The first setting does not call `detach` and `attach` functions. Remaining settings enclose all I/O operations with these function calls correctly, but with different sizes of the backup thread pool.

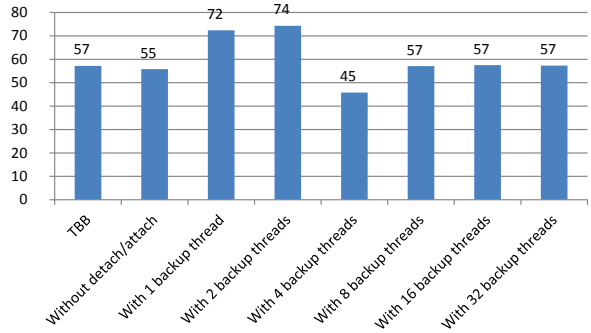



**Fig. 2.** Real times (in seconds) of the I/O tasks.

The results of this experiment are summarized in Figure 2. They indicate that if an appropriate number of backup threads is chosen, the Bobox significantly outperforms the TBB library. We have determined (and also verified by additional tests) that the our hard drive and the I/O controller work on their peak performance if four parallel reads are performed. For some reasons, the performance in other cases is much worse even than the performance of single threaded version. Therefore, we can choose four backup threads, thus four parallel reading operations, while the remaining worker threads can work on sorting operations using the full potential of our multi-core CPU.

The second experiment uses mutually independent blocking operations. It simulates I/O operations performed on independent storage devices, asynchronous tasks dispatched to co-processors, or waiting for response from peer servers participating in a distributed computation. For the sake of simplicity, the blocking operation is represented by the `sleep` system function that suspends calling thread for 2 s.

We used the same configuration for TBB and Bobox as in the previous experiment. The results are presented in Figure 3. As expected, they show that the overall speedup is proportional to the number of backup threads. We have observed that if

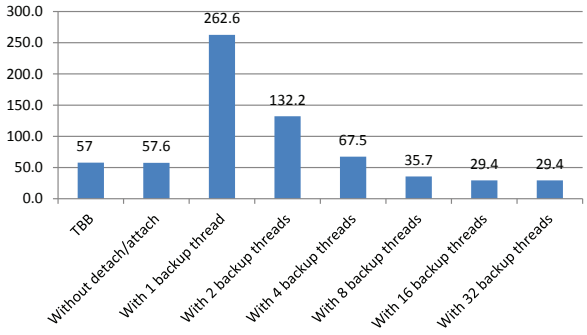



**Fig. 3.** Real times (in seconds) of the sleeping tasks.

less than 8 backup threads are used, the performance is worse than the performance of the solutions without detach/attach operations. On the other hand, with 16 threads we approach the 2× speedup with respect to the TBB solution. The poor performance observed for the small backup thread pools can be expected in this case, since the `sleep` function is called only from the backup threads and not from the worker threads.

Use of `detach` and `attach` functions improved performance significantly. However, the proper size of the backup thread pool must be determined. We will focus on this area in our future research.

### 5.3 Similarity search on GPUs

To present the benefits of our GPU scheduler, we chose an image similarity search problem. This problem and its GPU solution has been thoroughly described in previous work [10, 9]. We will summarize it briefly just for the purposes of these tests. The problem of similarity search is based on query-by-example paradigm. Let us have a database of images that are not annotated or otherwise classified. The user cannot search such database using conventional text-query interface, but rather provide an example image and expects to get similar images in response.

The images are represented as signatures, each signature is a set of points in 7-dimensional space with weights. A metric distance function (Signature Quadratic Form Distance in our case) is defined to compute distance (inversed similarity) of image signatures. The distance between a query signature and all (or at least a subset of) the signatures in the database needs to be computed in order to determine the results of the query. The distances are computed iteratively on available GPUs and each GPU task has following steps:

1. Gather image signatures into single block.
2. Copy the signatures to GPU in one transfer.
3. Invoke SQFD kernel that computes distances from query to all signatures in the block.
4. Transfer the distances back to host memory.
5. Use the distances to determine which images from the block will be included into the result.

In the following experiments we compare original naive approach which uses single CPU thread to both dispatch work to GPUs and process the distances to create a result with a solution that uses our GPU scheduler. The scheduler uses 2 end points per GPU[3] and one feeding thread per end point. We provided results measured for different numbers of GPU cards.

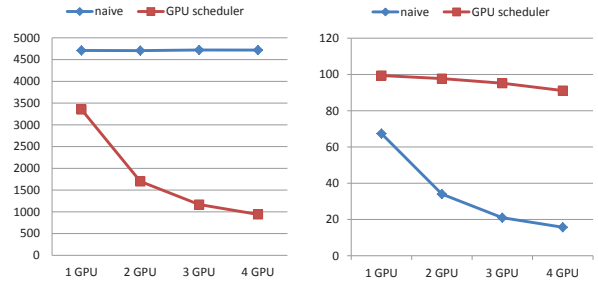



**Fig. 4.** Measured real times in ms (left) and average utilization of GPUs in percent (right).

Figure 4 depicts the results of experiments. The left graph shows absolute times required for processing database of 950,000 images. As we can see, the feeding threads help significantly in achieving better performance (1.4× for single GPU and 5× for 4 GPUs) and the GPU scheduler scales almost idealy with increasing number of GPUs. The reason for this behaviour is visualized in the right graph which shows the utilization of GPUs in percents of total time. The GPU scheduler is capable to utilize even 4 GPUs more than 90% of time. Naive approach utilizes single GPU only up to 65%, and more GPUs does not improve the situation since their utilization decreases in the proportion of their numbers. Furthermore, we have observed that GPU scheduler overlaps data transfers with SQFD computations significantly, while the naive approach does not.

## 6 Conclusions

We have proposed a method which successfully deals with blocking tasks in parallel environment and exhibits a significant speedup over traditional parallel frameworks. Furthermore, we have proposed a GPU task scheduler that can be easily combined with any other parallel framework for CPUs and create a hybrid CPU-GPU task scheduler. Our GPU scheduler is much faster than naive approach to OpenCL GPU programming and we have successfully tested its scalability up to 4 GPUs.

We were not able to provide any experiments of the GPU scheduler in combination with Bobox system. Preliminary results are promising, but the integration is still an ongoing process and we will address it in the future work. We are also planing to combine our scheduler with a message passing framework (like OpenMPI) to test it in the distributed environment.

[3] This was empirically determined as an optimum for overlapping data transfers and computations.

## References

1. *Intel(R) Threading Building Blocks - Reference Manual*, 315415-016us edition, January 2012.
2. C. Augonnet, S. Thibault, R. Namyst, P. A. Wacrenier: *Starpu: A unified platform for task scheduling on heterogeneous multicore architectures.* Concurrency and Computation: Practice and Experience 23(2), 2011, 187–198.
3. J. Bircsak, P. Craig, R. L. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, C. D. Offner: *Extending OpenMP for NUMA machines.* In: Supercomputing, ACM/IEEE 2000 Conference, IEEE, 2000, 48–48.
4. M. Cermak, Z. Falt, J. Dokulil, and F. Zavoral: *Sparql query processing using bobox framework.* In: SEMAPRO 2011, The Fifth International Conference on Advances in Semantic Processing, 2011, 104–109.
5. R. Chandra: *Parallel programming in OpenMP.* Morgan Kaufmann, 2001.
6. A. Duran, J. Corbalán, E. Ayguadé: *Evaluation of OpenMP task scheduling strategies.* In: Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism, Springer-Verlag, 2008, 100–110.
7. Z. Falt, D. Bednarek, M. Cermak, F. Zavoral: *On parallel evaluation of SPARQL queries.* In: DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications, 2012, 97–102.
8. Z. Falt, J. Yaghob: *Task scheduling in data stream processing.* In: Proceedings of the Dateso 2011 Workshop, Citeseer, 2011, 85–96.
9. M. Krulis, T. Skopal, J. Lokoc, C. Beecks: *Combining CPU and GPU architectures for fast similarity search.* Distributed and Parallel Databases, in press, 2012.
10. M. Krulis, J. Lokoc, C. Beecks, T. Skopal, T. Seidl: *Processing the signature quadratic form distance on many-core GPU architectures.* In: CIKM, 2011, 2373–2376.
11. M. Krulis, J. Yaghob: *Revision of relational joins for multi-core and many-core architectures.* In: DATESO, 2011, 229–240.
12. A. Kukanov, M. Voss: *The foundations for scalable multi-core software in Intel Threading Building Blocks.* Intel Technology Journal 11(4), 2007, 309–322.
13. S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, J F. Prins: *OpenMP task scheduling strategies for multicore NUMA systems.* International Journal of High Performance Computing Applications, 2012.
14. J. Reinders: *Intel threading building blocks.* O'Reilly, 2007.

# Validation of stereotypes' usage in UML class model by generated OCL constraints*

Zdenek Rybola[1] and Karel Richta[2,3]

[1] Faculty of Information Technology, Czech Technical University in Prague
rybolzde@fit.cvut.cz
[2] Faculty of Mathematics and Physics, Charles University in Prague
richta@ksi.mff.cuni.cz
[3] Faculty of Electrical Engineering, Czech Technical University in Prague
richta@fel.cvut.cz

**Abstract** *The Model Driven Development approach became popular in the past years. Domain-specific profiles are defined for various domains and tools are used to transform UML class models using these profiles to source code artifacts. However, rules need to be defined for the profile elements' usage so the transformation can be effective and reliable. The paper deals with an approach of expressing these specific rules using a special type of meta-model using UML class diagram notation with the stereotypes defined in the profile – we call them constraint diagrams. In these diagrams we can restrict usage of specific stereotypes according to the other connected stereotypes. OCL invariants can be generated from these diagrams that can be used to validate a model that uses the profile. The approach is illustrated on an example of a UML profile for J2EE and Flex application.*

## 1 Introduction

Model Driven Development is a modern and popular software development approach. It is based on Model Driven Architecture (MDA) [1] and consists of creation of models of various abstraction levels and transformations between these models. It also includes forward and reverse engineering processes. Forward engineering becomes especially popular for generation of source code from models.

Models of software systems are usually created using Unified Modeling Language (UML) notation [2,3] and transformed to a source code using a tool that generates all required artifacts from the model. For instance, many various artifacts for J2EE application such as Entities, Session beans, Message-driven beans and many others can be generated from a UML class model. For various domains of software systems, domain-specific UML profiles are usually defined. UML profile [2] is a meta-model that allows the definition of stereotypes and tagged values for model elements. If a profile is defined and used, the model transformation process can be adapted according to the specified stereotypes and tagged values. However, special rules usually come with the profile to restrict usage of various profile artifacts that need to be satisfied by any model based on the profile. To make those adapted transformations effective, model validation against the rules defined in the meta–model is required.

In our current research, we deal with an approach to express the rules for the use of the UML profile stereotypes by a generally known notation of UML class diagrams. These diagrams can be easily transformed by a tool such as Dirigent [4] to OCL constraints that can be used for model validation using various CASE and OCL tools. Object Constraint Language (OCL) [5] is a specification language used to define restrictions such as invariants – conditions that must be satisfied by all instances of the element –, pre- and post-conditions for connected model elements.

In [6], we introduced the basic idea of validating the model by generated OCL invariants from a special class diagram. In this paper, we define the more precise semantics of the constraint diagram. We also add the possibility to restrict the number of classes connected by the same stereotype from a single class in the model and the possibility to allow various number of stereotypes being connected by the same stereotype from a single class.

The paper is structured as follows: Section 2 presents related work and their difference from our approach. In section 3, we explain our approach. The UML profile definition, the application model creation, the rules definition and modeling and the constraints generation processes are explained in its respective subsection. Conclusions and further work plans are given in section 4.

## 2 Related work

There have been done some research on model checking and model validation using OCL. Richters and Gogolla [7] presented an approach of animating model

* This research was partially supported by Grant Agency of CTU No. SGS12/093/OHK3/1T/18, and partially also by the AVAST Foundation

snapshots and validating it against OCL constraints. The authors use USE tool [8] for model animation and validation. The authors also introduced a method of automatic model snapshot generation in USE tool [9]. However, the authors only generate snapshots of the model and the constraints must be defined directly in OCL.

Some research was made on model validation against fundamental properties of models defined in UML. Chae et al. [10] focuses on analysis class model used in many standard object-oriented processes where three basic stereotypes are defined for analytical classes – boundary, control and entity. The authors define a set of constraints in OCL that any analytical class model should satisfy including constraints for associations between classes of particular stereotypes. The authors also demonstrate a case study of validation of analysis models against defined OCL constraints using *OCLE* tool [11]. However, the authors only define particular constraints for these three basic stereotypes of analytical classes.

In [12], Guizzardi defines ontological extension of UML class diagram with ontology stereotypes and constraints called *OntoUML*. He presents a fine-grained approach for domain modeling using stereotypes to distinguish between various types of domain artifacts and relationships. In [13], Benevides and Guizzardi present a graphical editor for OntoUML and validation of OntoUML model using OCL constraints. These constraints are based on stereotypes of model elements and defined according to the specification of OntoUML.

In comparison to the approaches mentioned above, in our approach we do not define any particular constraints for any particular domain. Instead, we propose a general approach to create a constraint diagram using a domain-specific and user-defined profile to model domain-specific business rules. This constraint diagram is created using the well-known UML class diagram notation. It can be used to generate OCL constraints for validation of any model based on the user-defined profile. In this paper, we deal with generating OCL constraints to restrict the use of profile stereotypes and particular relationships between various stereotyped classes. We focus on UML class diagram models.

There is also a lot of tools that support model transformation and validation against OCL constraints. *Dirigent* [4] is a tool that can be used to transform models to source code files. The tool parses model created in Enterprise Architect [14] from the exported XMI or directly from the repository. A pattern can be defined for a class with particular stereotype with a set of Velocity [15] templates for various source code artifacts such SQL scripts, java classes and more. The tool tries to match each element of the model to the patterns provided. When a match is found, all the Velocity templates defined in the pattern are evaluated and source code files are created or updated.

Dirigent could also be extended to validate the model using the generated OCL constraints itself. However, there are also many other tools that support model validation using OCL constraints. OCLE tool [11] can be used as mentioned above. *DresdenOCL* [16] is a toolkit for creating and validating OCL constraints for specified model. It also supports model validation and transformation of the model along with the constraints to SQL or Java with AspectJ.

## 3 Our approach

In our approach, we do not try to define a set of exact constraints for any particular domain or programming language. We propose a method to define the rules for the usage of a UML profile stereotypes using the same constructs as used in the model itself. We define a *constraint diagram* that is based on the UML class diagram notation. In this diagram, designers and developers can express the rules using a familiar and well-known notation. From this diagram, OCL invariants can be generated using a tool such as Dirigent that can be used to validate the model.

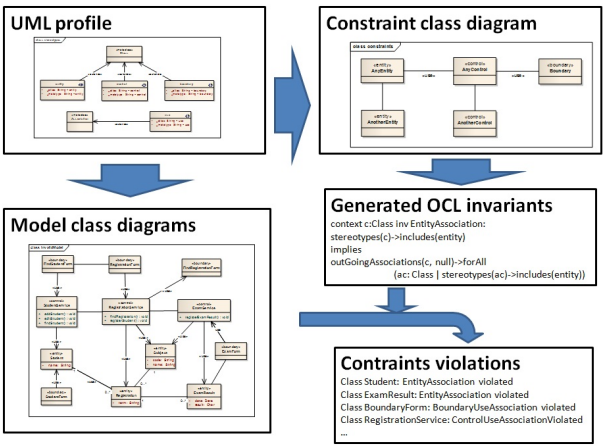



**Fig. 1.** Overview of the model validation process.

An overview of our approach is shown in Fig. 1 and can be expressed by the following steps:

1. Profile definition
2. Application model creation
3. Rules definition
4. Rules modeling
5. Constraints generation
6. Model validation

Each of these steps are explained in more details in the following subsections.

To demonstrate our approach to model validation, we created an example of a model of a J2EE application with the user interface written in Adobe Flex and ActionScript [17]. In such an application, value objects are used as containers to transfer data from the Java business layer to the ActionScript and Flex presentation layer and vice versa. The value objects are not persistent and persistent data from entities is transferred to these value objects first before transferring to the presentation layer.

We use Dirigent to generate the source code files of an application. Dirigent parses a model and generate source code files according to the stereotypes of the model elements. Using this tool, various types of classes can be generated for a J2EE application such as entity classes, data access classes, session bean classes and more. Furthermore, if extended appropriately, Dirigent would be able to parse the proposed constraint diagram and generate OCL constraints for validation of a model based on the same profile. However, this extension is not realized yet but planned for the near future.

### 3.1 Profile definition

To make transformations of a domain-specific model in UML to source code files effective, a domain-specific UML profile should be defined. Such a profile includes mostly stereotypes of classes and associations for class diagrams and can define tagged values of such stereotypes as well. In the model, various defined stereotypes are used to model various types of artifacts using classes and their associations. During the transformation of the model to source code files, various stereotyped classes can be transformed to various source code artifacts such as J2EE entities, session beans or servlets.

The UML profile of our example is shown in Fig. 2. It defines three stereotypes for classes – *Entity*, *ValueObject* and *FormView* – and three stereotypes for associations – *DataLink*, *FormModel* and *FormData* with the following meaning:

Entity – A class with the *Entity* stereotype represents a persistent class for data stored in the database.

ValueObject – A class with the *ValueObject* stereotype represents a class for data not stored in the database and used to encapsulate the data for the transfer between various business services in Java and presentation layer in ActionScript.

FormView – A class with the *FormView* stereotype represents a form that is a part of the user interface of the application developed.

DataLink – An association with the *DataLink* stereotype represents a standard data association between two objects.

FormModel – An association with the *FormModel* stereotype links an Entity or a ValueObject to a FormView to define the form structure and a container to hold the form data.

FormData – An association with the *FormData* stereotype links an Entity or a ValueObject to a FormView to provide lists of values for comboboxes and similar form elements.

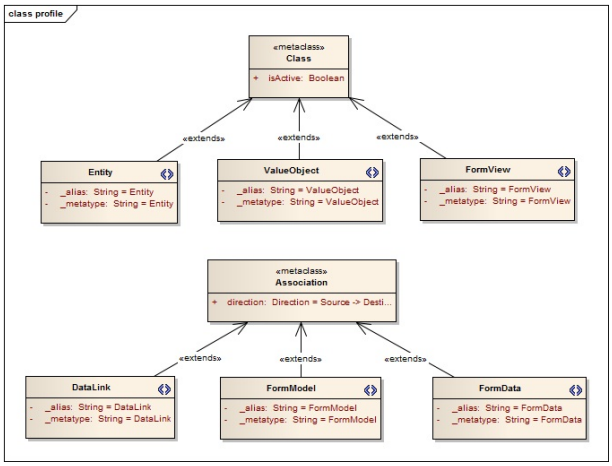



**Fig. 2.** The UML profile for developing a J2EE and Flex application.

Using the defined profile, the Dirigent tool is used to generate the source code artifacts for the application according to the profile stereotypes as follows:

- An entity class, a data access object class and a value object class in Java, a value object in ActionScript and an SQL creation script are generated for each Entity-stereotyped class.
- A value object in Java and a value object in ActionScript are generated for each ValueObject-stereotyped class.
- A form component in Flex and the form model in ActionScript are generated for each FormView-stereotyped class.

Associations with particular stereotypes used in the model are used to generate links and attributes of the generated source code artifacts as follows:

- The DataLink-stereotyped association is used to create a relationship between two classes. According to the multiplicities of both association ends,

a one-to-one, a one-to-many or a many-to-many relationship is created in the Java classes. Also, relationships in ValueObjects in Java and ActionScript are created. In the database, foreign keys are created for the association.
– The FormModel-stereotyped association is used to link a form with a class that define the structure of that form. Therefore, various form input elements are generated in the form component in Flex and a link to the value object class of the associated class is generated in the form model in ActionScript to store the form input data according to the association.
– The FormData-stereotyped association is used to link classes to a form to fill the form selection inputs such as combo-boxes, radio-button groups and check-box groups. Therefore, a list of links to instances of the value object class of the associated class is generated for each of such associations.

According to the defined transformations, we can easily create a model of an application consisting of UI forms using the defined stereotypes. According to the MDD approach, such a model can be processed by a tool such as Dirigent to generate source code classes with their attributes and links between various classes to save a lot of manual typing.

### 3.2 Application model creation

When a UML profile is defined, it can be used to create an application model. An example of the application based on the profile defined in 3.1 is shown in Fig. 3. It is a part of a university information system that allows to register and manage students of the university and subjects teached there. It also allows register students' results in various studied subjects.
Entities *Student*, *Subject* and *Classification* are created to store persistent data of the students, the subjects and the classifications of students in subjects, respectivelly. Also, forms are defined to create or update each of these entities. The *ClassificationForm* is defined to create or edit the student's classification. In this form, use chooses from a list of students to classify and a list of subjects to classify the selected student in. However, a value object *StudentWithGroup* instead of the entity is used for the student's selection where the student's name and group name are concatenated for the combo-box value label.

### 3.3 Rules definition

Because the transformations of the Dirigent tool are based on the stereotypes of model elements, the model must satisfy specific rules for the usage of the stereotypes to generate such artifacts and links effectively and correctly. These rules must be defined and obeyed during the modeling. According to the transformation patterns used in our example, the following rules can be defined:

1. Only an Entity-stereotyped class can be connected from an Entity-stereotyped class by a DataLink-stereotyped association.
2. Only an Entity-stereotyped or a ValueObject-stereotyped class can be connected from a FormView-stereotyped class by a FormModel-stereotyped association.
3. Only an Entity-stereotyped or a ValueObject-stereotyped class can be connected from a FormView-stereotyped class by a FormData-stereotyped association.
4. Exactly one class can be connected from a FormView-stereotyped class by a FormModel-stereotyped association.

The rule 1 is required to generate a valid relationship between existing Entity classes and database tables. Analogously, the rule 2 is required to generate valid references between the form components in Flex and ActionScript and the Entity or ValueObject classes that define the form structure and hold the input data. Similar rule 3 is required to generate valid references from the forms to the collections of data for the forms' choice elements such as combo-boxes. The rule 4 restricts the number of classes connected by the FormModel stereotype to a single class because only one class can define the structure of the form and keep the input data.

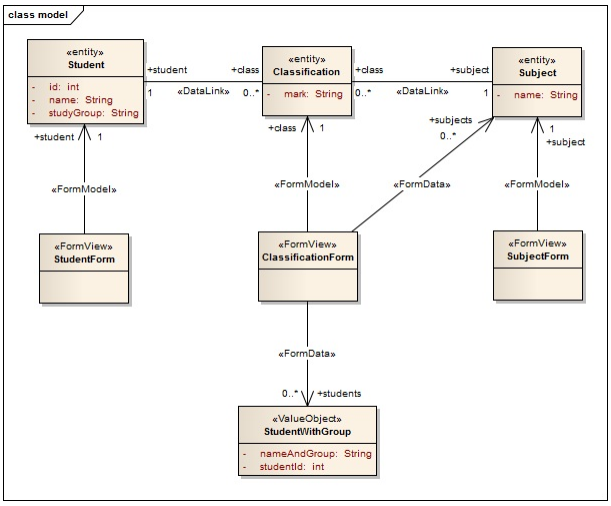



**Fig. 3.** A UML class model of an university information system created using our UML profile for developing a J2EE and Flex application.

Notice that the definition is always defined in the direction from the source class in the relation. This is to eliminate redundancy and to make each relation checked only once. If these rules are obeyed in the model, the transformation to the source code artifacts will be correct and no error shall appear.

### 3.4 Modeling rules

To generate source code artifacts effectively and correctly, the model have to be valid before the transformation and generation process is applied. Otherwise, the transformation process can fail or invalid source code artifacts can be generated containing invalid references or elements.

To validate the application model automatically, the rules must be defined in some formal way. We decided to use OCL invariants because there are tools that can be used for the validation such as OCLE [11] or DresdenOCL [16].

These constraints can be defined for each UML profile. However, we focus on a method to easily define the rules in a special constraint diagram using the well-known UML class diagram notation and generate the OCL invariants from that diagram. This way, we can easily define the rules for any possible domain or transformation tool.

We propose an approach of creating a special class diagram – we call it the *constraint diagram* – to model the rules using stereotypes used in the model and to generate the OCL constraints for model validation. The constraint diagram is defined in the following definition:

**Definition 1.** *The* constraint diagram *is a special diagram using the UML class diagram notation with classes and associations using domain-specific stereotypes used to define domain-specific rules for associations allowed between various stereotyped classes and to restrict the number of related target classes.*

The constraint diagram is used to restrict the stereotype of the target class of a particular stereotyped association from a particular stereotyped source class in the model and the number of target classes associated to a single source class by that association. In the constraint diagrams, we use the same notation as used in UML class diagrams because it is familiar to almost every developer and analyst. However, the semantics of the diagram differs from standard UML class diagrams of the UML model.

The constraint diagram for the rules of our UML profile for developing a J2EE and Flex application is shown in Fig. 4. Classes in the constraint diagram does not represent classes of objects existing in the modeled system. On the contrary, each of the classes in the constraint diagram represents any class with the same stereotype in the model. Therefore, the *AnyEntity* class in the constraint diagram in Fig. 4 represents the *Student* class as well as the *Subject* class or any other Entity-stereotyped class in the Fig. 3. The *AnotherEntity* class represents the same set of model classes because of the same stereotype.

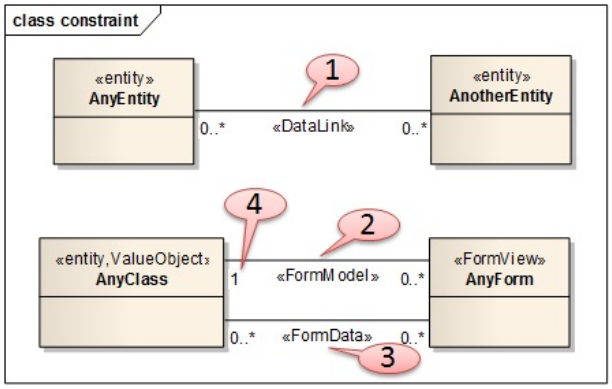



**Fig. 4.** The constraint diagram for our UML profile for developing a J2EE and Flex application.

If the class in the constraint diagram has more than one stereotype, it represents the set of all model classes with any of these stereotypes. Therefore, the *AnyClass* class in the constraint diagram in Fig. 4 represents all the Entity- and ValueObject-stereotyped classes in the model. If an association can be created between two classes with the same stereotype, two classes can be defined in the constraint diagram with the association connecting them, or a single class with a recursive association can be defined. We prefer the first option in our constraint diagrams.

Modeling an association between two classes in the constraint diagram represents the possibility of creating an association with the same stereotype between two classes with the same stereotypes as the source and target classes, respectively, of the association in the constraint diagram. Therefore, the FormModel-stereotyped association between the classes *AnyForm* and *AnyClass* in the constraint diagram in Fig. 4 means that there can be a FormModel-stereotyped association created between a FormView-stereotyped class and an Entity- or ValueObject-stereotyped class in the model. Examples of such associations in the model are the associations with the FormModel stereotype between the classes *StudentForm* and *Student*, and the classes *ClassificationForm* and *Classification*, respectively, in the model in Fig. 3.

```
context c:Class inv <SourceClassStereotype><RelationStereotype>Association:
let sts(c:Classifier) : Set(String) =
  c.stereotype->collect(name)->asSet()
let associationSet(c:Classifier, s:String) : Set(Association) =
  c.association.association.connection -> select(isNavigable = true)->
  select(a:Association | (s = "" and sts(a)->empty()) or sts(a)->includes(s))->asSet()
let associatedSet(c:Class) : Set(Class) =
  associationSet(c, <AssociationStereotype>)->collect(participant)->
  select(ac:Classifier | ac <> self)
sts(self)->includes(<SourceClassStereotype>) implies
  (associatedSet(self)->forAll(ac:Classifier | not(sts(ac)->excludesAll(<TargetClassStereotypes>)))
  and associatedSet(self)->size() >= <MinimalTargetMultiplicity>
  and associatedSet(self)->size() <= <MaximalTargetMultiplicity>)
```

**Fig. 5.** General form of OCL invariant generated from a constraint class diagram with wildcards for *SourceClassStereotype*, *TargetClassStereotype*, *AssociationStereotype*, *MinimalTargetMultiplicity* and *MaximalTargetMultiplicity*.

According to the rules defined in section 3.3, we have to express 4 rules in the constraint diagram – they are marked in the constraint diagram in Fig. 4. For the rule 1, we add two classes *AnyEntity* and *AnotherEntity* with the stereotype *Entity* to the constraint diagram. Then we add an association with the stereotype *DataLink* between the *AnyEntity* and the *AnotherEntity* classes. This association means that there can be a DataLink-stereotyped association between one entity – represented by the *AnyEntity* class – and another entity – represented by the *AnotherEntity* class.

Analogously, the rule 2 is expressed by the FormModel-stereotyped association between the classes *AnyForm* and *AnyClass* and the rule 3 is expressed by the FormData-stereotyped association between the same classes.

Furthermore, we can restrict the number of target classes connected to a single source class by an association with particular stereotype by defining the target multiplicity of the association in the constraint diagram. In the constraint diagram in Fig. 4, the target multiplicity of the FormModel-stereotyped association between the classes *AnyForm* and *AnyClass* is restricted to *1*. This stands for the rule 4. On the contrary, the FormData-stereotyped association has target multiplicity of *, therefore there is no restriction for the number of classes connected in the model.

Notice that the names of the classes and associations are not important in this diagram, they just represent any class with a particular stereotype and any relation of a particular type and stereotype in the model. Also notice that for correct rules definition and constraint generation, some rules for the constraint diagram modeling must be obeyed as well. However, we have to check the rules only for this single constraint diagram, while all the model diagrams based on the same profile with the rules defined in the constraint diagram can be validated and checked automatically by the generated constraints.

### 3.5 Generating OCL constraints

A set of OCL constraints can be generated from a constraint diagram using a tool such as Dirigent [4] to parse the model diagrams. The tool traverses the model and for each association in the diagram, an OCL invariant is generated. To explain the constraint structure, let us define several terms used in the constraint first.

**Definition 2.** *The* SourceClassStereotype *is the stereotype of the source class of the association in the constraint diagram just being processed by the generation tool. The* TargetClassStereotypes *is a collection of the stereotypes of the target class of that association. The* AssociationStereotype *is the stereotype of that association or an empty string if the association have no stereotype.*

**Definition 3.** *The* Minimal Target Multiplicity *is the lower multiplicity value of the target end of the association in the constraint diagram just being processed by the generation tool. The* Maximal Target Multiplicity *is the upper multiplicity value of the target end of that association.*

Both target multiplicity values are always extracted from the association. If only one value is defined for the target multiplicity, it is transformed to both values as usually, for example value *1* is equivalent to both minimal and maximal multiplicity values *1* and value * is equivalent to minimal multiplicity value *0* and maximal multiplicity value *.

The structure of such constraint is shown in Fig. 5. The invariant is defined in the context of Class with its name generated from the *Source Class Stereotype*, the

```
-- rule 1
context c:Class inv DataLink:
let associatedSet(c:Class) : Set(Class) =
  associationSet(c, "DataLink")->collect(participant)->
  select(ac:Classifier | ac <> self)
sts(self)->includes("Entity") implies
  (associatedSet(self)->forAll(ac:Classifier | not(sts(ac)->excludesAll("Entity"))))

-- rule 2 and 4
context c:Class inv FormModel:
let associatedSet(c:Class) : Set(Class) =
  associationSet(c, "FormModel")->collect(participant)->select(ac:Classifier | ac <> self)
sts(self)->includes("FormView") implies
  (associatedSet(self)->forAll(ac:Classifier | not(sts(ac)->
  excludesAll(Set {"Entity", "ValueObject"})))
  and associatedSet(self)->size() = 1)

-- rule 3
context c:Class inv FormData:
let associatedSet(c:Class) : Set(Class) =
  associationSet(c, "FormData")->collect(participant)->select(ac:Classifier | ac <> self)
sts(self)->includes("FormView") implies
  (associatedSet(self)->forAll(ac:Classifier | not(sts(ac)->
  excludesAll(Set {"Entity", "ValueObject"})))
```

**Fig. 6.** OCL invariants for the rules to check and validate the model.

*RelationStereotype* and the type of the relation. Then, three functions are defined – the function *sts()* returns a set of names of stereotypes of the classifier parameter – i.e. a class or an association –; the function *associationSet()* returns a set of associations from the given classifier that include the given stereotype according to the stereotype given as the parameter; the function *associatedSet()* returns a set of classes connected by the associationSet() except self – the class just being processed by the invariant. Then, the main constraint body is defined – if the class in context includes the *SourceClassStereotype* then target classes of all associations with the same stereotype as the *AssociationStereotype* must include at least one of the *TargetClassStereotypes* and the size of the associatedSet() must be between the *MinimalTargetMultiplicity* and *MaximalTargetMultiplicity* or equal to one of them. However, if the *MinimalTargetMultiplicity* is equal to *0* or the *MaximalTargetMultiplicity* is equal to *, the appropriate part of the constraint body may be omitted and if both multiplicities are the same value, the both multiplicity checking parts of the constraint body may be combined to a single one.

When parsing the constraint diagram shown in Fig. 4, the tool will find three associations to generate OCL invariants – DataLink-stereotyped association between two entities, a FormModel-stereotyped association leading from the *AnyForm* class to the *AnyClass* class and a FormData-stereotyped association from the *AnyForm* class to the *AnyClass* class. Therefore, three OCL invariants are generated as shown in Fig. 6. All generated invariants use the same functions *sts()* and *associationSet* as defined in Fig. 5.

### 3.6 Model validation

When a set of OCL constraints is generated for the rules for the usage of UML stereotypes in the model, the application model can be validated. A tool such as DresdenOCL can be used to validate the model using the OCL invariants. The validation process will identify violated rules and the context in which they are violated, pointing out taht the class is connected to some other class using wrong-stereotyped association, the target class have wrong stereotype attached or the number of connected classes is invalid, respectivelly. With this information, the designer or developer can modify the application model so it satisfies the OCL invariants and therefore also satisfies the rules.

If no violations are detected, the model is correct according to the defined rules and the transformation process can by applied to generate the source code of the application.

## 4 Conclusions

Model-Driven Development approaches for software development became popular in last years. Many soft-

ware development processes use a tool to transform models and to generate source code artifacts. Many domain-specific UML profiles are also created for various domains or software projects and generation tools are adapted to utilize these profiles during transformation and generation. However, this approach brings the need of domain rules definition of how the profile should be used.

In this paper, we presented an approach of modeling these domain rules for the use of user-defined stereotypes and relations between each other using a special constraint diagram based on the well-known UML class diagram notation. We presented a method how such diagram can be created for a set of example rules. We also presented a technique how to generate OCL invariants from the constraint diagram to be used for the model validation. Whole process was illustrated on an example of a J2EE application with the user interface written in Flex and ActionScript.

In our further research, we would like to extend our approach to enable to model directed associations constraints to validate usage of directed associations in the model, other types of relations such as generalizations or dependencies. Finally, extending the approach by other types of model artifacts such as actors, components or use cases can be explored.

## References


1. OMG, J. Miller, J. Mukerji: *MDA guide version 1.0.1.* http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf, June 2003.
2. OMG UML 2.4, Aug. 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4/
3. J. Arlow, I. Neustadt: *UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition).* Addison-Wesley Professional, 2005.
4. K. Hubl: *Dirigent.* Jan. 2012. [Online]. Available: code.google.com/p/dirigent/
5. OMG, Object constraint language, version 1.3, http://www.omg.org/spec/OCL/2.2/PDF, Feb. 2010.
6. Z. Rybola, K. Richta: *Using OCL in model validation according to stereotypes.* In: DATESO 2012, Zernov, Rovensko pod Troskami, Czech Republic, Apr. 2012, 93–102.
7. M. Richters, M. Gogolla: *Validating UML models and OCL constraints.* UML 2000 – The Unified Modeling Language, Proceedings - Advancing The, 1939, 2000, 265–277.
8. M. Richters, F. Buettner, F. Gutsche, M. Kuhlmann: *USE – a UML-based specification environment* http://www.db.informatik.uni-bremen.de/projects/USE/, Jan. 2011.
9. M. Gogolla, J. Bohling, M. Richters: *Validating UML and OCL models in USE by automatic snapshot generation.* Software and Systems Modeling 4(4), 2005, 386–398.
10. H. S. Chae, K. Yeom, T. Y. Kim: *Specifying and validating structural constraints of analysis class models using OCL.* Information and Software Technology 50(5), Apr. 2008, 436–448. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0B-4NVH7T5-1/2/02217cd36c68c34c93fc63253c28bf62
11. Chiorean: *OCLE 2.0 - object constraint language environment.* Feb. 2012. [Online]. Available: http://lci.cs.ubbcluj.ro/ocle/index.htm
12. G. Guizzardi: *Ontological foundations for structural conceptual models.* PhD Thesis, University of Twente, Enschede, The Netherlands, Oct. 2005. [Online]. Available: http://eprints.eemcs.utwente.nl/7146/
13. A. B. Benevides: *A model-based graphical editor for supporting the creation, verification and validation of OntoUML conceptual models.* Ph.D. Dissertation, Federal University of Espírito Santo (UFES), Vitória, E.S., Brazil, Feb. 2010.
14. Sparx Systems: *Enterprise architect – UML design tools and UML CASE tools for software development.* Mar. 2011. [Online]. Available: http://www.sparxsystems.com.au/products/ea/index.html
15. The Apache Software Foundation: *The apache velocity project.* Nov. 2010. [Online]. Available: http://velocity.apache.org/
16. B. Demuth: *DresdenOCL.* http://www.reuseware.org/index.php/DresdenOCL, Jan. 2011.
17. Adobe Systems Incorporated: *Adobe flex.* May 2012. [Online]. Available: http://www.adobe.com/products/flex.html

**ITAT'12**

**Informačné Technológie – Aplikácie a Teória**

# WORK IN PROGRESS

# New language for searching Java code snippets*

Tomáš Bublík[1] and Miroslav Virius[1]

Czech Technical University, Faculty of Nuclear Sciences and Physical Engineering, Trojanova 13, 120 00 Praha 2
WWW home page: http://www.fjfi.cvut.cz

**Abstract.** *This paper introduces the new scripting language which is used for searching the code snippets in a Java source code. This language, named Scripthon, is a compiled language outputting an abstract syntax tree. Next, this tree is compared with the Java abstract syntax tree. The Compiler Tree API is used in the searching algorithm. Both the language specification and searching are described in this paper.*

## 1 Introduction

The Scripton language is a response on a demand for a tool for the Java source code description and addressing. A purpose of the language is to provide a tool for finding code snippets which are needed for further processing.

Scripthon was designed to be very easy to understand and easy to read. In Scripthon, the code description is much quicker than finding wanted sections in a source code. Because it is possible to describe the Java structures by this language, its keywords are based on the Java structures. The keywords, as well as the statements, are the Java structures names. The Scripthon has a simple and modern syntax which is similar to the syntax of contemporary programming languages. Syntactic elements of the Scripthon are similar to analogical elements of the Java language. Even the layout and nesting of the statements is the same as in Java. A short example follows. Consider the following piece of the Java code:

```
public class TestDecompile {
   public static void main(String [] args) {
       String toPrintValue = "Hello world!";
       System.out.println(toPrintValue);
   }
}
```

This piece of code can be described by the following Scripthon script:

```
Class(Name="TestDecompile";Rest=public)
   Meth(Name="main";Ret=void;Rest=public)
       Init(Name="toPrintValue";Type=String)
       MethCall(Name="System.out.println")
```

* This work is supported by the SGS 11/167 grant.

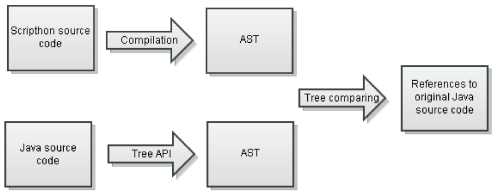



**Fig. 1.** Scripthon action flow

## 2 How it works

A script, written in the Scripthon language, corresponds to a required/desired piece of code in Java. Next, this script is compiled. The output of the compiler is an abstract syntax tree, hereinafter AST. Further, the AST is compared with the AST of Java code which was created by the Java Compiler API. This API is a part of the Java SDK distribution.

Scripthon is a language which offers a variable degree of Java source code description. This means that it can describe both the very precise and the very vague code snippet. The previous example can be also described by this script:

```
Class()
     Meth()
        Any()
```

It is obvious that the degree of detail will often vary depending on the search results. If there are few results (or no result at all), we have to decrease the degree of detail. If there are too many results, the degree of detail detail is to be increased and the results should be refined. Summing up, Scripthon allows a very detailed, but also a very free description of an intended structure. The detailed description refers to the source code level description, while the free description reaches up to the class blocks.

An intended code snippet is found by comparison of its compiled tree and the corresponding abstract syntax tree. Unlike in the case of the Java language, there is no need of the translation it into the bytecode. However, the files describing an AST are generated for a repeated searching. The Scripthon compilation process is fast enough to regenerate the files again in reasonable time.

The final output of the compilation is a list of references to the relevant parts of the source text. If any snippet does not match, the list is empty. Because the amount of the results can be disproportionately large, the compiler also detects scripts that are too general.

## 3 Language specification

Scripthon is a dynamically typed, interpreted, and non-procedural language. Its translation into a tree-expressing form and usage is very similar to the usage of other modern script languages. The language was formed on the basis of the references [1–8].Because the language is designed to be a scripting language only, there are no special constructions starting a script. Neither is this language pure object-oriented. An input for a compiler is a text file with a sequence of commands. This sequence describes consecutive statements in a Java source code. The commands with a variable detail degree correspond to a variable length code segment. The detail level is not fixed and can vary in every command. One command can correspond to a line of a source code; other one can describe the whole class in Java.

The Scripthon structure is very close to other contemporary dynamic programming languages. The individual commands are separated by lines. There is no command separator in Scripthon. Inner parts of blocks are tab nested. A block is not delimited by any signs; just a hierarchy of tabulators is used. An example:

```
Block() a
   Loop(Type=while)
```

### 3.1 Denotational semantics

The complete definition of the Scripthon language semantics is beyond the scope of this paper. Therefore, only the commented building blocks of the denotational semantics and syntax will be given.
Syntactic domains:

$$
\begin{aligned}
n &: Num\,(\,numerals\,)\\
V^+ &: Lang\,(\,language\,)\\
x &: variable\,names\\
a &: Aexp\,(\,arithmetic\,expressions\,)\\
b &: Bexp\,(\,boolean\,expressions\,)\\
Str &: structures\\
SAtr &: structure\,attributes\\
AVal &: attribute\,values\\
S &: statements\\
D &: declarations
\end{aligned}
$$

Sets definition:

$$
\begin{aligned}
digit &::= 0\,|\,1\,|\ldots|\,8\,|\,9\\
numeral &::= < digit > \,|\, < digit >< numeral >\\
integers &::= \mathbb{Z} = \{\ldots,-2,-1,0,1,2,\ldots\}\\
V &= \{A,B,\ldots,Z,a,b,\ldots,z\}\\
\mathbb{N} &= \{1,2,3,4,\ldots\}\\
W &= \{a^n\}_{i=1}^{n}\,\forall a_i \epsilon V\\
V^* &= \{w_j\}_{j=1}^{m}\\
e &\ldots empty\,word\\
V^+ &= V^*\backslash\{e\}\\
x &\;\epsilon\; V^+\backslash\{Str\}\backslash\{SAtr\}\backslash\{Aval\}\backslash n\\
\mathbb{B} &= \{true,false\}\\
Str &= \{Meth,Init,Block,Class,\ldots\}\\
SAtr &= \{Name,Lenght,Rest,Val,\ldots\}\\
AVal &= \{public,private,\ldots\}\\
AVal &\subseteq V^*\\
AVal &\subseteq Num
\end{aligned}
$$

The first few sets are similar to usual definitions. Therefore, a comment is not needed. The interesting domain is “Str”. This domain specifies the structural keywords. It is the set of words, starting with a capital letter, which corresponds to every source code structure in Java. For example, Meth, Init, Block, Stmt correspond to a method, a variable initialization, a block of code, and a statement in the same order. The next domain “SAtr” contains the structure attributes. The Java code structure can have a lot of attributes. For example, an attribute, or a method can have a scope, a variable can have a name and type, and so on. Therefore, examples of the set “SAtr” could be: Scope, ParamType, Type, ... The last interesting set is AVal. This set contains the values of the structure attributes. The typical values for a scope are “public”, “private” and “protected”. On the other side, this set can contain even other text values. Most often, these values are the names of the methods, variables and its values. The usage in a program can look like this:

```
Init(Name=increment)
```

It means that a variable is initialized (Str) with an attribute “Name” (SAtr), and the value of the attribute is “increment” (AVal). All three sets are linked with a dependency graph. This graph determines which specific structures are able to be used. Moreover, it determines the structure attributes and its values. While compiling, the source code is checked if it complies with requirements of the dependency graph. However, in fact, the dependency graph is much larger. Figure 2 shows a small example.

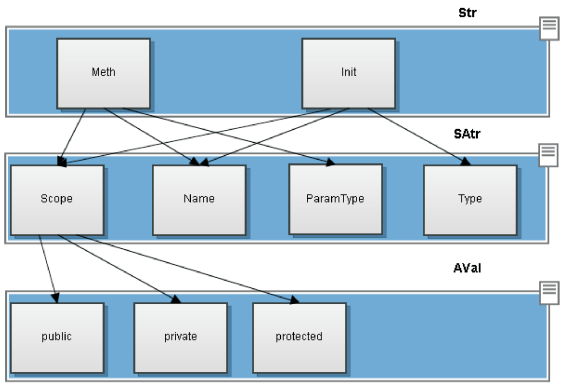



**Fig. 2.** The semantic sets dependency graph

### 3.2 Syntax

$$
\begin{aligned}
i, j \epsilon \mathbb{N} \\
x :: Str_i \\
AVal ::= n | x \\
a ::= true | false | AVal_1 == AVal_2 | \\
| AVal_1 ! = AVal_2 | !b | b_1 \,\&\&\, b_2 | b_1 \,||\, b_2 | \\
| a_1 == a_2 | a_1 ! = a_2 | \\
| a_1 \geq a_2 | a_1 > a_2 | a_1 \leq a_2 | a_1 < a_2 \\
D ::= Str_i() \, x \\
S ::= Str_i() | D | X.SAtr_i = AVal_j | skip | S_1 < cr > S_2 | \\
\S_1 < cr >< tab > S_2 | if \, b \, then \, S | \\
if \, b \, then \, S_1 \, < cr > \, else \, S_2 \, | \\
| while \, b \, do \, S | Str_i(SAtr_1 = AVal_1; SAtr_2 = AVal_2; ...)
\end{aligned}
$$

$$
\begin{aligned}
\Sigma \ldots set \, of \, all \, states \\
A : Aexp \rightarrow (\Sigma \rightarrow \mathbb{Z}) \\
B : Bexp \rightarrow (\Sigma \rightarrow \mathbb{B}_\perp) \\
S : Statement \rightarrow (\Sigma \rightarrow \Sigma_\perp) \\
[S] : \Sigma \rightsquigarrow \Sigma (partial \, function)
\end{aligned}
$$

According to the syntax, it is obvious that all the operations and expressions are the same like, or at least similar to, corresponding ones in Java. Scripthon has only one kind of a loop; the *while* loop. A state is defined by the variables stored in a stack, and by an abstract syntax tree which is created during compilation process. To simplify the definition of transition between states, the partial function is used.

### 3.3 Notes on the syntax

Scripthon distinguishes the uppercase and lowercase letters. Despite of the above described differences, the Scripthon syntax is very similar to the syntax of the Java language. Thus, the Java programmers do not have to learn the new rules and syntax. Except for a few differences, there are very similar operators and as in Java. The basics of the Scripthon syntax will be briefly introduced in the following paragraphs.

### 3.4 Description of the program structures

It is possible to describe every program element in Java program with Scripthon. The description is based on the so called structural keywords. A structural keyword starts with a capital letter, and ends with a parenthesis. Each structural keyword corresponds to an element in the Java language. Parameters are defined inside the parentheses which specify an element closer. It is not necessary to use the parameters; the level of description abstraction is higher without them. The names of structural keywords are the shortened names of the elements in Java. Some basic structural keywords will be mentioned here; the scope of this paper does not allow to include all of them.

The structural keywords correspond to the Java structures, the parameters are the shortcuts of their properties. The parameters are separated by a semicolon, and the value is assigned using the = operator. For example, the following series of keywords: `Block()`, `Loop()`, `Class()` indicates (in the same order): a block of statements, a loop, and a class. For a closer determination, the parameters can be used. For example, this command: `Loop(Type=for;Lines=4)` indicates a *for* loop with 4 lines.

### 3.5 Variables

Variables in the Scripthon language are initialized in a very similar way to Java. A variable can be declared only by its name and type, but it is possible to give variable a value, too. There are two kinds of variables in Scripthon: the basic ones and the structural ones. Basic types of variables are either textual or integer variables. In the case of a basic type variable, the = sign is used to assign it a value.

If no value is given, the value of a textual variable is the empty string (`""`), or 0 in the case of an integer variable. The types of basic variables are detected automatically. A compiler detects whether a variable is textual of integer one.

In the case of a structural variable, if no value is given, and the structure is not further specified, this variable is initialized in the most common form. Structural variables can have parameters in parentheses;

however, these variables can stand alone. An example follows:

```
Meth(ParamType=Long) method
method.ParamType=String
a = method.ParamType
```

It is obvious that the variable `a` has a String value.

### 3.6 Functional and blank lines

Blank lines are inadmissible in the middle of the block of code, and the compiler reports an error in such a case. Empty lines only separate the blocks. In that case, their number is not important and they act as one line. Similarly, the same rule holds at the end and at the beginning of each script. In contrast, the functional lines are those that are not empty, or are not a part of the comments. They have a logical functionality. Here is a difference between Scripthon and other languages where the blank lines mean nothing.

## 4 Compiler

The compiler of this language is written in Java, and it consists of the parser, the tokenizer, the state machine, and the lexical analyzer. The compilation process is similar to any other compilation. Thus, it consists of several steps: First, during the lexical analysis, the characters of source code are transformed into tokens. In order to determine, where the program starts, where the blocks start, or where it ends, other control tokens are are added. After this, a grammar and syntax is checked by the state machine. Finally, the semantics is checked and a tree describing the searched structure is generated. While building the AST, the structure is checked against the dependency graph described above. A visitor design pattern was used in the compiler. It allows easier adding the new structures into the language. In conclusion, the result is an AST which consists of interlinked objects representing the different structures of the language.

## 5 Abstract syntax tree

An abstract syntax tree is used to search the required parts of a source code. In this case, it is a tree which is quite similar to a classical abstract syntax tree (AST), except several parts. The AST is described in [11] and [12]. The main difference between these trees is in two special types of nodes arising from the Scripthon translation. These two special node types are `Any` and `Condition`. These types do not describe any data type,


**Fig. 3.** Compiler flow. Source: [10]

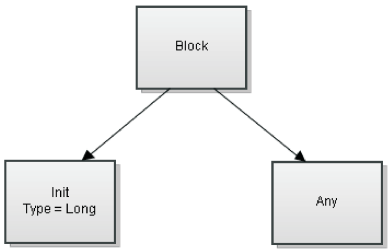



**Fig. 4.** Simple example of tree with Any node

block, or other element. Furthermore, they have a special list of features and rules, and they also behave differently in comparison with other trees.

The node type `Any` tells the comparing algorithm that this node corresponds to any other node of a searched tree.

```
Block()
    Init(Type=Long)
    Any()
```

If used together with a property `Subtrees=true`, the node type `Any` corresponds not only to any other node, but to the whole sub tree with the searched node as a root. However, the only node (element) is assumed by default.

A slightly more complicated situation occurs when using the node type `Condition`. While comparing, this type indicates that it is necessary to take into account other conditions, or to follow some special rules. These rules arise from a Scripthon source script. The comparison will be then enriched by considering the rules. The rules and restrictions are used according to the optimistic strategy. This means that when they are satisfied, the rest of the node acts like a type `Any`. For example, if the node type `Condition` with condition `ParamsNum=4` representing a method is compared with another method node, only those methods with four parameters will match. If nothing else will be mentioned (e.g. method name, visibility), and all the above conditions will be met, any method matching the conditions will be considered adequate.

## 6 Searching

The main reason why the language is developed is to find the specified Java source code structures. Searching is realized by a gradual comparing of two abstract syntax trees. The first tree is assembled by the compiler from the Scripthon source code. The second one is obtained by the Java Compiler API. This API is a result of the JSR (Java Specification Request) 199, and it has got the tools for controlling a Java compilation process. One of the important parts of this API is the Compiler Tree API. The Java source codes are possible to get using the Tree API in the form of an AST. The requested structures can be found by using the well known, and many times described, algorithms for trees comparing [11]. The results of comparing the trees are the references to the original lines of given Java source code. If nothing was found, the references are empty.

## 7 Language usage

There are several fields where Scripthon can be applied.

One area of Scripthon usage is the detection of code clones in the Java source code. According to [9], the machine-only made clone detection methods are not without complications in the case of non-ideal clones. An empirical element in the form of a human help could bring an improvement. A user (programmer) can help the detection by defining an expected duplicity with a simple code description, but he or she does not know the precise form of the code snippet; he or she can give only a rough description. The typical scenario is that a user, working on a project, notices similarities in a code. A lot of the so-called "non-ideal" clones occurred in a project; however, it is difficult to find them. He (or she) tries to use clone detection software but it fails; no one of the code snippets can be considered an exact clone.

Another usage of the Scripthon language is the teaching of the Java programming. The language is appropriate especially for the teachers. He or she can check the student's homework with a special condition of using some specific Java constructions or structures very fast. He or she can have prepared the Scripthon scripts corresponding to this construction. Finding these constructions is very easy then. Further, if he or she knows about the usual student's mistakes, he or she can be prepared for it with the scripts in advance. Scripthon can also check, whether the students rewritten the work from each other. The usual practice is to change a variable or method name. This easy example shows up the different source code but with the same results:

```
doSomething(Long v1,int b) {
    ...
}
newMethod(int a,Long v2) {
    ...
}
```

This code can be discovered by following script:

```
Meth(ParamType=Long;ParamType=int;Order=no)
```

This line of code in Scripthon will discover both the code snippets written in Java. The three parameters determine the method types, while the last one says that it is not order depending.

Unfortunately, one of the disadvantages of Scripthon is that it allows writing, for example, so general script that the whole source code could be a result. Furthermore, a user could create too large and general script that the searching will be inefficient.

## 8 Conclusion: Current state and the scheduled work

Currently, the compiler has been finished. Some fine features are scheduled to add. For example, the compiler should be able to detect the basic types of performance problems caused by an unnecessary searching, or infinite loops. Moreover, many optimizations can be applied in the algorithms for a corresponding code finding. The work on searching the trees is still running.

The complete deployment including the optimization, searching, and a preparation of a user interface is scheduled for the summer 2012.

## References


1. S. Krishnamurthi: *Programming languages: application and interpretation.* Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License Version 2007-04-26.
2. B. C. Pierce: *Types and programming languages.* The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142 `http://mitpress.mit.edu` ISBN 0-262-16209-1.
3. D. Grune, C. J. H. Jacobs: *Parsing techniques – second edition.* VU University Amsterdam, Amsterdam, The Netherlands Published (2008) by Springer US, ISBN 978-1-4419-1901-4.
4. M. Steedman: *The syntactic process.* MIT Press, 2000.
5. P. Selinger: *Lecture notes on the Lambda calculus.* MIT Press, 2000.
6. K. T. Kalleberg, E. Visser: *Fusing a transformation language with an open compiler.* Report TUD-SERG-2007-025, ISSN 1872-5392.
7. J. R. Cordy: *TXL – a language for programming language tools and applications.* ENTCS, 110, 2004, 3–31.
8. A. Chlipala: *A certifed type-preserving compiler from Lambda calculus to assembly language.* PLDI 2007: 54-65, University of California, Berkeley, 2007.
9. T. Bublík, M. Virius: *Automatic detecting and removing clones in Java source code.* In: Software Development 2011. Proc. of the 37$^{\text{th}}$ National Conference Software Development. Ostrava, May 25 – 27 2011. Ostrava: Technical University of Ostrava 2011., 10–18, ISBN 978-80-248-2425-3.
10. A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman: *Compilers: principles, techniques, and tools*(2nd Edition). Addison-Wesley, 1986. ISBN 0-201-10088-6
11. I. D. Baxter, A. Yahin: *Clone detection using abstract syntax trees.* Proc. IEEE Int'l Conf. on Software Maintenance (ICSM) '98, Bethesda, Maryland, Nov. 1998, 368–377.
12. N. Juillerat, B. Hirsbrunner: *An algorithm for detecting and removing clones in Java code.* Proc. of 3$^{\text{rd}}$ Workshop on Software Evolution through Transformations, Natal, Brazil, Sep. 2006, 63–74.

# D-Bobox: O distribuovatelnosti Boboxu*

Miroslav Čermák, Zbyněk Falt, and Filip Zavoral

Katedra softwarového inženýrství, Matematicko-fyzikální fakulta Univerzita Karlova v Praze
Malostranské nám. 25, 118 00 Prague, Česká republika
{cermak,falt,zavoral}@ksi.mff.cuni.cz

**Abstrakt** *Súčasné IT technológie a systémy generujú obrovské množstvo dát, či už vo forme rôznych záznamov behu systémov, alebo užívateľských dát, ktorých spracovanie, analýza a ukladanie začína byť problematické pre tradičné databázové systémy. Využitie lokálneho paralelizmu na spracovanie týchto dát naráža na jednej strane na hranice paralelizovatelnosti úloh, na druhej strane na finančnú náročnosť špecializovaného hardvéru. Nárast kvality počítačových sietí a dostupnosť relatívne výkonného bežného hardvéru prispieva k rastu popularity rozličných distribuovaných riešení pre prácu s veľkými dátami.*
*Systém Bobox je framework určený k paralelnému spracovaniu veľkého množstva dát. Návrh založený na rozdelení úlohy na samostatné menšie podúlohy, ktoré komunikujú pomocou správ, nabáda k jeho rozšíreniu na distribuovaný systém. Článok analyzuje možnosti a problémy rozšírenia a na ich základe definuje požiadavky na Bobox a jeho distribuovanú nadstavbu D-Bobox s prihliadnutím na schopnosť automatického generovania plánov pre distrtibuované spracovanie.*

## 1 Úvod

Rastúce množstvo súčasných systémov začína produkovať ohromné množstva dát. Môže sa jednať o záznamy činnosti behu rôznych systémov, alebo užívateľské dáta internetových aplikácii ako napríklad sociálne siete. Tieto dáta sú zdrojom užitočných informácií, ktoré je možné získavať rôznymi zložitejšími analýzami, alebo sú to dáta neštruktúrované, či bez pevnej schémy, ktoré sa bez predchádzajúcej prípravy nehodia pre spracovanie v klasických RDBMS systémoch a warehouse riešenia vyžadujú finančne náročné softwarové a najmä hardwarové riešenia. Zároveň rastie dostupnosť výkonného obyčajného hardvéru, ktorý môže byť vyčlenený primárne na spracovanie úloh, či vo forme dostupných pracovných staníc, ktoré bežia 24 hodín denne, avšak ich výkon je využitý iba minimálne. To je hnacím motorom výskumu vysoko škálovateľných paralelných a distribuovaných systémov schopných využiť túto dostupnú výpočtovú silu k spracovaniu (dátovo) náročných úloh.

Systém Bobox [1] je framework určený k vývoju aplikácií pre spracovanie veľkých dát v paralelnom prostredí. Bobox má dva primárne ciele: zjednodušiť písanie paralelných, dátovo náročných programov a slúžiť ako testovací základ pri vývoji paralelných algoritmov. Bobox je schopný dosiahnuť vysoký stupeň paralelizmu na jednom počítači [2], avšak ako lokálny systém nedokáže využiť potenciál spracovania úloh na viacerých počítačoch. Preto je ďalším logickým krokom jeho rozšírenie na D-Bobox pridaním podpory pre distribuované spracovanie škálovateľné od jedného uzlu, či malej skupiny uzlov, až po tisícky uzlov. Článok popisuje možnosti rozšírenia Boboxu a ukážeme, že takto rozšírený systém bude schopný poskytnúť podporu pre širšiu skupinu úloh než súčasné populárne systémy.

V práci sa nevenujeme problematike uloženia a lokality dát a prístupu k nim, ktorá závisí na konkrétne riešenom probléme. Obecne môžeme predpokladať, že dáta sú umiestnene na distribuovanom súborovom systéme resp. v distribuovanej databáze a rozumne dostupné na každom výpočtovom uzle. Ďalej predpokladáme nasadenie v jednej administratívnej doméne s výkonnou a spoľahlivou technológiou a známou sieťovou topológiou s centrálnou správou. To nám umožňuje zamerať sa na návrh s ohľadom na škálovatelnosť a prívetivosť frameworku, bez nutnosti riešenia ťažkých problémov distribúcie súvisiacich s komunikáciu po nespoľahlivých sieťach. Na záver načrtneme možný prínos na príklade systému pre spracovanie SPARQL požiadaviek.

### 1.1 Schéma Boboxu

Základnou myšlienkou za systémom Bobox je rozdelenie úlohy na nezávislé podúlohy, ktoré môžu byť usporiadane do (nelineárnej) pipeline [3]. Beh jednotlivých komponent, nazývaných krabičky, ktoré reprezentujú podúlohy je riadený dostupnosťou dát na ich vstupoch. Krabičky si predávajú dáta pomocou správ, ktoré prijímajú na svojich vstupoch a nové správy predávajú na svoje výstupy. Predávanie správ medzi výstupmi a vstupmi krabičiek prebieha automaticky v jadre Boboxu, krabičky nemusia riešiť problémy typické pre programovanie v paralelnom prostredí ako synchronizácia, plánovanie, alebo konkurentný prístup.

* Článok bol podporovaný Grantovou agenturou Univerzity Karlovy, projekty č. 277911 a č. 28910, grantom SVV-2011-263312, a projektom GAČR č. 202/10/0761.

Aby bolo možné úlohu konkrétneho problému riešiť paralelne, je nutné pripraviť jej plán umožnujúci takéto spracovanie. Zložitejšie problémy vyžadujú špecifické plány pre jednotlivé úlohy a ich automatické generovanie uľahčuje prácu s výsledným systémom. Zodpovednosť automatického generovania prináleží komponente frontend, ktorý je špecifický pre daný problém. Jednotlivé frontendy môžu navyše obsahovať optimalizačné algoritmy pomahajúce budovať efektívne plány. Príkladom kompexnejšieho použitia môže byť použitie Boboxu ako systému pre spracovanie SPARQL požiadavok [4].

### 1.2 Súvisiace práce

V súčasnosti je veľmi populárnym nástrojom na paralelné a distribuované spracovanie veľkých dát na komoditnom hardvéri open-source framework Hadoop [5], založený na MapReduce princípe navrhnutom a hojne využívanom spoločnosťou Google [6]. Avšak MapReduce ma obmedzenia, ktoré znižujú jeho flexibilitu a obmedzujú ho iba na úlohy na ktorých spracovanie sa dá aplikovať jednoduchá schéma rozdelenia úlohy (map) a redukcia výsledkov (reduce), pričom sú podporované hlavne agregačné funkcie ako napr. SUM, AVG ... Ďalším obmedzením je vykonávanie fázy redukcie dát až po ukončení fázy map. Naproti tomu Bobox umožňuje prúdové spracovanie dát, priebežným posielaním výsledkov medzi výpočtami tak, ako sú generované a umožňuje prácu s komplikovanejšími plánmi než Hadoop. Príkladom takých plánov môže byť kompilácia zdrojových kódov, ktorá obsahuje závislosti a neumožňuje nasadiť jeden veľký paralelizačný MapReduce krok [7].

V práci [7] autori experimentálne poukázali na to, že súčasný návrh a implementácia MapReduce frameworku neposkytuje dostatočnú flexibilitu a efektívnosť v prípade programov s mnohými paralelizovatelnými krokmi namiesto jedného obrovského paralelizovatelného kroku a prípadne taktiež netriviálnou logikou. Zároveň navrhli odľahčený framework, ktorý sa snaží odstrániť tieto obmedzenia a poskytnúť nízku latenciu pre obecnejšiu a flexibilnejšiu schopnosť paralelného spracovania v prostredí cloud computingu.

Podobným systémom ako D-Bobox je Dryad [8], postavený na podobných základoch rozdelenia úlohy na jednotlivé podúlohy a vytvorení plánu v podobe orientovaného acyklického grafu reprezentujúceho výpočet. Namiesto komunikácie pomocou správ podporuje rôzne spôsoby komunikácie pomocou TCP streamov, zdieľanej pamäte a súborov. Najzásadnejším rozdielom je, že neumožňuje automatické generovanie plánov a pre každú úlohu musí byť definovaná kostra plánu ručne. Bobox k tomuto účelu využíva špecializované frontendy [9] [3].

## 2 Problémy

Snaha využiť potenciál clustrov dostupných serverov, či bežných počítačov stojí za rozšírením Boboxu na D-Bobox. Pridanie distribúcie umožní väčšie škálovanie, avšak vyžaduje zváženie rôznych problémov, ktoré z tohto vyplývajú. Podstatnejšie z nich sú popísané v nasledujúcich kapitolách spolu s ich dopadmi, resp. požiadavkami na systém D-Bobox.

### 2.1 Riadenie behu

Riadenie behu v Boboxe má na zodpovednosti runtime časť v backende, konkrétne plánovač ktorý určuje ktoré krabičky majú byť naplánované vzhľadom na dostupnosť zdrojov, dát a ďalšie okolnosti [3]. D-Bobox musí byť rozšírený o logiku obsluhujúcu záležitosti ohľadom distribúcie a táto môže byť umiestnená buď v krabičkách alebo v jadre, pričom každé z týchto umiestnení kladie odlišné požiadavky a vyplývajú z neho iné dôsledky na D-Bobox.

Umiestnenie riadiacej logiky do krabičiek je z pohľadu štruktúry systému prenesenie problému na užívateľskú časť. To minimalizuje požiadavky na jádro D-Boboxu, ktoré nemusí tento problém priamo riešiť. Avšak plánovač si musí byť vedomý prítomnosti takýchto krabičiek v užívateľskej časti a prispôsobiť im plánovanie, pretože riadiace krabičky musia byť vysoko dostupné pre spracovanie komunikácie so vzdialenými uzlami. Podstatným následkom neschopnosti riadenia distribúcie z jadra je nutnosť vykonať všetok distribuovaný výpočet v rámci jednej krabičky. Nevýhodou tohto riešenia je, že prenesenie distribúcie na užívateľskú časť ide proti základnému princípu, ktorým je uľahčiť vývoj paralelných a distribuovaných aplikácií.

Druhou možnosťou je umiestnenie riadiacej logiky do jadra D-Boboxu, kde by tvorila nezávislú vrstvu. Táto vrstva by mala na starosti zadávanie úloh podriadeným uzlom, riadenie distribuovanej komunikácie (vzdialené posielanie správ) a finálny zber dát. Týmto sa podstatná časť problémov distribúcie skryje pred užívateľom, avšak nie je ju možné skryť úplne. V prípade potreby rozdelenia a redukcie dát poskytne D-Bobox implementáciu obecných krabičiek pre triviálne operácie rozdelenia, či redukcie dát. Pre špecifické úlohy bude môcť užívateľ poskytnúť vlastnú implementáciu potrebných operácií, ktoré sa použijú pri distribúcii na úrovni dát (viď kapitola 2.2). Implementácia frontendu si taktiež musí byť vedomá distribuovaného spracovania a generovať plán s pomocnými informáciami pre distribučnú riadiacu časť pomáhajúcimi pri delení plánu medzi jednotlivé uzly.

Integrácia do jadra D-Boboxu je preferovaným riešením pretože je jednak v súlade s ideou

(D-)Boboxu a umiestnením do samostatnej vrstvy vytvára prostredníka medzi frontendom a backendom, čím zachová zameranie backendu na lokálny paralelizmus a umožní jeho využitie s minimálnymi zmenami.

### 2.2 Úroveň distribúcie

Na distribúciu môžeme nahliadať z dvoch rôznych pohľadov a to: distribúcia na výpočtovej úrovni a distribúcia na dátovej úrovni. Distribúcia na dátovej úrovni požaduje, aby bolo možné vstupné dáta úlohy rozdeliť na menšie (nie nutne disjunktné) časti, ktoré je možné spracovávať paralelne. V prípade Boboxu nesmú byť vytvárané plány, ktoré vytvárajú závislosti medzi vstupnými dátami, ktoré neumožnia ich rozdelenie. Logiku pôvodných frontendov postačuje rozšíriť o schopnosť obalenia výpočtu o map a reduce logiku (pridaním vhodných map/reduce krabičiek) dostávame tradičnú MapReduce schému, avšak dáta medzi map a reduce fázou nie sú limitovane na key-value formát a na jednotlivé časti dát môže byť aplikovaný netriviálny algoritmus využívajúci paralelizmus pre zvýšenie výkonu výpočtu na jednom výpočtovom uzle.

Distribúcia na výpočtovej úrovni naproti tomu predpokladá rozdelenie plánu na dielčie časti, ktoré je možné vykonávať paralelne. Pre efektivitu paralelného spracovania sú dva základne predpoklady: nelineárny plán spracovania a prúdové spracovanie dát. V prípade, že ani jeden z predpokladov nemôže byť splnený, nedá sa hovoriť o paralelnom spracovaní. Ideálnym riešením môže byť kombinácia oboch prístupov, ktorá zvyšuje šance distribúcie výpočtu, ak je splnený aspoň jeden vstupný predpoklad.

### 2.3 Granularita

Vhodné zvolenie veľkosti granularity plánov pre distribúciu má dôležitý vplyv na návrh a výkon výsledného systému. Jemná granularita umožňuje lepšie rozdeliť úlohu medzi dostupné uzly, avšak za cenu veľkej komunikácie, naproti tomu hrubá granularita zaručí menej komunikácie, ale nemusí umožniť vhodné využitie dostupného výpočtového výkonu. Z pohľadu Boboxu existujú tri prístupy k rozdeleniu plánov z pohľadu granularity a to na úrovni *krabičiek*, *časti plánov* a *celých plánov*. Každé z týchto rozdelení kladie iné požiadavky na podporu zo strany systému.

Najjemnejšiu granularitu poskytuje rozdelenie na úrovni jednotlivých krabičiek (Obr. 1b). Tento spôsob nevyžaduje špeciálny prístup frontendu pri vytváraní plánov a na uzloch bude postačovať odľahčená verzia Boboxu pozostávajúca z backendu a distribučnej logiky obsluhujúcej komunikáciu s hlavnou riadiacou časťou, prijímajúcej úlohy a zabezpečujúcej vzdialené preposielanie správ. Vzdialená komunikácia v tomto prípade úplne nahradí komunikáciu lokálnu.

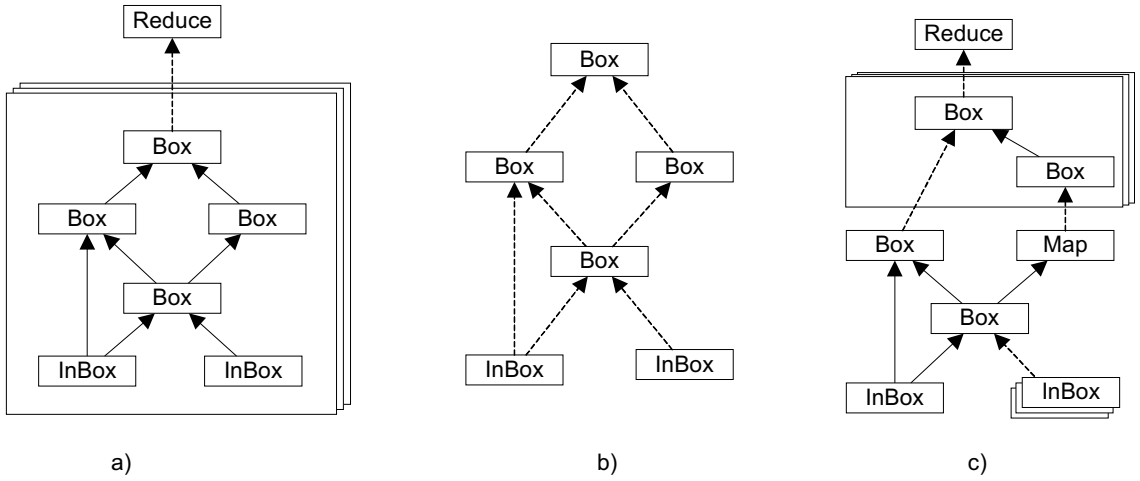
Distribúcia na úrovni celých plánov je hrubá forma granularity, ktorá redukuje veľkosť komunikácie, avšak degraduje možnosti distribúcie (Obr. 1a). Použitie má najmä v prípadoch, ak plán spracovania je veľmi jednoduchý, resp. ho nie je možné deliť na menšie časti. V takýchto prípadoch je nutné, aby bolo možné rozdeliť vstupné dáta medzi inštancie celého plánu. V opačnom prípade sa nemôže hovoriť o distribuovanom spracovaní. Toto riešenie požaduje, aby frontend vytváral plány s parametrizovatelnými vstupmi, čo umožní automaticky definovať rozsah dát, nad ktorými má konkrétny uzol pracovať. Taktiež bude musieť poskytnúť implementáciu Reduce krabičky, ktorá bude mať na starosti spojenie výstupov jednotlivých dielčich častí.

Distribúcia časti plánov je kombináciou predchádzajúcich riešení (Obr. 1c). Vyvážuje granularitu podľa veľkosti a typu aktuálne spracovávanej úlohy. Postupnosť operácií, ktoré je efektívnejšie vykonať lokálne, môže vytvoriť zložitejšie podplány, no je zachovaná možnosť rozdelenia plánu. Vyžaduje, aby na jednotlivých uzloch bežal plnohodnotný Bobox a vyžaduje náročnejšiu logiku vo frontende, ktorý musí byť schopný rozhodnúť o rozdelení plánu. Komunikáciu medzi krabičkami je nutné rozšíriť o vzdialenú.

### 2.4 Správy

Predávanie dát medzi krabičkami prebieha v Boboxe pomocou predávania sprav, ktoré sú zabalene v takzvaných obálkach. Formát obálok v Boboxe je optimalizovaný s ohľadom na beh na lokálnom systéme. Dáta sú v obálke reprezentované následovne: jednoduché dátové typy (číselné dátové typy a logické hodnoty) sú v obálke uložené priamo, naproti tomu pri zložených dátových typoch (typicky textové reťazce) je uložená ich hash hodnota a jednoznačný identifikátor (pre efektivitu typicky ukazateľ do pamäte). Táto reprezentácia zaručuje stabilnú veľkosť obálok a jej optimalizáciu vzhľadom na veľkosti cache pamäti v systéme pre dosiahnutie maximálnej rýchlosti výpočtu. Avšak táto reprezentácia zložených dátových typov je v prípade identifikátorov ako ukazateľov do lokálnej pamäte nepoužiteľná v distribuovanom prostredí.

V prípade použitia identifikátorov je nutné zabezpečiť dostupnosť mapovania identifikátorov na hodnoty reťazca vo výpočtových uzloch. K tomuto účelu je možné využiť napríklad distribuovaný súborový systém (DFS), distribuovaná databáza (DDB), alebo zdieľaná pamäť. Použitie robustnejšieho DFS alebo DDB, môže naraziť na




**Obr. 1.** Príklady plánov pre rôzne stupne granularity. Plné čiary reprezentujú tok lokálnych sprav, čiarkované tok vzdialených sprav. a) hrubá granularita - plán je vykonávaný v celku, avšak každý uzol pracuje na inej časti dát. Rozdelenie vstupu je docielené nastavením rôznych rozsahov pre InBox krabičky. Výsledok je dosiahnutý spojením dielčích dát v pridanej Reduce krabičke. b) jemná granularita - každá krabička je vykonávaná distribuovane a všetka komunikácia je vzdialená. c) kombinácia predchádzajúcich - plán je rozdelený na väčšie celky, ktoré môžu byť vykonávané distribuovane.

problém s rýchlosťou distribúcie dát v rámci použitého systému. Správa môže byť doručená krabičke na cieľový uzol a ta byť následne naplánovaná rýchlejšie, než budú dáta pre tento cieľový uzol reálne dostupné. Pridanie mechanizmu na pozdržanie spracovania do doby, než budú dáta skutočne dostupné, znamená zníženie výkonu systému ako celku. Riešenie so zdieľanou pamäťou nie je vhodné pre veľké objemy dát, keďže má rádovo menšie kapacity než DFS, či DDB a hostiaci uzol je úzkym komunikačným hrdlom.

Druhou možnosťou je použitie marshallingu na vzdialené obálky, tj. posielať konkrétne dáta. Týmto spôsobom je zaručené, že dáta budú stále so správou a teda je zaručená ich dostupnosť v čase výpočtu. Prebalenie obálok je z pohľadu krabičky transparentné. Predpokladáme, že takéto predávanie dát bude rýchlejšie než distribúcia dát v DSF, či DDB.

Vhodnosť každého zo spomenutých riešení závisí od veľkosti a povahy dát v riešenom probléme. Pokiaľ dáta obsahujú napr. veľké texty, ktoré sa v priebehu výpočtu nemenia (napr. dotazovanie sa nad RDF dátami pomocou SPARQL) tak je výhodnejšie použiť prvý spôsob. Mapovanie identifikátorov sa rozdistribuuje na začiatku výpočtu a je dostupné po celý čas výpočtu všetkým uzlom, čím dochádza k úspore času za (de)marshalling. Pri využití vhodného DFS je moplánovať vykonávanie medzi uzly obsahujúce potrebné dáta.

Avšak v prípade, ak sa dáta počas výpočtu neustále menia, je vhodné použiť druhý spôsob, pretože sa dá očakávať, že marshaling bude rádovo efektívnejší, než sa spoliehať na distribúciu dát v použitom úložisku. Taktiež v tomto prípade nebude nutné riešiť prípadne ďalšie problémy, ako napríklad migráciu výpočtu bližšie k dátam, ak DFS rozhodne o umiestnení mapovaní medzivýsledkov na iný uzol.

Keďže vhodný spôsob závisí na riešenom probléme, je vhodné podporovať oba postupy a použitie správneho z nich nechať na implementácii problému užívateľom.

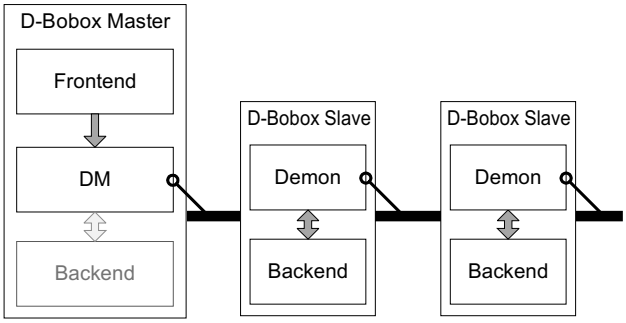
## 3 Návrh architektúry

Vzhľadom na základnú myšlienku systému D-Bobox je riadiaca logika distribúcie umiestnené do jádra, kde tvorí medzivrstvu medzi frontendom a backendom. Ďalej umožňuje podporou pre distribúciu na oboch úrovniach, t.j. na úrovni dát aj výpočtu. Veľkosť granularity bude určovaná frontendom a môže byť rôzna pre každú individuálnu úlohu. V ďalšej kapitole nasleduje podrobnejší popis návrhu architektúry systému D-Bobox.

### 3.1 Architektúra

D-Bobox pozostáva z dvoch celkov Bobox Master a Bobox Slave, líšiacich sa nasadením a súčasťami, ktoré obsahujú. Schému systému a jeho nasadenie popisuje obrázok 2.

Bobox Master (BM) beží v jedninej inštancii na hlavnom uzle. Je to riadiaci celok distribuovaného systému a poskytuje rozhranie pre komunikáciu s užívateľom. Pozostáva z frontendu, dis-




**Obr. 2.** Architektúra distribuovaného Boboxu — D-Boboxu.

tribučnej vrstvy a backendu. Frontend transformuje zadanie úlohy od užívateľa na plán spracovania. Tento plán oproti plánu Boboxu obsahuje pomocne informácie pre riadene distribúcie následujúcou vrstvou a môže navyše obsahovať špecifické distribučné krabičky (map, reduce). Distribučná vrstva je nová vrstva medzi frontendom a backendom. Je tvorená Distribučným Manažérom (DM), ktorý má za úlohu rozdistribuovať plán úlohy medzi dostupné Bobox Slave uzly, riadiť distribuované spracovanie a zber výsledkov. Distribučný Manažér pilotnej verzie podporuje iba statickú, pri spustení definovanú množinu pracovných uzlov, čo umožňuje primárne zameranie sa na možnosti a schopnosti automatického generovania distribuovateľných plánov. V ďalšom postupe vývoja bude rozšírený o podporu behového plánovania úloh v dynamickej množine dostupných pracovných uzlov, možnosti vyvažovania záťaže a podobné rozšírenia. Backend vrstva je na BM voliteľná, takže je na užívateľovi, či povolí, aby sa riadiaci uzol podieľal na samotnom výpočte, alebo aby sa venoval iba riadeniu.

Bobox Slave (BS) je pracovná časť D-Boboxu, ktorá beží na výpočtových uzloch. Pozostáva z dvoch častí: backend a démon. Úlohou backendu je lokálne paralelne vykonávanie zadaného (pod)plánu. Démon má za úlohu komunikáciu s ostatnými uzlami pri posielaní vzdialených správ a s hlavným uzlom, na ktorom beží Bobox Master. Od BM prijíma plány, ktoré majú byť vykonané na danom uzle a iniciuje beh backend časti. Zdrojom dát pre BS sú read krabičky z dostupného zdroja dát (DFS, DDB), alebo správy doručené od vzdialených krabičiek pomocou vzdialených správ. V prípade vzdialených správ je komunikácia obslúžená démonom a z pohľadu krabičiek sa javí ako lokálna.

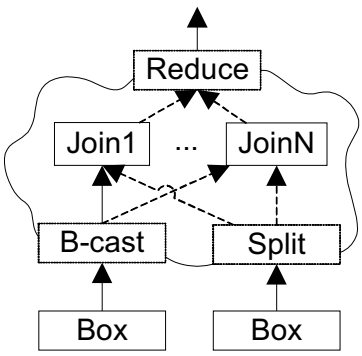
## 4 Príklad využitia: Distribuovaný SPARQL engine

Ako príklad možného využitia D-Boboxu môže poslúžiť SPARQL engine na vykonávanie požiadaviek nad RDF dátami. RDF je štandardizovaný model na výmenu dát na webe [10] a je jedným z dôležitých nástrojov sémantického webu. V súčastnosti je veľkosť a dostupných sémantických dát z rôznych oblasti obrovská a neustále narastá. SPARQL [11] je jazyk požiadaviek nad RDF dátami, pre ktorý je typická operácia join RDF trojíc. Táto často krát pracuje s veľkým množstvom dát a je teda úzkym hrdlom pri spracovaní na jednom uzle, avšak predstavuje vhodného kandidáta na zvýšenie efektivity pomocou distribuovaného spracovania [12,13].

Pri operácií join (bez ďalších predpokladov) dochádza k porovnávaniu každého prvku z jedného vstupu operácie s každým prvkom na druhom vstupe. Aj pri použití prúdového spracovania, keď sú výstupné dáta jednej krabičky priebežne odosielané ďalším krabičkám tak, ako sú generované, musia byť prítomne kompletné dáta jedného vstupu, ktoré môžu byť ďalej priebežne porovnávané s prúdom dát zo vstupu druhého. K zvýšeniu efektivity je možné použiť informácie o veľkosti vstupov, prípadne zotriedenie vstupov k efektívnejšiemu prehľadávaniu. Veľká časť typických SPARQL dotazov obsahuje join operácie, ktoré majú na vstupe veľké dáta. Pokiaľ sú veľké dáta iba na jednom vstupe, môžu byť spracovane prúdovo oproti uloženým menším dátam z druhého vstupu. V prípade oboch veľkých vstupov je navyše kladená zvýšená pamäťová náročnosť na udržanie jedného celého vstupu.

Distribuovaním tejto operácie medzi N uzlov je možné teoreticky dosiahnuť lineárneho zrýchlenia výpočtu. V prvom prípade sa to dosiahne rozdelením veľkých vstupných dát medzi N uzlov a porovnaním týchto časti s kópiou dát druhého vstupu na každom uzle. V druhom prípade zrýchlenie dosiahneme rozdelením uložených dát medzi uzly a preposlaním prúdu dát druhého vstupu na všetky uzly (nie je možné deliť dáta oboch vstupov, pokiaľ má dôjsť k porovnaniu každého záznamu s každým). Týmto dôjde nielen k zmenšeniu počtu porovnaní na jednom uzle, ale aj k zníženiu pamäťovej náročnosti na uloženie porovnávaného vstupu.

Príklad možnosti konkrétneho plánu pre distribuovanú operáciu join znázorňuje obrázok 3. Dáta z ľavého vstupu sú preposlané na vstup join krabičiek na všetkých zúčastnených uzloch pomocou špeciálnej krabičky *B-cast*. Dáta z druhého vstupu sú rozdelene medzi jednotlivé uzly, napríklad pomocou metódy round-robin v krabičke *Split*. Výstupy z jednotlivých uzlov sú následne redukované pomocou *Re-*




**Obr. 3.** Ukážka jednej z možností distribúcie pre operáciu join.

*duce* krabičky. Expanziu plánu v obláčiku má na starosti DM, pričom pôvodný plán frontendu bude obsahovať prechodové krabičky (B-cast, Split, Reduce) a jednu operáciu Join.

Podobne efektívne využitie distribúcie bude možné využiť aj pre ďalšie operácie ako napríklad optional (left join), filter, sort [14], či načítanie trojíc.

## 5 Záver

V článku sme navrhli spôsob rozšírenia systému Bobox na D-Bobox pridaním možnosti distribuovaného spracovania úloh. Popri tom sme ukázali, že navrhnutý systém je schopný pokryť nielen úlohy, na ktoré sa v súčasnosti používa MapReduce schéma, ale podporuje aj úlohy vyžadujúce zložitejšiu schému výpočtu.

Oproti v súčasnosti populárnemu Hadoopu nemá Bobox limitácie ako napríklad vykonávanie operácií Reduce až po vykonaní všetkých Map operácií. Jeho prúdové spracovanie dát zvyšuje efektivitu spracovania a v ideálnych prípadoch minimalizuje nároky na veľkosť ukladaných medzivýsledkov. Taktiež umožňuje distribuované vykonávanie zložitejších plánov ako napríklad kompilácia zdrojových kódov, ktoré obsahujú závislosti a nemôže byť teda vykonaná v jednom kroku ako to vyžaduje MapReduce.

Budúca práca pozostáva z dokončenia implementácie a nasadenie na SPARQL engine. To nám umožní otestovať prínos nášho návrhu a experimentálne overiť schopnosť škálovania D-Boboxu. Následne sa zameriame na podpore behu D-Boboxu na nespoľahlivých sieťach a v dynamickom prostredí s cieľom poskytnúť kvalitný nástroj na ľahký vývoj efektívnych, dátovo a výpočtovo náročných aplikácií v distribuovanom prostredí.

## Literatúra

1. D. Bednarek, J. Dokulil, J. Yaghob, F. Zavoral: *Using methods of parallel semi-structured data processing for semantic web*. In 3rd International Conference on Advances in Semantic Processing, SEMAPRO. IEEE Computer Society Press, 2009, 44–49.
2. Z. Falt, D. Bednarek, M. Cermak, F. Zavoral: *On parallel evaluation of sparql queries*. In DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications. IARIA, 2012, 97–102.
3. J. Dokulil, D. Bednarek, J. Yaghob: *The bobox project: Parallelization framework and server for data processing*. Charles University Prague, Technical Report 1, 2011.
4. M. Cermak, J. Dokulil, Z. Falt, F. Zavoral: *SPARQL Query Processing Using Bobox Framework*. In SEMAPRO 2011, The Fifth International Conference on Advances in Semantic Processing, IARIA, 2011, 104–109.
5. "Apache hadoop." [Online]. Available: http://hadoop.apache.org/
6. J. Dean, S. Ghemawat: *Mapreduce: simplified data processing on large clusters*. Commun. ACM, 51(1), Jan. 2008, 107–113. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492
7. Z. Ma, L. Gu: *The limitation of mapreduce: A probing case and a lightweight solution*. In CLOUD COMPUTING 2010: Proc. of the 1st Intl. Conf. on Cloud Computing, GRIDs, and Virtualization, 2010, 68–73.
8. M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly: *Dryad: distributed data-parallel programs from sequential building blocks*. SIGOPS Oper. Syst. Rev., 41(3), Mar. 2007, 59–72. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273005
9. M. Cermak, J. Dokulil, F. Zavoral: *SPARQL Compiler for Bobox*. In SEMAPRO 2010, The Fourth International Conference on Advances in Semantic Processing. IARIA, 2010, 100–105.
10. J. J. Carroll, G. Klyne: *Resource description framework: concepts and abstract Syntax*, W3C, 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/
11. E. Prud'hommeaux, A. Seaborne: *SPARQL query language for RDF*. W3C, 2008. [Online]. Available: http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/
12. B. Gedik, P. S. Yu, R. R. Bordawekar: *Executing stream joins on the cell processor*. In Proceedings of the 33rd International Conference on Very Large Data Bases, ser. VLDB '07. VLDB Endowment, 2007, 363–374.
13. C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, P. Dubey: *Sort vs. hash revisited: fast join implementation on modern multi-core cpus Proc. VLDB Endow.*, 2, August 2009, 1378–1389.
14. Z. Falt, M. Kruliš, Y. Jakub: *Optimalizace třídicích algoritmů pro systémy proudového zpracování dat*. In Informačné Technológie – Aplikácie a Teória. PONT s. r. o., 2011, 69–74.

# Zachovanie mentálnej mapy farebného zobrazenia popiskov hrán*

Jiří Dokulil[1] and Jana Katreniaková[2]

[1] Universität Wien, dokulil@gmail.com
[2] Univerzita Komenského v Bratislave, katreniakova@dcs.fmph.uniba.sk

**Abstrakt** *V oblasti kreslenia grafov sa významná časť venuje zobrazovaniu popiskov hrán. Medzi bežné spôsoby patrí umiestnenie popiskov na okraji obrázku. Takto zobrazené popisky sa spájajú s hranou pomocou vodiacich čiar, čo zneprehľadňuje zobrazenie grafu. Navrhli sme spôsob, ako pomocou farieb túto neprehľadnosť odstrániť. V prípade zmien v grafe potrebujeme miesta popiskov a ich farby meniť v závislosti od týchto zmien. Používateľ si však už pri prvom zobrazení grafu vytvoril svoju mentálnu mapu, ktorá zohľadňuje použité farby a miesta popiskov. V článku popisujeme spôsob, ako upraviť zobrazenie popiskov ak chceme zachovať túto mentálnu mapu používateľa.*

## 1 Úvod

Kreslenie grafov sa bežne využíva na vizualizáciu štruktúrovaných dát. Informácia sa delí na objekty (reprezentované vrcholmi) a vzťahy medzi nimi (reprezentované hranami). Techniky kreslenia grafov následne priradia vrcholom pozície v priestore a hranám krivky spájajúce zodpovedajúce vrcholy na základe vybraných estetických kritérií.

Avšak pre množstvo aplikácii je dôležitý aj nejaký bližší popis objektov a relácií – ohodnotenie vrcholov a hrán. V oblasti kreslenia grafov sa preto venuje významná oblasť aj umiestňovaniu popiskov vrcholov a najmä hranám.

V tomto článku sa venujeme umiestňovaniu popiskov hrán. Hrany (na rozdiel od vrcholov) totiž samé o sebe nemôžu obsahovať nejakú informáciu a preto ich popisky zaberajú miesto niekde v priestore a môžu kolidovať s ostatnými vykreslenými prvkami. Od umiestnenia popisku vyžadujeme, aby nekolidoval s inými prvkami a bol jednoznačný (t.j. jednoznačne priraditeľný k práve jednej hrane).

Tradične sa popisky umiestňujú niekde v blízkosti čiar, ktoré reprezentujú hrany (napríklad [3,13]). Pre popis hrany sa vyhradí obdĺžnikový priestor a tento sa umiestni na základe zvolených kritérií nad alebo pod hranu vo vhodnej vzdialenosti od vrchola. Vo všeobecnosti ak máme graf a pre každú hranu $e$ konečný počet možných kandidátov na umiestnenie $C_e$, potom nájdenie umiestnenia pre každú hranu aby žiadne dve nekolidovali je NP-ťažké [11]. Preto existuje najmä množstvo heuristík riešiacich problém umiestnenia popiskov. Prehľad existujúcich prác na túto tému možno nájsť v [12].

Práve pre NP-ťažkosť problému umiestnenia popiskov sa začali objavovať alternatívne riešenia. V niektorých prácach [1,6] sa popisky hrán neumiestňujú priamo do vykresleného grafu, ale na okraj a s príslušnou hranou sa spoja pomocou vodiacich čiar. Vodiace čiary ani popisky sa nesmú navzájom pretínať. Toto riešenie je vhodné najmä v oblasti kreslenia máp, kde sa na okraji umiestňujú popisky objektov. Pre klasické kreslenie grafov je ale prítomnosť vodiacich čiar skôr mätúca a spôsobí ešte neprehľadnejší obrázok. Preto sme v [2] navrhli framework pre zobrazovanie popiskov na kraj, ale bez vodiacich čiar. Tieto sú nahradené farbami, ktoré označujú hrany a ich popisky.

V tomto článku naše riešenie popíšeme podrobnejšie a zameriame sa aj na dynamické zmeny v grafe. V prípade zmien v grafe, či už sa jedná o zmeny v štruktúre grafu alebo zmenu pohľadu na graf, potrebujeme miesta popiskov a ich farby meniť v závislosti od týchto zmien. Používateľ si však už vytvoril svoju mentálnu mapu, ktorá zohľadňuje použité farby a miesta popiskov. Túto mentálnu mapu je možné zachovať, ak nebudeme graf pri zmene od základu prekresľovať, ale skôr spravíme drobné ústupky v ostatných estetických kritériách – dĺžky hrán, počet farieb a pod.

Článok je ďalej štruktúrovaný nasledovne: V nasledujúcej časti popisujeme spôsob umiestňovania popiskov pre hrany. Ďalej popíšeme, čo je pre používateľa pri zobrazovaní popiskov dôležité a načrtneme problémy, ktoré potrebujeme riešiť, ak chceme zachovávať mentálnu mapu. Ponúkame dva rôzne prístupy k zachovaniu mentálnej mapy – update schémy pre jednotlivé operácie na grafe a komplexné riešenie využívajúce zmenu váhovania riešenia v pôvodnom algoritme. Na záver zhrnieme naše riešenie a pozrieme sa na otvorené problémy, ktoré bude potrebné riešiť.

## 2 Algoritmus

Popisky zobrazujeme ako obdĺžniky s textom a farebným pozadím v stĺpci naľavo od nakreslenia grafu. Ich pozícia však nie je úplné ľubovoľná – musia ležať v dopredu určených slotoch pevnej šírky. Pre vizuálne spojenie s hranou pridáme na každú popisovanú hranu

* Práca bola podporená grantom VEGA 1/1085/12.

farebný štvorček, tzv. značku. Potom popisok s napríklad červeným pozadím obsahuje text prislúchajúci k hrane s červenou značkou. Keďže rozlíšiteľných farieb je iba obmedzené množstvo používame pre viac dvojíc popisok-značka rovnakú farbu. Pridelenie rovnakej farby však nesmie spôsobiť nejednoznačnosť priradenia popisku ku značke a naopak.

Algoritmus pre zobrazovanie popiskov hrán pracuje v troch základných fázach.

V prvej fáze sa zvolí pozícia popiskov – popisky pritom umožňujeme umiestňovať na ľavom kraji obrázku a ich maximálny počet je dopredu daný. Vstupom je informácia, ktoré časti hrán sú v aktuálnom pohľade viditeľné. Je totiž možné, že z grafu je zobrazená iba určitá oblasť, ktorá neobsahuje všetky hrany. Preto o každej hrane zistíme minimálnu a maximálnu y-ovú súradnicu, na ktorej je hrana zobrazená. Pre každú hranu a potenciálne umiestenie popisku spočítame cenu, ktorá určuje nakoľko je pozícia vhodná na popisok danej hrany. V súčasnosti je cena vzdialenosť pozície od priemeru minimálnej a maximálnej y-ovej súradnice. Pokiaľ je pozícia mimo týchto súradníc je cena dvojnásobná. Výsledné pozície popiskov sú zvolené tak, aby súčet cien jednotlivých umiestnení bol minimálny – v podstate ide o problém hľadania najlacnejšieho párovania.

V druhej fáze algoritmus vyberá pozíciu značiek – farebného označenia hrán, ktoré zodpovedá farbe popisku. Aj tu je použitá cenová funkcia, algoritmus výberu je však iný. Prechádza postupne jednotlivé hrany a pre každú vyberie kandidátov – body na nakreslenie značky na hrane. Potenciálne body by mali byť primerane husto, v našom prípade volíme v odstupe cca 5 pixelov. Pre každý takýto bod vypočítame cenu – vhodnosť umiestnenia značky. Body mimo aktuálne zobrazený výsek neuvažujeme rovnako ako body prekryté inými objektami – vrcholmi. Základná cena je vzdialenosť y-ovej súradnice bodu od y-ovej súradnice popisku plus x-ová súradnica bodu vydelená 100. Preferované riešenie je teda umiestnenie značky v rovnakej výške ako popisok s tým, že pokiaľ je to možné, tak čo najbližšie k ľavému kraju, kde sú umiestňované popisky.

Posledným krokom je pridelenie farieb. Vstupom sú tzv. „intervaly vplyvu“, čo je množina intervalov obsahujúca pre každú dvojicu popisok-značka interval y-ových súradníc ktorý je tak veľký, aby zaručil, že pokiaľ v ňom nebude umiestený popisok alebo značka rovnakej farby ako k nemu prislúchajúca dvojica, tak bude možné jednoznačne priradiť správny popisok ku značke podľa farby (t.j. najbližší popisok farby značky je práve ten správny). To isté musí platiť aj opačne. Vznikne graf, kde vrcholy sú hrany pôvodného grafu a medzi dvoma vrcholmi je hrana práve vtedy, ak sa prekrývajú príslušné intervaly vplyvu. Ofarbením tohto grafu vznikne ofarbenie popiskov a značiek. S ohľadom na špecifické vlastnosti tohto grafu je možné nájsť optimálne ofarbenie v čase $\mathcal{O}(n * log(n) + n * k)$, kde $n$ je počet hrán a $k$ počet farieb.

Príklady výstupu sú na obrázku 1. Prvý z obrázkov je identický s výstupom tu popísaného nemodifikovaného algoritmu. Zvislé čiary napravo od popiskov znázorňujú interval vplyvu.

## 3 Zachovanie mentálnej mapy

Pri statickom vykreslení grafu sa ako najdôležitejšie považujú statické kritériá ako napríklad veľkosť obrázku, počet krížení hrán, dĺžka hrán a podobne. Preto pôvodný algoritmus eliminoval nepotrebné vodiace čiary, ktoré spôsobovali zbytočné kríženia čiar, čím zneprehľadňovali obrázok.

V prípade dlhšej práce používateľa s grafom a jeho vykreslením sa však graf stáva dynamickým a počas práce sa môže meniť (pridávať a odoberať vrcholy a hrany). Okrem toho, najmä pri väčšom grafe je možné, že v aktuálnom okne nie je zobrazený celý, ale iba nejaká jeho časť. Potom okrem spomínaných zmien v grafe je potrebné riešiť aj situáciu, kedy sa posúva zobrazené okno.

Pri každej z týchto zmien sa graf (alebo jeho zobrazená časť) mení. Používateľ však už má zapamätaný istý vnútorný obraz grafu a jeho súčastí, ktorému sa hovorí mentálna mapa grafu [8]. V prípade, že sa následne vykonajú zmeny v grafe je zachovanie tejto mentálnej mapy dôležitejšie ako statické estetické kritériá [10,7,5].

### 3.1 Mentálna mapa

Ako už bolo spomínané vyššie, používateľ si pri pohľade na vykreslený graf uchováva v pamäti niektoré významné črty tohto grafu. Pri zmene v grafe, ktorá ovplyvní aj vykreslenie grafu, vie používateľ na základe týchto uchovaných čŕt nájsť/detekovať zmeny v grafe. Je preto dôležité, aby tieto zmeny boli v takom rozsahu, aby ich používateľ vedel všetky zachytiť.

Klasicky mentálna mapa grafu zahŕňa hlavne pozície vrcholov pred a po zmene – príslušnosť do clustra vrcholov, vzájomná pozícia vrcholov a podobne. O rozšírenie pojmu mentálna mapa aj pre hrany sa pojednáva v [9]. Pri vykresľovaní hrán v grafe preto zrejme bude dôležité zachovávať mentálnu mapu hrán, ktorá obsahuje hlavne smery vychádzania hrán a umiestnenie významných ohybov. Náš algoritmus však spracováva popisky na hranách a tak mentálna mapa celého vykreslenia musí obsahovať aj informáciu o umiestnení popiskov a ich farbe.

*Umiestnenie značiek a popiskov* Asi najdôležitejším kritériom je umiestňovanie popiskov na okraji obrázku (umiestnenie značky na hrane sa určuje na základe umiestnenia popisku). Preto by sme mali zachovávať približnú pozíciu z popiskov a ich vzájomné polohy (v ideálnom prípade nemeniť poradie). Kritické sú najmä zmeny poradia popiskov s rovnakou farbou – tieto výmeny by sa vyskytovať nemali, nakoľko sú pre používateľa mätúce.

*Farby popiskov* Podobne dôležité ako umiestnenie popiskov je aj ich farba. Používateľ si môže z predchádzajúceho používania pamätať farbu popisku a táto by sa nemala výrazne meniť. Tento problém však zatiaľ nevieme pre všetky modifikácie grafu uspokojivo riešiť bez zbytočného plytvania farbami.

### 3.2 Úpravy grafu

V prípade nejakých zmien v grafe – pridávanie alebo mazanie vrchola alebo hrany je potrebné graf a teda aj jeho popisky prekresliť. Najjednoduchšie riešenie by bolo vykresliť graf ako úplne nový nezávisle od predchádzajúceho vykreslenia. Toto riešenie môže umiestniť popisky na úplne iných miestach a dokonca použiť iné množstvo farieb (a teda úplne iné farby).

Keďže však chceme zachovávať mentálnu mapu pri zmenách v grafe, musíme riešiť prekreslenie s ohľadom na predchádzajúce umiestnenie popiskov a ich farbu. Pre každú modifikáciu grafu preto potrebujeme vlastný update, ktorý zachováva mentálnu mapu. Je zrejmé, že toto zachovávanie mentálnej mapy nás bude stáť istú neefektívnosť v ostatných parametroch.

*Odobratie hrany* Najjednoduchším prípadom je odobratie hrany. Ak odoberieme hranu a jej popisok, nevznikne žiadna kolízia, ktorú by bolo treba riešiť. Každá hrana má svoju značku a popis, ktoré zodpovedajú správnemu označeniu a nie sú navzájom kolidujúce.

*Pridanie hrany* Pri pridaní hrany by sme radi nemenili pozície a farby popiskov ostatným hranám. Toto je možné v prípade, že existuje voľný slot na umiestnenie popisku. V prípade, že takýto slot existuje (nech je aj ľubovoľne ďaleko) je možné ho použiť ak by sme vedeli pridať farbu tak, aby interval vplyvu novej hrany nekolidoval s iným intervalom tejto farby. Najhorší prípad nastáva, ak máme obsadené všetky sloty a preto je potrebné počet slotov zdvojnásobiť. Tým sa však dostávame do situácie, kedy sme iste získali voľné miesto pre nový popisok priamo na najvhodnejšom mieste, pre ktoré vieme nájsť farbu.

*Pridanie a odobratie vrchola* Pridanie a odobratie vrchola nám vo všeobecnosti môže zmeniť komplet nakreslenie hrán. Ich smerovanie je totiž závislé od prekážok – vrcholov. Keby sme pri odobratí vrchola všetky hrany upravili, aby zodpovedali estetickým kritériám pre hrany, veľmi jednoducho by sa mohlo stať, že neprechádzajú cez y-ovú súradnicu svojej značky. Sú dve možnosti, ako sa s problémom vysporiadať:

- Hrany pri pridaní alebo odobratí vrchola meníme len minimálne – pri odobratí iba vyrovnáme úsek pôvodne obchádzajúci odobratý vrchol a pri pridaní úsek porušený pridaním vrchola presmerujeme aby neprechádzal cez žiaden vrchol. Na presmerovanie môžeme použiť napríklad algoritmus [4]. Toto riešenie vo väčšine prípadov spôsobí, že značky hrán môžu ostať na svojom mieste a teda môžu ostať na mieste aj popisky. Týmto zároveň dosiahneme, že ani farby nemusíme meniť.
- Hrany naozaj prekreslíme s tým, že hrany neovplyvnené pridaným/odobratým vrcholom dostanú prednostne svoju pozíciu. Ostatné hrany nakreslíme odznovu a v prípade, že obsahujú aj y-ovú pozíciu bývalej značky tak ju prednostne využijeme. V prípade že nie, je potrebné pozície meniť a tým pokaziť mentálnu mapu používateľa.

### 3.3 Globálne riešenie pomocou váh

Alternatívou k riešeniu, ktoré sa snaží vyriešiť konkrétne situácie pri modifikácii grafu je zovšeobecnenie algoritmu pre zobrazovanie popiskov tak, aby dokázal vygenerovať nové zobrazenie po zmene (grafu alebo pohľadu) s aspoň čiastočným zachovaním mentálnej mapy.

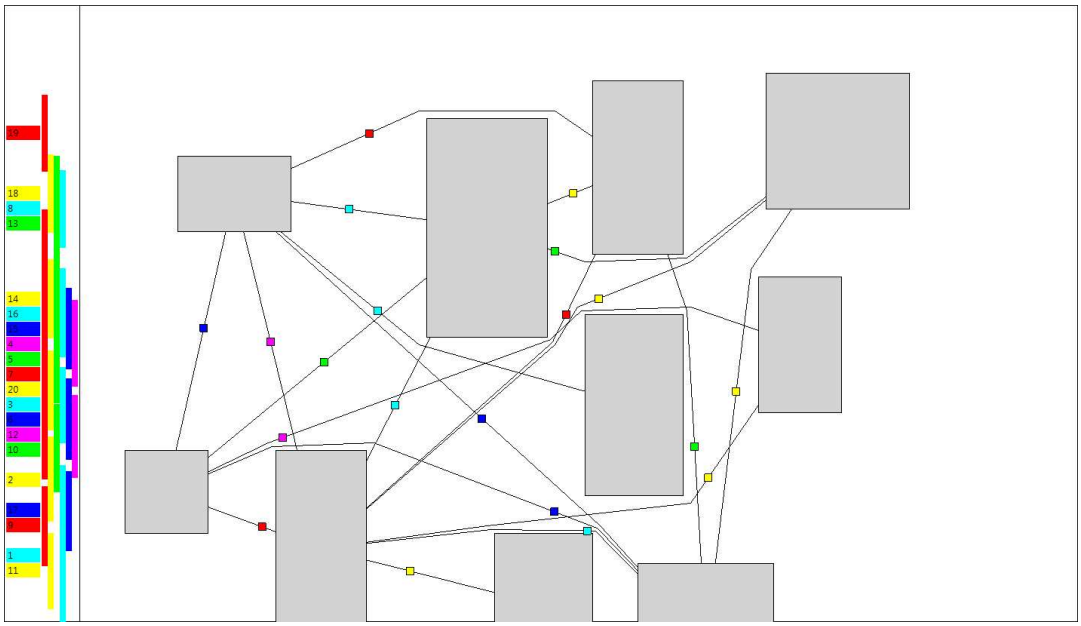
Keďže pre výber pozície popisku používame minimalizáciu ohodnocovacej funkcie nad množinou potenciálnych riešení, môžeme dosiahnuť zaujímavé výsledky jednoduchou zmenou tejto ohodnocovacej funkcie. Vďaka tomu, že táto funkcia nevyžaduje splnenie žiadnych významných predpokladov, možeme si túto zmenu dovoliť.

Najviac sa zatiaľ osvedčilo riešenie, kedy k cenovej funkcii ešte pripočítame vzdialenosť zvažovanej pozície popisku od pôvodnej pozície (pred zmenou). Pokiaľ taká pozícia neexistuje (hrana bola pridávaná, alebo nebola pred zmenou vidieť) nepripočítame nič.

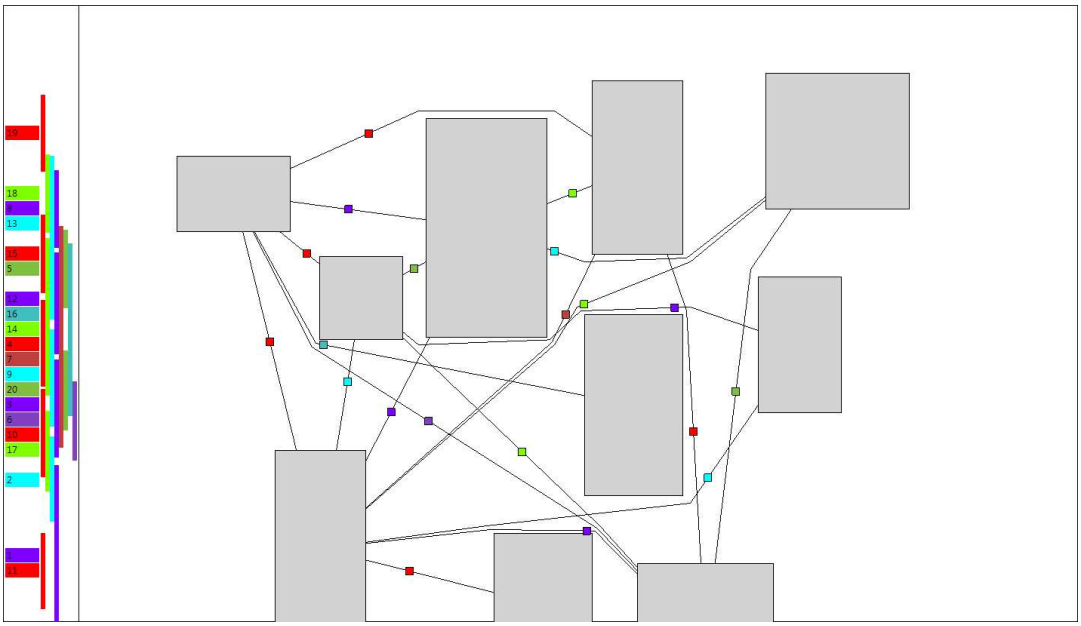
Celkovo dosiahneme, že popisky „preferujú“ zachovanie svojej predchádzajúcej pozície alebo aspoň minimalizáciu zmeny. Pri nových popiskoch umiestnenie nie je ovplyvnené.

Ďalším krokom je umiestnenie značiek na hranách. Aj v tomto prípade používame cenovú funkciu, ktorú je možné takmer ľubovoľne upraviť. Podobne ako v predchádzajúcej fáze sa snažíme, aby preferovala svoju predchádzajúcu pozíciu tým, že k cene pripočítame vzdialenosť od predchádzajúcej pozície.

Najväčším problémom je výber farieb, ktorá nepoužíva cenovú funkciu, ale ofarbí popisky aj značky



(a) Pôvodný stav



(b) Po presune

**Obr. 1.** Príklad aktualizácie – presunutie vrcholu

s použitím minimálneho množstva farieb. Aj relatívne malá zmena v pozícii značky môže spôsobiť veľkú zmenu intervalu vplyvu a tým môže vyvolať „kaskádu“, takže sa zmení počet farieb a ich alokácia. Napriek tomu v súčasnej dobe používame toto riešenie.

Príklad výstupu modifikovaného algoritmu ukazuje obrázok 1.

## 4 Záver

Zobrazenie popiskov na hrany, ktoré sme prezentovali, sprehľadňuje zobrazený graf, nakoľko sa potenciálne dlhé popisky umiestnia na okraji a nie sú napojené na hrany ani vodiacimi čiarami. V prípade zmien v grafe mentálna mapa používateľa zohľadňuje okrem men-

tálnej mapy vrcholov a hrán aj pozície popiskov a ich farby. Preto sme algoritmus na vykreslenie modifikovali tak, aby sa snažil tieto zachovávať. Pri zachovávaní umiestnenia popiskov dosahuje veľmi dobré výsledky. V prípade, že chceme zachovávať aj farby bude pravdepodobne potrebné vedieť meniť paletu pridaním farby počas behu programu, na čo sa chceme v budúcnosti zamerať.

## Literatúra


1. M. A. Bekos, M. Kaufmann, A. Symvonis, A. Wolff: *Boundary labeling: Models and efficient algorithms for rectangular maps.* In Graph Drawing'04, 2004, 49–59.
2. M. Cermak, J. Dokulil, J. Katreniakova: *Boundary labeling of graph edges using colors, 2012.* To appear in IV2012: Proceedings of the 16th International Conference Information Visualisation.
3. U. Dogrusoz, K. G. Kakoulis, B. Madden, I. G. Tollis: *Edge labeling in the graph layout toolkit.* In Proceedings of the Symposium on Graph Drawing (GD'98), Springer-Verlag, 1998, 356–363.
4. J. Dokulil, J. Katreniakova: *Edge routing with fixed node positions.* In IV'08: Proceedings of the 2008 12th International Conference Information Visualisation, Washington, DC, USA, 2008. IEEE Computer Society, 626–631.
5. W. Huang, P. Eades: *How people read graphs.* In Proceedings of the 2005 Asia-Pacific Symposium on Information Visualisation, Vol. 45, APVis'05, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc., 51–58.
6. M. Nöllenburg, V. Polishchuk, M. Sysikaski: *Dynamic one-sided boundary labeling.* In Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS'10, New York, NY, USA, 2010. ACM, 310–319.
7. H. C. Purchase: *Which aesthetic has the greatest effect on human understanding?* In Giuseppe Di Battista, editor, Graph Drawing, Vol. 1353 of Lecture Notes in Computer Science, Springer, 1997, 248–261.
8. H. C. Purchase, E. Hoggan, C. Görg: *How important is the "Mental Map"? – An empirical investigation of a dynamic graph layout algorithm.* In Graph Drawing 2006, Spriger-Verlag Berlin Heidelberg, 2007, 184–195.
9. M. Ďuriš: *Zachovanie mentálnej mapy hrán pri interakcii s grafom.* Master Thesis, Comenius University Bratislava, 2013, v štádiu spracovávania.
10. C. Ware, H. Purchase, L. Colpoys, M. McGill: *Cognitive measurements of graph aesthetics.* Information Visualization, 1, June 2002, 103–110.
11. A. Wolff: *A simple proof for the np-hardness of edge labeling.* Technical Report 11/2000, Institut für Mathematik und Informatik, Universität Greifswald, September 2000.
12. A. Wolff, T. Strijk: *The map-labeling bibliography*, 1996. `http://i11www.ira.uka.de/map-labeling/bibliography`.
13. yFiles. Class library. `http://www.yworks.com/en/products_yfiles_about.html`.

# Příklad pravidelných slovotvorných vzorců v automatickém zpracování češtiny a ruštiny

Jaroslava Hlaváčová, Anja Nedolužko
Ústav formální a aplikované lingvistiky, MFF UK, Univerzita Karlova v Praze
Malostranské náměstí 25, Praha 1

**Abstrakt.** *V češtině i v ruštině existuje množina předpon, jejichž připojením k nedokonavému slovesu a přidáním zvratného zájmena pozměníme význam původního slovesa vždy téměř stejným způsobem. Toho lze využít při automatickém rozpoznávání slovních tvarů, aniž by bylo třeba je ukládat do morfologických slovníků.*

## 1 Úvod

Při automatickém zpracování textů se většinou využívá morfologických slovníků, které obsahují "všechny" slovní tvary (dále jen slova) daného jazyka. Slůvko "všechny" je v uvozovkách proto, že umístit všechna slova do slovníku v praxi nelze, a to z mnoha důvodů. Jedním z nich je neshoda mezi uživateli jazyka, která slova do jazyka patří, a která ne – např. spory o slova přejatá z jiných jazyků, v dnešní době především z angličtiny. Stejně tak není možné zahrnout do slovníku všechna vlastní jména, tedy jména osob, zeměpisné názvy, názvy firem apod.

Dalším důvodem je přirozený vývoj jazyků, který neustále přináší nová a nová slova. Některá se uchytí natrvalo, jiná se objeví jen v několika málo textech, nezřídka jde jen o tzv. okazionalismy, tj. slova vytvořená pro jednu konkrétní příležitost. Jiná slova naopak ze slovní zásoby mizí, a přestože se s nimi můžeme ve starší literatuře setkat, časem přestávají být v běžné populaci srozumitelná. Patří taková slova ještě do současného jazyka, nebo ne?

Slovník tedy z principu nemůže obsahovat všechna slova nějakého jazyka. Při morfologické analýze textů, která je základním kamenem pro všechny další automatické jazykové aplikace, však je třeba rozpoznat co největší množství slov. Naštěstí jsou jazyky do značné míry pravidelné, čehož lze využít k vytvoření tzv. guesserů, tedy k hádání, co slovo, ač nepřítomno ve slovníku, znamená, a jaké má vlastnosti.

V našem příspěvku jsme se zaměřili na dva zástupce slovanských jazyků - češtinu a ruštinu. Zabýváme se zde jen jedním malým výsekem, a to několika slovesnými předponami, které pozměňují nedokonavá slovesa vždy stejným způsobem, takže je lze velmi snadno a spolehlivě analyzovat.

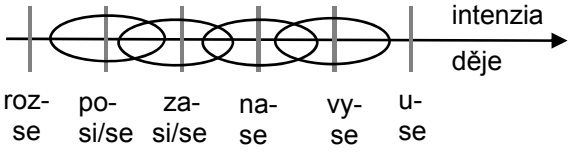
## 2 „Stupňování sloves" v češtině a ruštině

### 2.1 „Stupňovací" předpony

Většina českých a ruských nedokonavých sloves má schopnost spojovat se s některými speciálními předponami v kombinaci se zvratným zájmenem, přičemž takto vzniklé slovo modifikuje význam původního slovesa, a to podobným způsobem.

Jde o české předpony: *roz-, po-, za-, na-, vy-* a *u-* a jejich ruské ekvivalenty *раз*-(*рас*-), *по-, за-, на-, вы-* a *у-*. Pro ruštinu tuto řadu můžeme prodloužit ještě předponou *из-* /=z/.

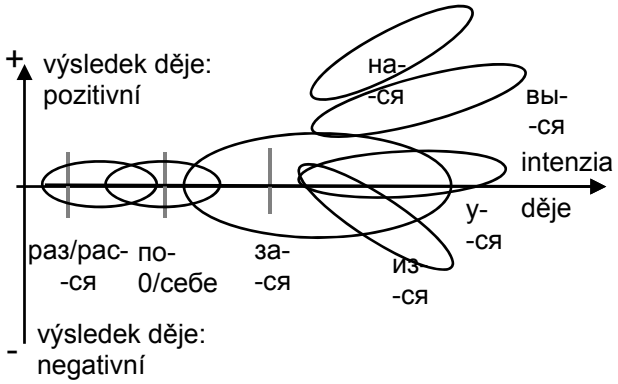
Jednotlivá prefigovaná zvratná slovesa v češtině je možno s určitou tolerancí uspořádat podle intenzity děje, jak ukazuje následující obrázek:




Krajní body tvoří předpony *roz-* a *u-*, uprostřed je podle intenzity posloupnost *po-, za-, na-* a *vy-* s vágním až překrývajícím se rozsahem.

Z tohoto důvodu nazýváme tento způsob tvoření s jistou nadsázkou pracovně „stupňování" intenzity slovesného děje.

Podobně, i když ne úplně stejně, vypadá situace s těmito předponami a zvratným „ся" v ruštině. Srov. následující obrázek::




První tři předpony bychom mohli schématicky zobrazit stejně jako pro češtinu, i když význam předpony *за-* už zasahuje „přes obvyklé hranice". Předpona *раз*-(*рас*-) navíc není umístěna tak striktně na samém začátku škály, je vágnější. Předpony *вы-, на-, у-* a *из-* však jsou již z hlediska intenzity děje na stejné úrovni a liší se sémantickými odstíny jako je např. kladný a negativní výsledek děje, což je v obrázku schematicky znázorněno svislou osou po levé straně.

Obrázek zobrazuje význam předpon zjednodušeně. Více o významu kombinací daných předpon se zvratným zájmenem je popsáno např. v [3].

V následujícím přehledu uvádíme významy jednotlivých předpon v kombinaci se zvratným zájmenem.

U každé dvojice předpon (pro češtinu a pro ruštinu) uvádíme slovotvorný vzorec: X značí nějaké nedokonavé sloveso, přidáním předpony a zvratného zájmena vytvoříme sloveso „stupňované“. Uvádíme stručně význam. Rozdíl mezi češtinou a ruštinou (pokud existuje) je uveden jen velmi vágně. Každou předponu ilustrujeme příkladem z obou jazyků, ruský příklad je přeložen do češtiny. České příklady pocházejí z korpusu SYN, ruské příklady byly nalezeny pomocí vyhledávače Google na internetu.

#### *2.1.1 roz-X se, раз-X-ся*

Význam: začít X
rusky + zesílit intenzitu X

Příklady:

*Z jara se člověk nejdříve musí rozlétat.*

*Просто расслушаться надо, тогда почувствуешь*
*/= Musíš se prostě rozposlouchat, pak to pochopíš./*

#### *2.1.2 po-X se/si, по-X 0/себе*

Význam: X nějakou dobu
česky + většinou spíš v klidu, užít si to
rusky + nejčastěji relativně krátce
Poznámka: Rozdíl 0/себе je stylistický, přítomnost komponentu себе (=sobě) je neobligatorní, přidává hovorový subjektivní odstín.

Příklady:

*Pak vyslechl Aliciny kritické poznámky, pár minut si popřemítal, načež prohlásil, že všecko souhlasí.*

*Анатоль Максимович укрепился на своих ногах, еще немного пообдумывал ситуацию, а затем, очень внезапно, начал проявлять повышенную активность.*
*/=Anatol Maximovič se upevnil na svých nohou, ještě chvíli si popřemýšlel nad situací a pak najednou začal projevovat zvýšenou aktivitu./*

#### *2.1.3 za-X se/si, за-X-ся*

Význam:X po delší dobu
česky + a užít si to
rusky + déle než obvykle, příliš dlouho

Příklady:

*Loupil tady i loupežník Klempera a zaloupežil si tu i slavný lupič Babinský.*

*Избавляемся от вещичек, которые давно зависелись в нашем шкафчике!*
*/=Zbavujeme se věcí, které příliš dlouho visí (lit. zavisely si) v naší skříni!/*

#### *2.1.4 na-X se, на-X-ся*

Význam: hodně X
rusky + úplně, často spíš s kladným výsledkem

Příklady:

*Přestože profesor napsal tolik dopisů a tak na sebe upozorňoval, tolik se snažil a napředstavoval se v desítkách kanceláří, jeho stále hektičtější úsilí se bohužel nesetkalo s úspěchem.*

*Она руку привязала, чтобы не позвонить Новикову и не спросить, какими же это чудесными именами он Евгению Гордееву напредставлялся.*
*/= Uvázala si ruku, aby nezavolala Novikovovi a neřekla mu, jakými úžasnými jmény se napředstavoval Jevgeniji Gordejovovi./*

#### *2.1.5 vy-X se, вы-X-ся*

Význam: hodně X
česky + s víceméně kladným výsledkem
rusky + až do vyčerpání

Příklady:

*Francis Kennedy se tam uchýlil, aby se vytruchlil ze smrti své ženy.*

*Я много выстрадался с тех пор, как расстался с вами в Петербурге.*
*/=Hodně jsem si vytrpěl od té doby, kdy jsme se rozloučili v Petrohradě./*

#### *2.1.6 u-X se, y-X-ся*

Význam: až do vyčerpání

Příklady:

*Zemřel na těžkou bronchitidu. Prostě se ukašlal!*

*Всем смешно, а я уже весь укашлялся и усопливился...*
*/=Všichni se smějí, ale já jsem se už úplně ukašlal a usmrkal.../*

#### *2.1.7 ---, из- X-ся (jen rusky)*

Význam: velká intenzita, často s víceméně negativním výsledkem

Příklady:

*За 40 с лишним лет рисования он не изрисовался.*
*/= lit. Za 40 let malování se nevymaloval (ve smyslu nepřestal být dobrým malířem)/*

### 2.2 „Stupňovaná slovesa“ v kontextu běžné slovní zásoby

Jak je vidět z předchozího přehledu a uvedených příkladů, situace v ruštině a češtině je velice podobná. Liší se jednak produktivitou v každém z jazyků, ale hlavně odstíny významů zkoumaných předpon. To ale bude předmětem dalšího, spíše lingvisticky zaměřeného výzkumu.

Některá slova utvořená podle výše popsaného způsobu, jsou běžnou součástí jazyka. Např. slovo *rozesmát se* znamená začít se smát, vyjadřuje tedy přesně to, co uvádíme v předchozím výčtu. Podobně je na tom jeho ruský ekvivalent *рассмеяться*.

Jiné složeniny tohoto typu jsou nezvyklé a vznikají skutečně jen příležitostně (viz většina výše uvedených příkladů). Existují však i taková slova, která jsou součástí běžného slovníku, mají však jiný význam. Příkladem je *usmát se.* Běžný význam je vyjádřen v SSJČ jako „úsměvem projevit radost" (např. *Hezky se na mě usmál.*) V našem stupňovaném významu nejde o úsměv, ale o smích. Následující příklad je vymyšlený, protože vyhledat kontext, ve kterém by dané slovo vystupovalo v jiném významu, je obtížné, navíc jde o velmi řídký jev.

*Celý večer jsme se něčemu smáli, až jsme se skoro usmáli.*

# 3 Využití v automatickém zpracování jazyků

## 3.1 Morfologické slovníky a guessery

Při automatickém zpracování přirozeného jazyka je třeba nejprve rozpoznat slova textů. K tomu slouží morfologická analýza, která každému slovu přiřadí nějaké morfologické charakteristiky. Nejčastěji to bývá základní slovní tvar (lemma), potom slovní druh a případně další hodnoty morfologických kategorií, jako je rod, číslo, pád, čas apod., v závislosti na slovním druhu. Tyto hodnoty bývají většinou vyjádřeny pomocí morfologické značky (tagu). Lemmatem sloves v češtině i v ruštině (ale i ve většině ostatních jazyků) bývá infinitiv.

Mnoho jazyků má dnes k dispozici morfologický slovník, který běžná slova obsahuje, takže ho lze využít pro jejich rozpoznání. Ostatní slova, která slovník neobsahuje[1] je třeba nějak odhadnout, označit pomocí nějaké heuristiky. Takovým nástrojům se říká guesser[2]. Obecně může guesser využívat nejrůznějších pravidel odpozorovaných z textů daného jazyka. Pozorování může být jednak "ruční", potom mluvíme o pravidlových guesserech, jednak automatické, směřující ke guesserům statistickým.

Naše pozorování, popsané v předešlých sekcích, se týká spíše pravidlového přístupu. Vzhledem k velké pravidelnosti tvoření stupňovaných sloves není třeba (a vlastně ani není možné) všechna zahrnovat do slovníku. Ta, která jsou běžnou součástí jazyka, ve slovnících většinou jsou (např. již dříve uvedený příklad *rozesmát se*), ostatní se mohou velmi spolehlivě rozpoznat.

## 3.2 Homonymie běžných a stupňovaných sloves

V případech homonymie (viz příklad slovesa *usmát se*) by samozřejmě bylo nejsprávnější analyzovat slovo obojím způsobem, tedy jako běžné sloveso týkající se úsměvu, i jako nejvyšší stupeň stupňování slovesa *smát se*. Vzhledem k nízkému výskytu tohoto významu to však neděláme[3].

Přesto existují určité příznaky, které by bylo možno při rozlišování významu v takových případech použít. Ve větách se stupňovaným slovesem často nebývá výslovně vyjádřeno jeho doplnění. V našem příkladě se člověk při úsměvu usměje na něco nebo na někoho, při smíchu jako takovém sloveso *usmát se* ztrácí svůj předmět. To platí o většině sloves, která mají ve svém nestupňovaném významu nějaké rozvití (ať už předmět, nebo příslovečné určení). Příkladem může být sloveso *vyhrabat se*. Ve svém běžném významu je toto zvratné sloveso rozvito nejčastěji nějakým příslovečným určením, jako v příkladě z korpusu SYN: *Vyhrabu se z peřin.* Jako stupňované sloveso by kontext mohl vypadat např. takto (příklad je opět vymyšlený): *Holčička se hrabala v písku celé odpoledne. Když se do sytosti vyhrabala, šla si hrát s panenkou.*

Rozpoznání stupňovaných sloves může usnadnit další intenzifikátor ve větě, jako *do sytosti, dosyta, úplně, k smrti, do (úplného) vyčerpání* a podobně, v ruštině *совсем, досыта, до смерти*.

## 3.3 Lemma stupňovaného slovesa

Na první pohled je otázka lemmatizace jednoduchá. Přidáním předpony vzniklo nové slovo, přiřaďme mu tedy jeho infinitiv i s předponou, stejně jako ostatním slovesům. Vzhledem k vysoké pravidelnosti se však na vytvoření stupňovaného slovesa můžeme dívat nikoli jako na slovotvorbu, jak by bylo z pohledu přísné lingvistiky přirozené, nýbrž jako na morfologii, čili na operaci tvaroslovnou[4]. Uvedené předpony totiž v těchto případech pouze modifikují význam základního slovesa, jak bylo ukázáno v přehledu výše.

Z toho důvodu nám připadá přirozenější zvolit jako lemma základní sloveso bez předpony. Tedy např. u slovesa *rozsténat se* by mělo být lemmatem základní sloveso *sténat*, nikoli *rozsténat*. Podobně pro ruštinu.

Kromě toho by se tím usnadnilo rozlišení mezi homonymními slovesy, jejichž příklady jsme uvedli v předchozí sekci. Ve zmíněném příkladu homonymie slovesa *usmát se* bychom dostali dvě lemmata – *smát se* pro stupňovaný význam, *usmát se* pro sloveso odkazující k úsměvu.

Toto značkování se také snáze využije při následných procesech automatického zpracování textů. Např. při automatickém překladu je velmi málo pravděpodobné, že se stupňované sloveso vyskytne v překladovém slovníku, neboť, jak jsme upozornili výše, jde většinou o příležitostná slova. Podle předpony se potom mohou vytvořit pravidla pro překlad. Např. česká předpona *roz-* při překladu do angličtiny by mohla být nahrazena výrazem *start (begin)to* + lemma.

Budeme-li tedy chtít aplikovat uvedené pravidlo na sloveso *rozsténat*, překladový slovník nám přeloží jeho lemma – sténat – moan, a dostaneme překlad *start (begin) to moan*:

*James Stidham, který to vše sledoval z nosítek, se tiše rozsténal hrůzou.*

*James Stidham, who watched all this from a stretcher, began to moan quietly with horror.*

Podobná pravidla by se dala vytvořit i pro ostatní uvedené předpony.

[1] Používá se zkratka OOV – out of vocabulary words

[2] Přes velké úsilí se nám nepodařilo zvolit vhodný český termín.

[3] Homonymie většinou odradí mluvčí od použití stupňovaného slovesa. Pokud se takové užití přesto vyskytne, bude to spíše v mluveném projevu.

[4] Podobné spory se vedou o zařazení stupňování přídavných jmen a příslovcí. Většinou se ale považuje za součást morfologie, i když některé vlastnosti jsou slovotvorné. Blíže viz práce [4], která pro nás byla inspirací pro termín „stupňování sloves".

### 3.4 Poznámky k morfologické analýze stupňovaných sloves

Právě představené poznatky o „stupňování sloves“ jsou zatím jen rozpracovaným tématem, nicméně tzv. předponový guesser už se v české morfologii používá, a to v projektu Morfo (viz [7]). Zde však ještě nebyla implementována lemmatizace tak, jak je popsaná v předchozí sekci. Slovesa, jakož i ostatní slova, jsou lemmatizována i se svými předponami.

Rozpoznávání stupňovaných sloves by bylo vhodné u všech sloves s předponami, aby se v případech homonymie mohlo rozhodnout, o které sloveso se jedná – zda má být předpona součástí lemmatu, či nikoli. Jelikož se ale stupňovaná slovesa objevují v textech jen zřídka, budou se zpočátku analyzovat jen ta slovesa, která nerozpozná morfologický analyzátor využívající morfologický slovník. Vzhledem k řídkosti výskytu je riziko chyby poměrně malé.

Kvůli malému počtu výskytů nemůžeme ani uvést žádné číselné charakteristiky zlepšení úspěšnosti morfologické analýzy, byly by zanedbatelné. Zpřesnění morfologických a slovotvorných charakteristik však vždy může přispět k přesnějšímu zpracování jazyka.

## 4 Závěr

V tomto článku jsme upozornili na existenci řady produktivních slovotvorných vzorců skládajících se z předpony (*roz-, po-, za-, na-, vy-, u-* v ruštině také *z-*) a zvratného zájmena. Na příkladech z češtiny a ruštiny jsme ukázali, že při aplikaci na téměř libovolné nedokonavé sloveso tyto vzorce mění jeho význam vždy podobným způsobem. Tato vlastnost má však zatím jenom charakter pozorování. Pro efektivní použití v NLP je třeba se podrobně věnovat některým formálně gramatickým i sémantickým vlastnostem těchto konstrukcí.

Z gramatických rysů je zajímavá tranzitivita. Zdá se, že při stupňování tranzitivního slovesa se jeho objekt generalizuje (srov. *читать /=číst/ - учитаться /=učíst se/*) a ve větě se většinou nevyjadřuje.

Ze sémantických rysů pravděpodobnost výskytů daných slovotvorných vzorců zvětšuje existence životního aktora, který je schopen daný děj řídit. U různých typů sloves mají analyzované slovotvorné vzorce různou míru produktivity. Také samotné přepony nejsou vždy stejně produktivní v češtině a ruštině, což může ovlivnit pravděpodobnost výskytu „našich“ významů při vyhledávání.

V dalším výzkumu se chceme věnovat i dalším jazykům, zejména slovanským, kde předpokládáme také existenci „stupňovaných sloves“.

## Poděkování

Práce na tomto příspěvku byla podpořena z prostředků grantů GAČR č. P406/10/0875 a P406/12/0658.

## Literatura

1. Ruský národní korpus, www.ruscorpora.ru
2. Český národní korpus SYN, www.ucnk.ff.cuni.cz
3. Малый академический словарь, МАС. Словарь русского языка в 4-х томах (М., Русский язык, 1999. Т. 1-4.
4. P. Karlík, Z. Hladká:. *Kam s ním? (Problém stupňování adjektiv).* In Život s morfémy. Sborník studií na počest Zdenky Rusínové, s. 73-93. Brno, MU v Brně, 2004.
5. Slovník spisovného jazyka českého. Praha: Academia, 1989
6. J. Hlaváčová: *Formalizace systému české morfologie s ohledem na automatické zpracování českých textů*. Disertační práce. Praha 2009.
7. J. Hlaváčová , D. Kolovratník: *Morfologie češtiny znovu a lépe*. In: Informačné Technológie – Aplikácie a Teória. Zborník príspevkov, ITAT 2008, Copyright © PONT s.r.o., Seňa, Slovakia, ISBN 978-80-969184-8-5, pp. 43-47, 2008

# Tvaroslovník – databáza tvarov slov slovenského jazyka*

Stanislav Krajči and Róbert Novotný

Ústav informatiky, Prírodovedecká fakulta, UPJŠ Košice,
stanislav.krajci@upjs.sk, robert.novotny@upjs.sk

**Abstrakt** *Prezentujeme databázu tvarov slov slovenského jazyka obsahujúcu cca 30 miliónov riadkov, z čoho cca 320 tisíc sú základné tvary slov zo Slovníka slovenského jazyka a Veľkého slovníka cudzích slov. Zároveň navrhujeme jednoduchý systém na prácu s týmito dátami a prezentujeme výkonnostné testy pri typických operáciách s týmito dátami. Systém v kombinácii s databázou možno efektívne používať na lematizáciu slov i na vyhľadávanie všetkých tvarov daného slova.*

## 1 Úvod

Používanie prirodzeného jazyka je jedným z najdôležitejších atribútov ľudstva, ktoré ho podstatne odlišujú od ostatných živých tvorov. Je totiž prejavom myslenia, odráža jeho štruktúru a, čo je ešte dôležitejšie, spätne ho významne ovplyvňuje, ba až podmieňuje. Jeho dôkladné spracovanie je tak neustálou výzvou, a stalo sa preto významnou súčasťou informatiky.

Dnešnou „lingua franca“ je (dokedy?) angličtina, ktorej výskum je (hlavne preto) značne preskúmaný a prepracovaný. Výraznou pomôckou je pritom jeho jednoduchá morfologická štruktúra – slovotvorbu tu možno vykonať pomerne jednoducho zo spoločného slovného základu. Jeho nájdenie je už takpovediac štandardizované v pomerne jednoduchom, ale efektívnom Porterovom algoritme [9], ktorého podstatou je odstránenie niekoľkých prípon (napríklad „-ed“, „-ing“, či „-s“). Takýto postup je potom výraznou pomôckou napríklad pri vyhľadávaní textových dokumentov podľa kľúčových slov.

V prípade slovenčiny je však morfológia podstatne komplikovanejšia. Tá totiž na rozdiel od angličtiny patrí k tzv. flexívnym jazykom, kde môže mať slovo značný počet tvarov, čo je však horšie, pravidlá ich tvorby sú neporovnateľne zložitejšie a je v podstate nemožné zachytiť ich do akéhokoľvek efektívneho algoritmu. Klasický školský prístup používať niekoľko málo tzv. vzorov („chlap“, „hrdina“, „dub“, „stroj“, ...) je preto iba veľmi slabou aproximáciou riešenia tejto úlohy, ktorá je porušovaná neúmerným množstvom výnimiek.

K hľadaniu tvarov slova (či už základného, alebo všetkých) však existuje alternatíva. Tá spočíva v *jednorazovom získaní všetkých tvarov* (rozumnej množiny) slov a ich uložení do databázy. Takéto získanie je, ako neskôr uvidíme, značne prácne a časovo náročné, jeho výhodou však je, že ho netreba robiť druhý raz.

Pri takomto prístupe si treba uvedomiť jednu dôležitú vec: Jazyk je takpovediac živý organizmus, lebo každodenný život prináša potrebu nových slov. Tempo ich vzniku však nie je natoľko výrazné, aby takúto databázu nebolo možné raz za (primeraný) čas aktualizovať. Navyše tu možno pozorovať zaujímavý fakt, že čím novšie je slovo, tým sú jeho tvary pravidelnejšie – nepravidelnosť ohýbania slova svedčí o jeho starodávnosti jeho pôvodu. Nové slová, často preberané z iných jazykov, sú preto buď neohybné, alebo ich používatelia poslovenčia tým, že pri ich ohýbaní na nich aplikujú už existujúce koncovky z iných slov. Toto pozorovanie tak umožňuje efektívne spracovať i slová ešte nezaradené do databázy. (Príslušný jednoduchý algoritmus popíšeme nižšie.)

Téma spracovania prirodzeného – slovenského – jazyka nás dlhodobo zaujíma. Po zvážení všetkých aspektov sme sa napriek tušeným problémom rozhodli vyrobiť spomínanú databázu tvarov slovenských slov, ktorú sme priliehavo nazvali Tvaroslovník. S týmto cieľom prebehla na Ústave informatiky UPJŠ v Košiciach v poslednom desaťročí elektronizácia Slovníka slovenského jazyka [4] i Veľkého slovníka cudzích slov [14]. V tejto práci sme ďalej pokračovali v rámci úspešného projektu NAZOU [10], ale i dlho po jeho ukončení. Získali sme tak zoznam okolo 320 000 doteraz oficiálnych slovenských slov s celkovým počtom zhruba 30 miliónov tvarov. Ako ukážeme, napriek tomuto množstvu dát nie je rýchlosť práce s touto databáze márna.

Netajíme, že sme pri svojej práce boli (či už pozitívne, alebo negatívne) inšpirovaní i ďalšími pokusmi spracovávať slovenčinu. Spomeňme tu prístup Emila Páleša [8], založený skôr na hľadaní pravidiel slovotvorby, I-spell s manuálnym značkovaním jednotlivých lexém [3], či na vyhľadávaní tvarov s minimálnou Levenshteinovou vzdialenosťou [2]. Prehľad existujúcich prístupov v počítačovej lingvistike je zhrnutý napr. v [1] a [7].

* Táto práca je čiastočne podporená grantom VEGA (1/0832/12), grantom APVV (APVV-0035-10) a projektom CaKS (ESF, č. 008/2009/2.1/OP VaV, 262201-20007).

## 2 Proces tvorby Tvaroslovníka

Projekt tvorby Tvaroslovníka bol iniciovaný myšlienkou prof. Vojtáša. Predprípravou bolo, samozrejme, získanie zoznamu slov z fyzických zdrojov. Ručné spracovanie vzhľadom na rozsah dát (už len Slovník slovenského jazyka [14] má šesť pomerne hrubých zväzkov) neprichádzalo do úvahy, rozhodli sme sa preto pre digitalizáciu zdrojov pomocou OCR softvéru ABBYY [11]. Desiatky našich študentov (nie úplne nezištne) tak vytvorili textové súbory s obsahom ako na obrázku:

```
31101  @extravaganza@ [-nca] -y ž. /tal./ umel. výpra
31102
31103  @extravaskulárny@ /lat./ lek. položený, ležiac
31104
31105  @extravazácia@ -ie ž. /lat./ lek. vystúpenie t
31106
31107  @extravazát@ -u m. /lat./ lek. telová kvapalin
31108
31109  @extravertovaný@ /lat./ -> extrovertný
31110
31111  @extraverzia@ -ie ž. /lat./ -> extroverzia
31112
31113  @extravilán@ -u m. /lat./ stav. oblasť, územie
31114
31115  @extrém@ -u m. /lat./ 1. krajnosť, výstrednosť
31116
31117  @extrémista@ -u m. (@extrémistka@ -y ž.) /lat.
31118
31119  @extrémistický@ /lat./ odb. krajný, výstredný,
31120
31121  @extremit@ -u m. /lat./ meteor. extrémna, t. j
31122
31123  @extremitas@ ž. /lat./ anat. končatina ako cel
31124
31125  @extrémizmus@ -mu m. i @extrémnosť@ -ti ž. /la
31126
31127  @extrémny@ /lat./ 1. výstredný, krajný 2. lek.
31128
31129  @extrinzitný@ /lat./ odb. nevlastný
31130
31131  @extrofia@ -ie ž. /gréc./ lek. chybné uloženie
```

(Treba však povedať, že nie každý účastník tohto procesu k svojej práci pristúpil dostatočne zodpovedne, čím vzniklo množstvo chýb (zlé rozpoznanie textu, zlé vyznačenie hesiel, dokonca chýbajúce strany), z ktorých mnohé doteraz nie sú opravené.)

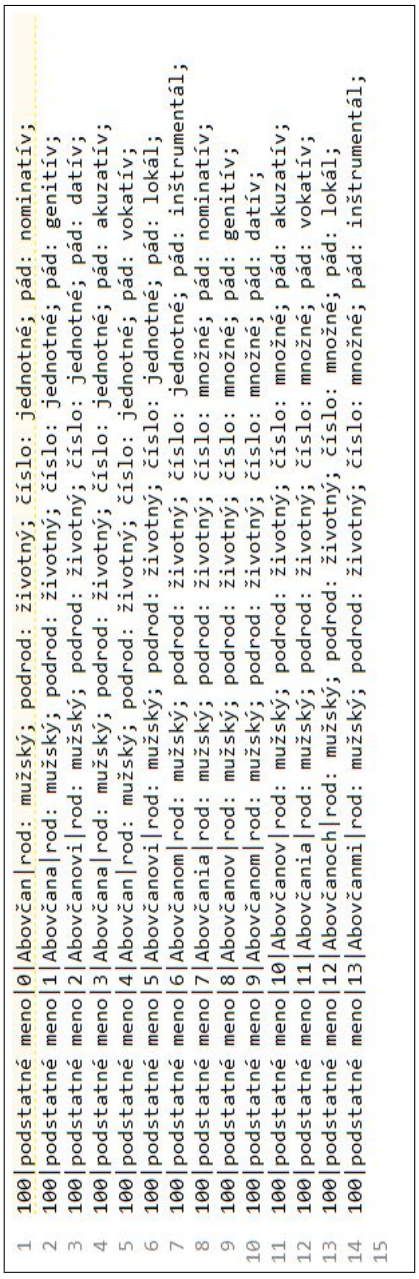
Po ukončení tejto prípravy sme z týchto textových súborov získali zoznam slov, v drvivej väčšine prípadov i ich slovných druhov a prípadne ďalších pre ohýbanie dôležitých informácií. (Cieľ takto získať aj ich koncovky sa vzhľadom na nedokonalosť a značnú chybovosť OCR procesu, žiaľ, nepodaril.) Získané slová sme potom uložili do tabuľky relačnej databázy. Úloha tým však nebola ani zďaleka splnená, veď v slovníkoch sú len slová v základnom tvare.

Pustili sme sa teda do vytvárania ostatných tvarov. Ako sme už uviedli, dáta sme mali vo forme textových súborov, naším cieľom ich však bolo umiestniť do relačnej databázy. S dátami sme teda potrebovali pracovať v týchto dvoch módoch. Aby sme sa vyhli zbytočným technickým komplikáciám, vytýčili sme si tieto zásady:

- Kvôli jednoduchosti práce i cenovej dostupnosti budeme pracovať s jednoduchým, ale pre naše potreby dostatočne silným systémom MySQL [12].
- Vzhľadom na očakávaný vysoký počet slovných tvarov nebudeme používať SQL príkaz `INSERT`, ale omnoho rýchlejšiu MySQL utilitu `load`. To však znamená, že štruktúra takto importovaných súborov musí zodpovedať štruktúre databázy. Jej dátová časť teda bude pozostávať z jednej tabuľky, ktorá bude mať takéto stĺpce (prípadne v inom poradí):
  - `idSlovo` – jedinečný identifikátor slova,
  - `idTvar` – jedinečný identifikátor tvaru v rámci daného slova (pričom základný tvar bude mať pevné číslo, napr. `0`),
  - `tvar` – samotný textový tvar slova,
  - `slovnýDruh` – slovný druh (v prípade podstatných mien včítane rodu),
  - `charakteristika` – zoznam hodnôt relevantných gramatických kategorií.

  Primárny kľúč bude pritom tvorený dvojicou stĺpcov `idSlovo` a `idTvar`. Pripúšťame teda, samozrejme, že niektoré slová majú rovnaký tvar.
  (Pri stĺpci `charakteristika` sme si, pravdaže, vedomí zrejmého porušenia zásady zvanej prvá normálna forma. Na druhej strane je však zrejmé, že po konečnom umiestnení všetkých potrebných dát nebude problémom preusporiadať ich tak, aby bol tento princíp dodržaný.)
- Bolo by naivné očakávať, že sa proces vytvárania tvarov daného slova vydarí vždy na prvý pokus. V negatívnom prípade to znamená opravu či doplnenie tvarov tohto slova. Vzhľadom na ťažkopádnosť takýchto úprav ich nebudeme robiť v databáze, ale všetky údaje o slove exportujeme do textového súboru, ktorý ľahko upravíme, a potom údaje z neho opätovne importujeme do databázy. To však implikuje, že najlepšie bude udržiavať pre každé slovo samostatný súbor, ktorý bude obsahovať všetky jeho tvary. Súbory tak budú mať primeranú veľkosť, s ktorou textový editor nebude mať žiadne problémy, a navyše ich budeme môcť prehľadne značiť v tvare `<idSlovo>-<tvar>.txt`, čo nám značne uľahčí orientáciu v súborovom systéme.
  Vzhľadom na kvality používaného textového editora Textpad [13] pritom nestratíme ani možnosť hromadnej opravy prípadných viacnásobne sa vyskytujúcich chýb.

Typický súbor, s ktorým takýmto spôsobom pracujeme, vyzerá takto:



```
100|podstatné meno|0|Abovčan|rod: mužský; podrod: životný; číslo: jednotné; pád: nominatív;
100|podstatné meno|1|Abovčana|rod: mužský; podrod: životný; číslo: jednotné; pád: genitív;
100|podstatné meno|2|Abovčanovi|rod: mužský; podrod: životný; číslo: jednotné; pád: datív;
100|podstatné meno|3|Abovčana|rod: mužský; podrod: životný; číslo: jednotné; pád: akuzatív;
100|podstatné meno|4|Abovčan|rod: mužský; podrod: životný; číslo: jednotné; pád: vokatív;
100|podstatné meno|5|Abovčanovi|rod: mužský; podrod: životný; číslo: jednotné; pád: lokál;
100|podstatné meno|6|Abovčanom|rod: mužský; podrod: životný; číslo: jednotné; pád: inštrumentál;
100|podstatné meno|7|Abovčania|rod: mužský; podrod: životný; číslo: množné; pád: nominatív;
100|podstatné meno|8|Abovčanov|rod: mužský; podrod: životný; číslo: množné; pád: genitív;
100|podstatné meno|9|Abovčanom|rod: mužský; podrod: životný; číslo: množné; pád: datív;
100|podstatné meno|10|Abovčanov|rod: mužský; podrod: životný; číslo: množné; pád: akuzatív;
100|podstatné meno|11|Abovčania|rod: mužský; podrod: životný; číslo: množné; pád: vokatív;
100|podstatné meno|12|Abovčanoch|rod: mužský; podrod: životný; číslo: množné; pád: lokál;
100|podstatné meno|13|Abovčanmi|rod: mužský; podrod: životný; číslo: množné; pád: inštrumentál;
```

Opäť si uvedomili, že popri slovnom druhu a v prípade postatných mien hlavne rodu je najdôležitejším (i keď určite nie jediným) faktorom pri ohýbaní slova jeho koniec. Slová jednotlivých ohybných slovných druhov sme preto zoradili retrográdne a roztriedili ich podľa koncoviek (vzhľadom na rytmický zákon však bolo často treba uvažovať až predposlednú slabiku). Vznikli tak niektoré veľké skupiny, pri ktorých bolo možné aplikovať pravidelné ohýbanie, a tak slová v nich bolo možno vybaviť naraz. Takéto hromadné spracovanie potom prebiehalo podľa nasledujúceho postupu:

1. Zo skupiny sme vybrali typického reprezentanta.
2. Jeho súbor sme ručne upravili tak, aby obsahoval všetky (správne očíslované) požadované tvary príslušného slova.
3. Tento súbor sme importovali do databázy (nevyhnutnou súčasťou importu je, samozrejme, vymazanie starých záznamov).
4. Príslušné koncovky sme aplikovali na všetky ostatné slová z tejto skupiny určenej slovným druhom, rodom a koncovkou podľa nižšie uvedeného algoritmu, a automaticky sme tak vygenerovali ich súbory.
5. Aby sme sa vyhli prípadnej systematickej chybe, takto vygenerované súbory (alebo aspoň ich typických reprezentantov) sme opticky prezreli.
   - Ak chyba naozaj nastala, upravili sme množinu slov, na ktorú treba algoritmus aplikovať, a postup sme zopakovali.
   - V opačnom prípade sme dáta z týchto vygenerovaných súborov importovali do databázy.

Algoritmus používaný v kroku 4 je malou modifikáciou postupu prevzatého z našej práce [6] a mohli by sme nazvať *podvojná výmena*. Vyzerá takto:

- neznáme slovo: $X = \texttt{ponuka}$
- koncovka: $K = \texttt{uka}$
- známe slovo (tzv. predloha): $P = \texttt{ruka}$
- začiatok predlohy: $\overline{P} = \texttt{r}$ (teda $P = \overline{P} + K$)
- začiatok slova: $\overline{X} = \texttt{pon}$ (teda $X = \overline{X} + K$)
- ohnutý tvar predlohy: $P' = \texttt{rúk}$
- koniec ohnutého tvaru predlohy: $K' = \texttt{úk}$ (teda $P' = \overline{P} + K'$)
- ohnutý tvar slova: $X' = \overline{X} + K' = \texttt{ponúk}$

Dôležité je si tu uvedomiť, že všetky gramatické kategórie ohnutého slova $X'$ sa zhodujú s gramatickými kategóriami slova $P'$, ktoré už poznáme.

Takto sme teda viac-menej automaticky získali tvary pomerne veľkých skupín slov. Tým sme však, žiaľ, ani zďaleka nevyčerpali všetky slová. Keďže zvyšné skupiny boli natoľko malé, že akýkoľvek pokus o ich aspoň čiastočné automatické spracovanie by bol skôr kontraproduktívny, ostávalo jediné – prebrať ich ručne...

## 3 Typy a počet údajov v Tvaroslovníku

V Tvaroslovníku sú pre hlavné slovné druhy evidované takéto typy údajov:

- podstatné mená:
  - rod (mužský, ženský, stredný)
  - (v prípade mužského rodu) podrod (životné, neživotné)
  - číslo (jednotné, množné, alebo pomnožné)
  - pád (nominatív, genitív, datív, akuzatív, vokatív, lokál, inštrumentál)

- prídavné mená:
  - rod (a prípadne podrod)
  - číslo
  - pád
  - stupeň (prvý, druhý, tretí)
- sloveso (pri všetkých formách aj zvratnosť (sa, si, –) a negácia (ne-, –)):
  - neurčitok
  - prítomný čas:
    - rod (mužský, ženský, stredný)
    - číslo (jednotné, množné)

    minulý čas:
    - osoba (prvá, druhá, tretia)
    - číslo (jednotné, množné)

    rozkazovací spôsob:
    - osoba
    - číslo
  - prechodník
  - činné príčastie:
    - rod (a prípadne podrod)
    - číslo
    - pád
  - trpné príčastie:
    - rod (a prípadne podrod)
    - číslo
    - pád
  - minulé príčastie:
    - rod (a prípadne podrod)
    - číslo
    - pád
  - slovesné podstatné mená:
    - rod
    - (v prípade mužského rodu) podrod
    - číslo
    - pád
- príslovky:
  - stupeň (prvý, druhý, tretí)

Ostatné slovné druhy (zatiaľ) väčšinou nemajú uvedené žiadne kategórie.

Ako sme už spomenuli, Tvaroslovník obsahuje okolo 30 miliónov tvarov približne 320 000 doteraz oficiálnych slovenských slov. Na jedno slovo tak pripadá priemerne približne 100 tvarov. Toto číslo sa možno na prvý pohľad zdá prekvapivo veľké, keďže napríklad podstatné meno má, ako vieme, obvykle len 14 tvarov (sedem pádov v dvoch číslach). Uvedomme si však, že temer každé prídavné meno prináša 168 tvarov (7 pádov, 2 čísla, 4 rody (pri mužskom sú totiž životný a neživotný podrod) a 3 stupne), a sloveso dokonca 394 tvarov.

## 4 Testovanie Tvaroslovníka

Výsledok našej práce sme obalili do jednoduchého internetového programu, ktorý je umiestnený na adrese `http://158.197.31.218:8080/slovnik/`, respektíve `http://tvaroslovnik.ics.upjs.sk/`. Vie plniť tieto tri požiadavky (a to podľa potreby s diakritikou i bez nej):

- vyhľadanie všetkých výskytov daného slova (včítane ich gramatických kategórií),
- vyhľadanie všetkých tvarov daného slova (včítane ich gramatických kategórií),
- vyhľadanie základného tvaru daného tvaru slova.

Vzhľadom na statickú povahu dát sme zvolili databázový stroj MyISAM, v ktorom môžeme s výhodou využiť zabudovanú možnosť fulltextového vyhľadávania. (Z implementačného hľadiska ide o využitie `FULLTEXT` indexu na stĺpci `tvar`.) Výkonnostné experimenty sme vykonali pri štandardnom nastavení MySQL na operačnom systéme Windows 7 64bit, CPU Intel i7, 8MB RAM.

Priemerná rýchlosť pre uvedená operácie na uvedených dátach dosahovala 250 ms pre vyhľadanie tvarov slova, pričom samotné vyhľadanie trvalo 10 ms (zvyšok ide na vrub komunikácii a práci s databázovými prostriedkami). Analogické hodnoty sme dosiahli aj pre vyhľadávanie všetkých tvarov slova. Vyhľadávanie základného tvaru trvalo v priemere 145 ms.

Ďalšie experimenty sme vykonali na kolekcii 353 článkov zo servera `bulvar.sk`, ktoré obsahovali v priemere 525 slov. Priemerná rýchlosť lematizácie jednotlivých dokumentov dosahovala 132 slov za sekundu (minimum 93, maximum 145). Rýchlosť je ovplyvnená predovšetkým počtom slovných tvarov pre jednotlivé slová a tiež počtom variantov základného tvaru slova, ktoré vyplývajú z nejednoznačnosti lematizácie.

## 5 Zhrnutie, problémy a ďalšia práca

Predstavili sme náš dlhodobý projekt Tvaroslovník – databázu (takmer) všetkých tvarov (takmer) všetkých slovenských slov. Sme si, samozrejme, vedomí rezerv ukrytých za oboma slovami „takmer“:

- V procese tvorby Tvaroslovníka sa vyskytli objektívne i subjektívne ťažkosti, ktoré spôsobili, že niektoré chybné dáta ostali neopravené (domnievame sa však, že takýchto chýb je hlboko pod 1 promile). Budeme preto musieť pripraviť testy, ktorými takéto chyby, hlavné systematické, identifikujeme, a následne opravíme.

– V Tvaroslovníku nie sú zohľadnené údaje z nového Slovníka súčasného slovenského jazyka [5], ktorého dva diely z predpokladaných ôsmich už vyšli. Tvaroslovník bude preto treba doplniť. Ako sme však už naznačili, tvary nových slov sú väčšinou vytvárané pravidelne, predpokladáme preto, že proces zahrnutia takýchto slov do databázy bude môcť byť automatizovaný, a to využitím vyššie uvedeného algoritmu podvojnej výmeny.
– Je otázne, ktoré vlastné mená majú byť v slovníku a ktoré nie, špeciálne napríklad priezviská. Aj ich tvary však možno získať viac-menej automaticky opäť podvojnou výmenou. Treba „len“ získať tú správnu predlohu.
– Pri niektorých menej zastúpených slovných druhoch chýbajú gramatické kategórie. Bude ich treba doplniť. Našťastie ich početnosť nie je výrazná.
– V slovníkoch (dokonca ani v najnovšom [5]) z nepochopiteľných dôvodov nie je uvádzaná informácia o životnosti mužských (ale i ženských) podstatných mien, hoci je veľmi dôležitá ako pre ich správne skloňovanie, tak i pre existenciu privlastňovacích prídavných mien (typu „otcov“). V Tvaroslovníku sme zatiaľ zvolili extrémistický prístup, že tieto odvodené tvary pripravíme pre každé, teda aj neživotné, podstatné meno mužského a ženského rodu.

Rýchlosť práce s Tvaroslovníkom je prijateľná, možno ju však zvýšiť nasledujúcim spôsobom: Je zrejmé, že nie všetky slová či ich tvary majú rovnakú frekvenciu používania, s mnohými slovami sa bežný človek nestretne ani raz za život. Po istom čase teda možno získať primerane veľkú (lepšie povedané malú) podmnožinu často používaných (tvarov) slov a tie umiestniť do databázy s rovnakou štruktúrou. Pri lematizácii by sa potom primárne používala táto chudobnejšia databáza, materská by sa použila iba v prípade neúspechu lematizácie.

Je tiež zrejmé, že morfologická analýza nie je takýmto slovníkom uzavretá, a to pre problémy s disambiguáciou – rozlíšením, ktorý z prípadných viacerých možných základných tvarov je ten pravý.

Napriek tomu sa však domnievame, že náš Tvaroslovník môže byť výraznou pomôckou pri riešení problému morfologickej analýzy slovenského jazyka. Sme presvedčení, že ho tiež bude možné použiť i vo vyššej vrstve spracovania jazyka, a to pri vetnom rozbore. Práve ten považujeme za našu dlhodobú výzvu.

Keďže pomocou Tvaroslovníka možno združovať rôzne tvary jedného slova, môže byť efektívnou pomôckou pri plnotextovom vyhľadávaní. Ďalšou aplikáciou môže byť pomoc pri vytváraní rôznych štatistík (napríklad korpusu či valenčného slovníka). Oblúkom sa tak môžeme vrátiť k naoko zavrhnutej úlohe hľadať v jazyku pravidlá, tentoraz však podporenej dátami uloženými v Tvaroslovníku.

## Literatúra


1. M. Ciglan, M. Laclavík: *Dostupné zdroje a výzvy pre počítačové spracovanie informačných zdrojov v slovenskom jazyku.* Proceedings of 1st Workshop on Intelligent and Knowledge oriented Technologies (WIKT 2006).
2. R. Garabík: *Slovak morphology analyzer based on Levenshtein edit operations.* Proceedings of 1st Workshop on Intelligent and Knowledge oriented Technologies (WIKT 2006).
3. ISpell. An interactive spell-checking program for Unix. [online] `http://ispell.hq.alert.sk/`
4. kol.: *Slovník slovenského jazyka.* Vydavateľstvo SAV, Bratislava, 1959–1968.
5. kol.: *Slovník súčasného slovenského jazyka.* Vydavateľstvo SAV, Bratislava, 2006–
6. S. Krajči, R. Novotný, L. Turlíková: *Použitie lematizácie vo fulltextovom vyhľadávaní v slovenských dokumentoch.* 2nd Workshop on Intelligent and Knowledge oriented Technologies, F. Babič, J. Paralič (eds.), November 2007, Košice, Slovakia, 147—152, [ISBN 978-80-89284-10-8].
7. S. Krajči, R. Novotný, L. Turlíková, M. Laclavík: *The tool Morphonary/Tvaroslovník: Using of word lemmatization in processing of documents in Slovak.* In: P. Návrat, D. Chudá (eds.), Proceedings Znalosti 2009, Vydavateľstvo STU, Bratislava, 2009, 119–130
8. E. Páleš: *Sapfo, parafrázovač slovenčiny.* Veda, Bratislava, 1994. ISBN 80-224-0109-9.
9. M. F. Porter: *An algorithm for suffix-stripping.* Program 14 (3), 1980, 130–137.
10. projekt NAZOU – `http://nazou.fiit.stuba.sk/home/index.php`
11. software ABBYY – `http://www.sk.abbyy.com/`
12. software MySQL – `http://www.mysql.com/`
13. software TextPad – `http://www.textpad.com/`
14. S. Šaling, M. Ivanová-Šalingová, Z. Maníková: *Veľký slovník cudzích slov.* SAMO, 2003

# Using noisy GPS for good localization on a graph

David Obdržálek, Ondřej Pilát
Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic

**Abstract.** *In this work, we present a method how to handle noisy GPS signal for good localization of an autonomous robot travelling along pathways in a park. One of the requirements on the robot's behaviour is it shall not leave the pathways and "step" on the grass. That is maintained by its internal local control system, but global localization is needed as well for the robot to be able to navigate and reach the destination place. For this purpose, the localization problem is addressed by an implementation of the Monte Carlo Localization method with connection to a graph-based map. We show that our method serves as an appropriate and robust solution for this problem even in situations when the GPS signal is not good or if the robot leaves the pathway despite the requirements.*

## 1 Introduction

Once, robots were used only in closed environments like factories and other well-protected areas. However, robots started recently to emerge from such concealed places and immerse in the "human world". Such examples include transport carts in buildings where they transport stuff through the same corridors as people walk on (e.g. in the Forth Valley Royal Hospital (UK) or in the Fakultní nemocnice Motol (CZ) [1]) or autonomous cars running on regular streets together with human-controlled vehicles (e.g. the currently popular "Google Car" [2]).

In all these cases, their authors have to deal with number of theoretical as well as practical problems; one of such problem is the localization and navigation of these robots. For indoor areas, it is often possible to adapt the environment so that the localization problem is easier to address (e.g. using passive or active beacons, creating easily detectable artificial landmarks etc. [3]). In outdoor areas, the localization techniques are more difficult, but since the satellite localization is publicly and freely available and covers the whole Earth, the robot outdoor localization can be solved efficiently too despite imprecise primary (input) data.

In this paper, we show the usage of a cheap GPS localization module (device) on an autonomous robot running in a publicly accessible park. Data from such module is filtered and further used as one of the inputs for the Monte Carlo Localization (MCL) [8,9]. The system then localizes the robot on a vector map (from the OpenStreetMap project [4]) and provides the control system of the robot sufficiently precise information about its position even if the GPS source data mistakenly indicate the robot is off the pathway or if on the other hand the robot really leaves the pathway.

The following text is organized as follows: In Section 2, we outline the backgrounds for this project. Section 3 gives short introduction to Monte Carlo Localization, Section 4 describes the OpenStreetMap data, which was used in this project. Section 5 explains some general aspects of using vector-map based localization in connection with MCL and GPS and Section 6 briefly explains the implementation. Section 7 concludes the work by showing some results of the localization on real data gathered during the Robotour 2011 contest and giving some ideas for future work.

## 2 Backgrounds and setup

This project follows a project of an autonomous robot for the Robotour contest [5]: the task of an autonomous robot is to travel in a public place (a park) from one given place to another. The robot shall travel only using the pathways and shall not leave it ("Do not step on the grass!" rule).

Practical experiences from observing this robot and several others during the 2011 and earlier editions of Robotour have confirmed the expectation that standard GPS cannot be straightforwardly used for robot localization. Even the theoretical precision of GPS is on the edge of usability, but practically such precision is never reached, and it is not rare that GPS error rises to tens of metres or higher or the GPS fix gets lost at all. Local real conditions prevent reaching good precision nearly at all times. The most important sources of error in GPS measurement in a park are the obstacles in the direct line of sight towards the localization satellites – the trees, and indented terrain that limits the number of satellites detectable on the visible part of the sky. Other factors like the weather, solar activity, multipath effect, actual satellite geometry, EMI etc. contribute too [6], but these two mentioned are the gravest.

Having good experiences with Monte Carlo Localization indoors [7], we have decided to study the possibility of connecting GPS as one of the sources of Monte Carlo Localization also for outdoor use with special requirements as mentioned earlier in this section.

## 3 Monte Carlo Localization

Monte Carlo Localization (MCL) [8,9] is currently one of the widestly used methods for handling noised data in the localization process. Being one of the probabilistic localization methods, MCL does not deal with *exact positions* but with *beliefs about positions*. The basic principle is as follows:

The MCL tracks bigger number of the *particles* – possible robot states (in our case its position as coordinates and rotation in the space). The probability that the particle mirrors the actual robot state is represented by the *particle weight*. The MCL repeatedly maintains this set by updating the particles and their weights. Based on the intended movement of the robot, all particles in the set are updated (moved) in the *Prediction phase*, and based on other measurements, their weights are updated too, reflecting the cor-

respondence between the particle state and the measurement in the *Measurement Phase*. For the long-time stability, usability and success of the algorithm, additional management steps are taken during the MCL work in the *Resampling phase*. Besides other impacts, these additional steps help to find out the robot position from scratch or to keep the set valid even in the case of "kidnapping" the robot, i.e. forcibly changing its position by other means than its regular movement.

During the position update step, random noise is added to the particles. This apparent deterioration helps to overcome the fact that the real move of the robot may be different to the intended move due to random and/or systematic errors (mechanical, electrical and others). By adding the noise, the algorithm gets less prone to such problems.

The MCL algorithm outline is:

```
1: For every x_i in S do
2:     set x'_i = x_i moved based on u_t
3:     set w'_i = valued x_i based on z_t
       and M
4: For every x'_i do
5:     insert x'_i into S' based on
       probability w'_i
6: Add random particles to S'
```

where $x_i$ is a particle (position or state of the robot), $S$ the current particle set, $x'_i$ the new particle, $u_t$ the move in time step t, $z_t$ the measurement after the move, $M$ the map and $S'$ the new particle set.

At the beginning, the particle set needs to be initialized. If we know the position of the robot, we can set the initial set $S$ to contain just a single particle representing that place and state and assigning it the maximum weight (i.e. the probability = 1).

If we do not know the position of the robot, the initial set may be created by preparing a uniformly distributed set of N particles, which all have the same weight (i.e. the probability of each particle is 1/N). The space, over which the set is generated, does not have to be identical with the free-space area covered by the graph. Such situation is shown on Figure 1 – the particles are generated only on the graph edges.

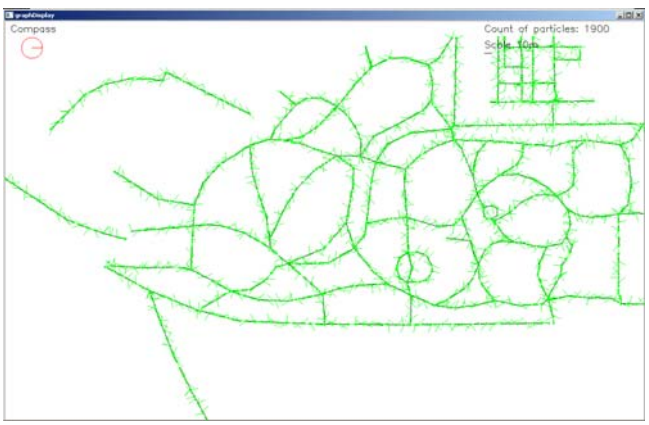



**Fig. 1.** An example of a particle set just after the initialization (Türkenschanzpark in Vienna, Austria).

## 4 Open Street Map

The OpenStreetMap Project [4] maintains and develops a vector map of the world; its main goal is to provide this map for free and with the possibility to edit it and extend it. The map format is a topological data structure with four basic elements:

- *Node* – a point with known position.
- *Way* – an ordered list of nodes forming a closed polyline, open polyline or area.
- *Relation* – a group of elements with certain properties; may be recursive.
- *Tag* – name & value pair of additional information (metadata).

There are several data file formats used for OSM data:

- *OSM XML* – XML format,
- *OSM Binary* – binary format,
- *PBF* – optimized binary format,
- *o5m* – xml-structured data with PB coding,

and others.

For our purposes, the XML structured data was the best for its simplicity and readability.

For our project, we needed to acquire a vector map of pathways in a particular park (the Türkenschanzpark in Vienna where Robotour 2011 was held). The OSM provides adequate data and tools for this task: A pathway in the Türkenschanzpark is represented in OSM as a *Way* element with a *Tag* attribute featuring a key *highway* with the value of *footway*. Such *Ways* contain references to *Nodes*, which form the pathway shape. The *Nodes* have attributes describing their position (*lat* for latitude and *lon* for longitude) which are used by our project, and number of others, which are not important for us.

Example of a small area covered by Open Street Map and corresponding data structures is given on Figures 2, 3 and 4.


**Fig. 2.** An example of an Open Street Map (Riegrovy sady, Prague, Czech Republic).

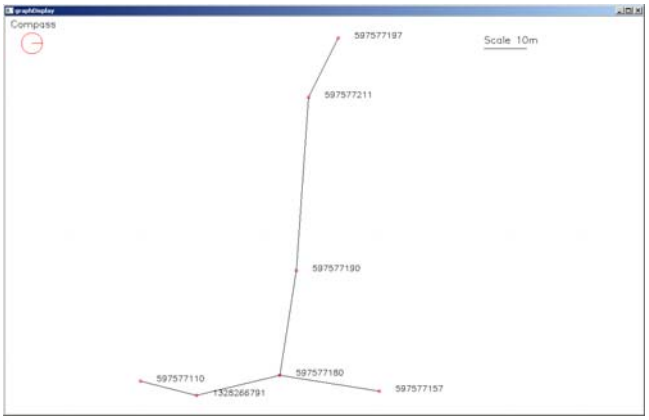



**Fig. 3.** Extracted graph map corresponding to the highlighted area on Figure 2.

```
<node id="597577110" lat="50.0800108" lon="14.4398193"/>
<node id="597577157" lat="50.0799893" lon="14.4405980"/>
<node id="597577180" lat="50.0800229" lon="14.4402738"/>
<node id="597577190" lat="50.0802411" lon="14.4403271"/>
<node id="597577197" lat="50.0807300" lon="14.4404646"/>
<node id="597577211" lat="50.0806052" lon="14.4403671"/>
<node id="1328266791" lat="50.0799797" lon="14.4400023"/>
<way id="46754360">
    <nd ref="597577180"/>
    <nd ref="597577190"/>
    <nd ref="597577211"/>
    <nd ref="597577197"/>
    <tag k="highway" v="footway"/>
</way>
    <way id="46754409">
    <nd ref="597577157"/>
    <nd ref="597577180"/>
    <nd ref="1328266791"/>
    <nd ref="597577110"/>
    <tag k="highway" v="footway"/>
</way>
```

**Fig. 4.** OSM data in XML format corresponding to the highlighted area on Figure 2 and drawn on Figure 3 (only relevant elements and their attributes are shown).

Data from the OSM project can be easily transformed into a topological graph: an OSM *Node* may represent a graph node and an OSM *Way* may define the edges (the edge connects the two adjacent nodes). As can be seen, there could be many nodes with the degree of 2. For the localization, the shape of the curve and absolute distances are important, therefore no optimization (e.g. number of nodes reduction) of the graph is done. On the other hand, from the path-planning or navigation point of view nodes with degree 2 are not interesting so an optimized graph may be easily created by systematic extraction of such nodes or by marking the nodes and using only a subset of the graph for navigation.

## 5 Vector map localization

One of the basic questions in implementing MCL on a vector map is how the noise should be applied during the MCL Prediction phase. There are several possibilities: firstly, it is possible to adapt the new particle position so that it lies again on the graph edge. Secondly, the particles could be freely placed in the map area without snapping to the graph edges while their relation to graph edges is address in later phases of MCL. That can be done during the Measurement phase by adjusting their weights based on their relation to an edge (e.g. the distance to the closest edge etc.). It could be also possible to deal with off-path particles during the Resampling phase by e.g. snapping them back to the edges. However, that might affect the MCL algorithm and damage its robustness because it tampers the effect of random noise and its distribution. Thirdly, we can separate the MCL particle set and the map, dealing with the MCL as without the knowledge the robot travels on a vector graph. This knowledge could be used later by the robot control system: let us consider a particle whose position indicates the robot is off the pathway. Such situation may have the following explanations: either the robot actually has left the pathway, or the robot is still on the pathway but its last move was not performed in real but should it be performed, the robot would move off the pathway, or the sensor measurements were erroneous and let position the particle wrongly off the pathway. However, in all the three cases, this does not mean a principal problem for the localization subsystem of the robot. If the robot is off the pathway, then this particle might correctly represent the robot position. If the robot is still on the pathway, then this particle does not represent the true position of the robot, but that is the characteristic of the MCL method – the particle set is composed of many particles, each one representing a *possible* state/position of the robot together with their respective weights which represent the *probability* of that state.

Another question arises if we want to use GPS device for graph-locked movement. As we mentioned earlier, the GPS position is error-prone. If we know that the robot should travel only on the pathways and the GPS provides the position, which is off any pathway, we cannot move the particle exactly as the GPS says. Instead, we should keep it on the graph edge. We considered two possible options: first, the nearest edge to the GPS position could be selected. However, because we know the expected precision (the GPS device provides it), we should consider all edges, which are at least partly inside the circle with the centre at the GPS position and the diameter of the reported precision. Figure 4 shows the example.

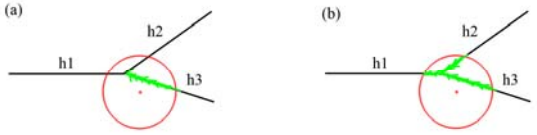



**Fig. 4.** Particles (green) generated on the closest edge (a) and considering the GPS precision (b).

## 6 Implementation

For the pilot implementation, the following decisions have been made:

The number of particles in the particle set will not be constant; it will be adaptable based on the set quality.

The quality of the particle set will be controlled by measuring the short-term and the long-term average of the particle weights. If the short-term weight average substantially drops below the long-term weight average, it means that the measurements do not correspond with the robot state belief represented by the particle set. In such case, a recovery step is taken: the most appropriate edge is detected and new particles are generated on this edge. The

particle direction will be either compass-determined (if the compass is used as one of the MCL inputs) or the edge direction is used (otherwise) because we expect the robot travelling along the edge. In either case, this direction may be not exactly true, but it does not pose any problem from the MCL point of view.

We consider that the robot always moves on the pathways and so all correct particles can lie only on the graph edges. This substantially reduces the allowed space for the particles and so there could be much less particles handled in comparison to covering the complete space.

In real life, the robot could move to places not depicted on the map as a pathway. That could mean that the robot has mistakenly left the pathway and runs on the grass, or that the robot is moving on a pathway (or other "allowed" surface) which is not recorded on the map. In both cases, its virtual position in the particle set might leave the edges too to better match the position and address this situation. However, during the testing phases of the implementation, we decided not to implement such behaviour for two reasons. Firstly, the robot control system should aim for not allowing such situation. Should it happen, it should aim to recover and return to the pathway recorded in the map: the robot is not allowed to travel on the grass and if it is not on the grass but started to run along an unrecorded pathway, the navigation module would not be able to use that newly explored path anyway. Secondly, the MCL algorithm is so robust that when the robot leaves the track and later returns back, the localization can recover by itself without external help or without the considered modification.

It should be also noted that for the algorithm, the graph nodes are not much important. The robot moves on the edges so the particles are on the edges too. The graph nodes are used only to switch from one edge to another.

## 7 Conclusion and future work

The presented MLC implementation for a special case – the localization on a graph – was dry tested using data gathered during the Robotour 2011 contest in Vienna. After the contest, the authors of the Eduro robot (see Fig. 5) provided us with the odometry information, compass readouts and GPS readouts record in time. During the contest, the robot control system (which was not part of our work and is not our concern) kept fortuitously the robot running in accordance with the requirements only on pathways. Without any modifications, we have used that recorded data as input for our implementation. The resulting particle set (evolving in time) well matched the logged behaviour of the robot in respect to the position along the pathway[1]. As we do not have exact measurements of the *absolute* robot position in the park[2], we can only compare the two localization algorithms. The correspondence of our testing MCL implementation with the original control algorithm was very good – our calculated position differed from the recorded data only in centimetres. However, in contrast to standard odometry and sensor measurements methods, MCL can recover from otherwise untraceable position changes (slips, skids or outer-force caused changes). Therefore, we are convinced this algorithm can be a useful addition to the robot control system.


**Fig. 5.** The Eduro robot which was used as a data source for the tests.

Our next planned work (now in progress) is the integration of this localization technique in a robot control system and its test in real world environment, e.g. in the next Robotour edition. Consequently, the comparison, evaluation, and measurements of the impact and contribution of MCL to the control system can be performed.

Concerning the algorithm itself, we would also like to consider the implementation of core parts on a GPU. Current GPUs are designed for handling points in the space and their manipulation, which is similar to particle shifting based on the robot move and weight recalculation. The particles are independent so the needed manipulation could be done in parallel. However, we foresee some challenges with the adaptive particle count.

## Acknowledgement

The authors wish to thank the Robotour contest organizers and contestants (especially from the Eduro team) for their help with data acquisition and for their cooperation on the project.

## References

1. Rise of the Robots: Robots deliver FM services at UK hospital. FM World magazine online, 2012.
2. Google self-driving car project. *http://goo.gl/dI6qA*
3. P. Anderson-Sprecher: *Probabilistic robot localization using visual landmarks.* Senior Honors Thesis, Macalester College, St. Paul, Minnesota, 2006.
4. OpenStreetMap project. *online resource:* http://www.openstreetmap.org
5. Robotour Contest. *online resource:* http://www.robotika.cz/robotour
6. A. Köhne, M. Wößner: *Fehlerquellen bei GPS. online resource:* http://www.kowoma.de/gps/Fehlerquellen.htm, 2012.
7. D. Obdrzalek: *Small autonomous robot localization system.* Procs. of IEEE SCOReD Conference, 2009.
8. S. Thrun, W. Burgard, D. Fox: *Probabilistic Robotics*. MIT Press, Cambridge, Massachusetts, 2005.
9. F. Dellaert, D. Fox, W. Burgard, S. Thrun: *Monte Carlo Localization for Mobile Robots*. Proc. of the IEEE International Conference on Robotics & Automation (ICRA99), 1998.

[1] The sideways position (perpendicularly to the path direction) cannot be by principle checked because the map does not contain the information about the pathway width.

[2] No independent robot absolute tracking was done during the contest so no such data is available. We have just a record of what the robot itself believed in.

ISBN 978-80-971144-1-1