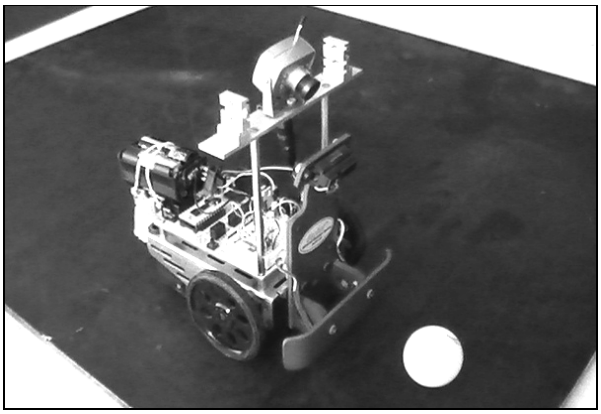
Multiagentový prístup k modelovaniu mysle - alebo ako sledovať pingpongovú loptičku

Andrej LÚČNY[1]

Abstrakt. Prezentujeme syntézu prác Marvina Minského, Rodneyho Brooksa a ich nasledovníkov. Na príklade riadenia mobilného robota, ktorého úlohou je sledovať pingpongovú loptičku, demonštrujeme multiagentový prístup k modelovaniu mysle. „Myseľ“ robota modelujeme ako decentralizovaný systém zložený z autonómnych a proaktívnych modulov – tzv. agentov – ktorého globálne správanie povstáva z ich vzájomných interakcií. Vysvetľujeme princípy ako situovanosť, subsumpcia a ďalšie.

1 Úloha

Za modelovú úlohu, ktorá nás bude sprevádzať celým výkladom, zvolíme napodobeninu známej hry s mačiatkom, ktoré naháňa imitáciu myši: mobilný robot vybavený dvojkolesovým podvozkom a kamerou bude sledovať pingpongovú loptičku zavesenú na šnúrke (Obr. 1).



Obr. 1. Modelová úloha: robot sledujúci pingpongovú loptičku.

[1] Katedra aplikovanej informatiky, FMFI UK, Bratislava, E-mail: lucny@fmph.uniba.sk

Táto málo užitočná ale zaujímavá úloha je – ako uvidíme neskôr – pekná z toho dôvodu, že ju možno dobre škálovať, t.j. stupňovať podmienky za ktorých ju má robot vykonávať i požiadavky na kvalitu jeho správania.

2 Klasický prístup k riadeniu robota

Pokiaľ znížime svoje nároky na kvalitu na minimum, na implementáciu takéhoto robota nám stačia základné znalosti z programovania, spracovania obrazu a ovládania motorov (Kód 1)[2].

```
SerialPort motors(comport,baudrate);
VideoCapture camera(0);
for (;;) {
    camera >> image;
    cvtColor(image,gray,CV_BGR2GRAY);
    GaussianBlur(gray,gray,Size(9,9),2,2);
    vector<Vec3f> circles;
    HoughCircles(gray,circles,CV_HOUGH_GRADIENT,2,50,200,100);
    if (circles.size() > 0) {
        float x = circles[0][0];
        float radius = circles[0][2];
        if (x < image.cols/2 – tolerance) motors << turnleft;
        else if (x > image.cols/2 + tolerance) motors << turnright;
        else if (radius < optimal – tolerance2) motors << forward;
        else if (radius > optimal + tolerance2) motors << backward;
        else motors << stop;
    }
    else motors << stop;
    image.release();
    delay(100); // wait 100 ms
}
```

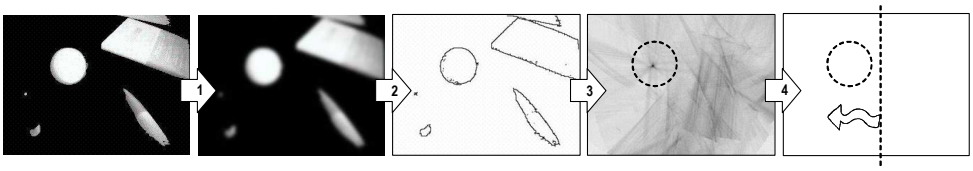
Kód 1. Ideový kód klasického riešenia (C++, OpenCV).

Robot je tu riadený tak, že opakovane snímame obraz a spracujeme ho na príkaz, ktorý vyšleme do podvozku robota. Celý proces spracovania vstupného obrázku na daný príkaz prebieha priamočiarou postupnosťou transformácii. Obrázok sa v prvom rade premení na čiernobiely. Potom ho trochu rozmažeme, čo je obľúbený a veľmi užitočný krok pred každou operáciou, ktorá pristupuje k obrazu ako keby bol spojitý (keďže to zníži šum). Potom vyvoláme Houghovu transformáciou[3], ktorá nám vráti

[2] Kódy a ďalšie materiály možno získať na www.agentspace.org/uiakv3

[3] Kľúčovou myšlienkou Houghovej transformácie je obrátenie procesu nakreslenia objektu určitých parametrov v binárnom obraze, s použitím hrubej sily. Tak ako z daného stredu a polomeru vieme vykresliť obraz kružnice, vieme aj z obrazu kružnice odhadnúť najpravdepodobnejší stred a polomer, z ktorých bola táto kružnica vykreslená. Odhad prebieha tak, že každý vykreslený bod hlasuje za všetky možné parametre kružnice, ktoré by ho mohli vykresliť (v rámci určitého rozsahu týchto parametrov). Následne sú tieto hlasy spočítané a parametre s najväčším počtom hlasov zodpovedajú hľadanej kružnici. Tento proces sa dá významne zrýchliť tým, že sa nehlasuje za všetky možné stredy, ale len za tie,

stredy a polomery kružníc na obraze (Obr. 2). Potom zoberieme prvú z nich a ak jej stred je príliš vľavo, vyšleme na motory príkaz na točenie vľavo, ak je príliš vpravo, tak točíme vpravo, ak je polomer príliš malý ideme dopredu, ak je príliš veľký, dozadu. Inak motory zastavíme.



Obr. 2. Jednotlivé fázy premeny čiernobieleho obrazu na motorický príkaz (1. rozmazanie, 2. Hranový detektor, 3. Houghova transformácia, 4. logika riadenia motorov)

Ostáva vysvetliť, aký význam má príkaz `delay(100)`, tj. čakanie 100 milisekúnd. Pokiaľ by sme ho nepoužili, náš program by okupoval procesor v maximálnej možnej miere. Hoci pri vstupno-výstupných operáciách (pri čítaní obrázku z kamery a vysielaní príkazu do podvozku) sa vykonávanie nášho programu na chvíľu zastaví, nestačí to na to, aby náš program ponechal priestor ďalším procesom v našom počítači. A to v prvom rade vláknam, ktoré sami nekódujeme, ale sú súčasťou knižníc ktoré linkujeme. Ľahko by sme sa o potrebe explicitného čakania presvedčili, keby sme si v našom kóde dali zobraziť obraz `gray`, vložením príkazu `imshow("gray",gray);` – pokiaľ teraz zakomentujeme `delay()`, nebudeme v zobrazovacom okne pre tento obraz nič vidieť (hoci obraz do tohto okna posielame, jeho zobrazenie beží v inom, pre nás neviditeľnom vlákne a to sa nedostane ku slovu). Je to preto, že sme pripustili stav, ktorý je z logického hľadiska v poriadku, ale z hľadiska reálneho vykonávania nášho programu v čase, v poriadku nie je. Ide o tzv. LiveLock, jeden zo základných problémov, s ktorými každý systém pracujúci v reálnom čase zápasí.

Zaujímavá je taktiež otázka, ako veľké má byť toto čakanie. Zdola je ohraničené výkonom procesora, vstupno-výstupných zariadení ako aj našou potrebou prenechávať kapacitu procesora iným procesom. Nemá napríklad zmysel ísť tu na 1 milisekundu: hoci procesoru by aj to pomohlo, kamera nám nedáva nové obrázky v takej frekvencii. Ešte zaujímavejšie však je, že toto číslo je ohraničené aj zhora, pričom tento limit je daný rýchlosťou motorov robota. Pre procesor by bolo lepšie, keby sme čakali napríklad celú sekundu – zatiaľ by však robot mohol pri otáčaní sa za loptičkou urobiť aj čelom vzad a úplne ju stratiť z dohľadu! Tomuto problému sa dá čeliť tým spôsobom, že motormi nebudeme hýbať kontinuálne, ale každý príkaz bude vykonávaný len určitý vopred stanovený čas. Takže ak aj príkaz na otočenie generujeme len raz za sekundu, robot sa bude točiť vždy len 100 milisekúnd

ktoré ležia na kolmici ku hrane, na ktorej sa nachádza hlasujúci bod. Úvodnou fázou tejto transformácie je teda jednak nájdenie bodov hrán (odpočítavaním jasu vzájomne blízkych bodov v smere x a y nájdeme hrubé hrany, prahovaním ich binarizujeme a ďalej upravujeme iteratívnym procesom stenčovania a orezávania), jednak ich orientácie v týchto bodoch (smernica ich normály je daná podielom diferencií v smere x a y) Viď [5] [16].

a zvyšných 900 milisekúnd bude stáť. Tento trik sa v robotike aj všeobecne používa a vyplývajú z neho charakteristické trhané pohyby, radšej rýchle, ale trvajúce krátky čas, prekladané pauzami v ktorých sa nič nedeje[4]. Pokiaľ čitateľovi toto riešenie stačí, môže rovno prejsť na ďalšiu kapitolu tejto knihy. My to takto robiť nechceme. Pre nás príkaz vyslaný na motory znamená, že sa vykonáva, až kým nevyšleme príkaz nový. Produkovaný pohyb robota je takto plynulý, prirodzený a podobný pohybu živých tvorov. Na druhej strane náš program takto svojim spôsobom „jazdí na neosedlanom koňovi" a ak sa pri svojich výpočtoch niekde zdrží, „hrozí mu pád". V ďalšom sa budeme snažiť ukázať ako sa dá „na tomto koni udržať aj bez sedla".

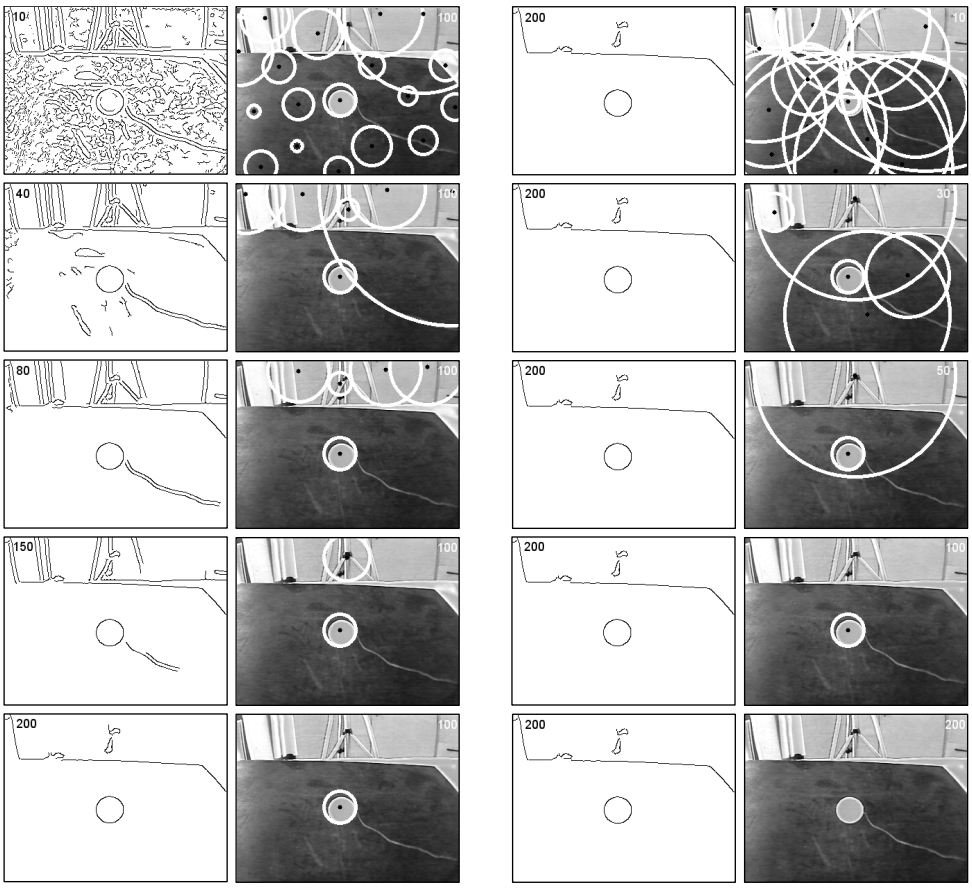
Na záver ešte spomeňme, že pri zvolenom riešení nebudú generované motorické príkazy perfektne pravidelné. Samotný výpočet príkazu totiž zaberie pre rôzne obrázky rôzne dlhý čas. Pauza má naopak perfektne rovnakú dĺžku. Oveľa krajšie by bolo, keby sme vedeli pre generovanie príkazu zabezpečiť stálu frekvenciu. Vtedy by sa dĺžka pauzy musela prispôsobiť dĺžke výpočtu. V praxi to znamená, že musíme pracovať so systémovými hodinami.

Aký nevinný príkaz – `delay(100);` – a koľko umenia sa v ňom skrýva!

3 Potreba paralelizmu a výmeny dát

Ako sme už spomenuli, zvolená úloha sa dá stupňovať do zložitejších podôb. Jeden zdroj takého stupňovania vidíme pri volaní Houghovej transformácie, kde vystupujú štyri konštanty (`2,50,200,100`). Prvá určuje presnosť s akou určujeme parametre kružnice, druhá minimálnu vzdialenosť stredov dvoch detekovaných kružníc. Tretia určuje hodnotu prahu pri hľadaní hrán, ktoré sa budú brať v úvahu a posledná prah od ktorého berieme v úvahu zdetekovanú kružnicu. Najmä tretia a štvrtá konštanta majú veľký vplyv na to, či sa podarí loptičku na obraze správne nájsť a pritom ich optimálne hodnoty nemožno vopred vedieť (Obr. 3), lebo závisia od podmienok na scéne, v ktorej sa robot nachádza. Tieto podmienky sa menia nielen zmenou osvetlenia scény, ale aj samotným pohybom robota v nej. Bolo by preto vhodné vyskúšať viac potenciálnych prahov a hľadať ich optimálne hodnoty, pri ktorých zdetekujeme niekoľko málo kružníc. To samozrejme môžeme spraviť aj v rámci klasického riešenia tak, že zakaždým v cykle vyskúšame veľa rôznych hodnôt prahov a podľa vráteného počtu kružníc budeme vedieť ktorý výsledok máme zobrať v úvahu pre generovanie motorického príkazu. Lenže týmto sa jeho generovanie značne spomalí a my môžeme mať – pri kontinuálnom hýbaní s motormi – problémy s rýchlosťou reakcií, ako aj s ich nízkou frekvenciou. Samozrejme ak raz vyhodnotíme hodnotu optimálneho prahu, môžeme predpokladať, že istý čas sa táto hodnota nezmení, alebo zmení len málo. Mohli by sme teda niekoľko najbližších krokov vykonať s rýchlymi reakciami (spoliehajúc sa na jedinú hodnotu prahu) a potom si dať jednu pomalú reakciu, pri ktorej hodnotu prahu prehodnotíme a pohyby robota radšej na chvíľu zastavíme.

[4] Túto vlastnosť bravúrne a inštinktívne postrehli herci, ktorí práve takéto pohyby robia, keď chcú vyvolať v divákovi dojem, že sú robotmi.



Obr. 3. Vplyv konštánt na výsledok Houghovej transformácie. Vľavo: vplyv prahu na detekciu hrán (viď číslo v ľavom hornom rohu), vpravo vplyv prahu na akceptovanie potenciálnej kružnice (viď číslo v pravom hornom rohu).

Čuduj sa svete, zase sme sa dostali k tomu charakteristickému trhanému pohybu. Ako sa mu vyhnúť? Pri spôsobe programovania, keď kladieme do sekvencie príkaz za príkazom, to proste nie je možné[5]. Musíme si uvedomiť, že riadenie robota musí do určitej miery napodobňovať procesy, ktoré sa dejú v našej mysli. A niet charakteristickejšej vlastnosti pre tieto procesy, než je ich paralelizmus. Hoci aj nemáme paralelný počítač, musíme vedieť tento paralelizmus napodobniť v tom, že pomalé procesy (ako je v našom prípade hľadanie optimálneho prahu) nesmú brzdiť vykonávanie rýchlejších procesov (ako je generovanie motorického príkazu, keď už

[5] Jedna škaredá možnosť by tu ešte bola a to naprogramovať oba procesy ako tzv. podprogramy s viacnásobným prístupom, tj. rozbiť ich kód na malé úseky a volať v správny čas správny úsek. To už ale zavádzame do aplikačného programu primitívne plánovanie procesov, čo sa patrí ponechať operačnému systému. Pri tomto type programovania sa kód stáva veľmi rýchlo neprehľadný. Preto táto stratégia nie je vhodná pre komplexnejšie systémy.

tento prah poznáme). Aktualizácia výsledkov pomalých procesov v systéme musí mať nižšiu frekvenciu než aktualizácia výsledkov procesov rýchlych. (Cieľom je, aby sme radšej reagovali rýchlo s neoptimálnym prahom, než nereagovali, čakajúc na optimálnu hodnotu. Rýchla a menej presná reakcia sa dá dodatočne zdokonaliť vo chvíli, keď je už optimálna hodnota známa.)

Takže už pri tak jednoduchej komplikácii našej úlohy, akou je potreba zvládnuť rôzne svetelné podmienky na scéne, vidíme potrebu paralelného behu viacerých procesov (zatiaľ dvoch, ale je snáď jasné, že pri stúpajúcich nárokoch na správanie robota, to pri tomto čísle neostane). Okrem toho vidíme potrebu prevádzkovať v istom súlade rýchle a pomalé procesy. A vidíme aj potrebu, aby si tieto procesy vzájomne vymieňali dáta (v našom prípade hodnotu optimálneho prahu).

4 Multiagentový prístup

K potrebe paralelizmu procesov a výmeny dát medzi nimi sa prepracuje tvorca každého zložitejšieho robota. Obvykle ju však vyrieši takzvane ad-hoc. V závislosti od platformy, ktorú má k dispozícii využije multiprocesové alebo multivláknové prostredie, ktoré mu vytvára operačný systém, spoliehajúc sa na spravodlivé (preemptívne) striedanie procesov či vlákien v procesore. Výmenu dát potom zrealizuje s využitím niektorého zo synchronizačných mechanizmov (semafory a pod.), keďže nie je ani tak problém dáta vymieňať, ako pri tejto výmene zachovať ich korektný obsah[6]. Tu sa väčšina vývojárov zastaví a pokojne využije na dosiahnutie tohto cieľa v rámci jedného systému rôzne techniky. My však s menšinou pôjdeme ďalej a budeme požadovať normalizáciu použitého mechanizmu.

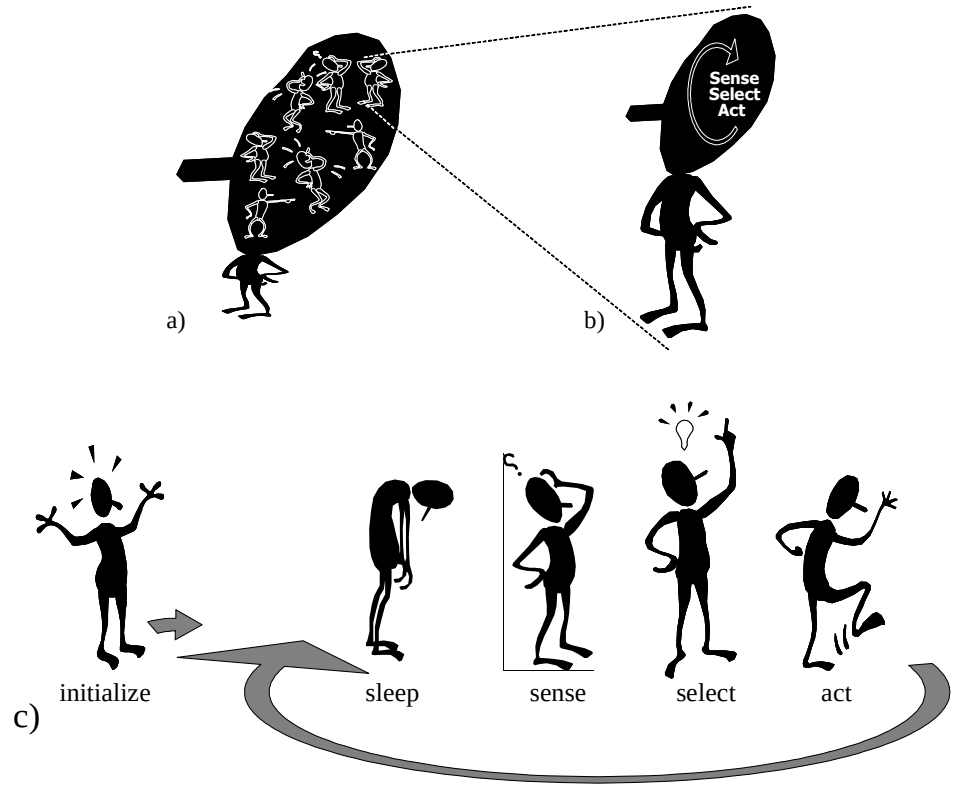
Jednu z možností takejto normalizácie poskytuje tzv. multiagentový prístup. Na akýkoľvek systém pri ňom pozeráme ako na distribuovaný, t.j. skladajúci sa z modulov, ktoré musia spolu na diaľku komunikovať, aby si vymieňali dáta[7] (Obr. 4a). Vývojár tohto systému potom miesto explicitnej implementácie správania systému, explicitne implementuje správanie jeho jednotlivých modulov a to tak, aby globálne prejavy systému, ktoré z jednotlivých správaní týchto modulov vyplynú, zodpovedali želanému správaniu. Navyše pri multiagentovom prístupe sa tieto moduly vyznačujú tým, že majú vlastné riadenie, t.j. ide o paralelne bežiace procesy, o tzv. agentov (Obr. 4b). Samozrejme pokiaľ takýto systém prevádzkujeme na jednom počítači, paralelný beh bude len napodobnený a žiadny agent zďaleka nesmie vyčerpať všetok strojový čas (to by nastal spomínaný LiveLock) – inými slovami väčšinu času musí prespať. Čo sa týka tvaru kódu, každý agent vyzerá podobne: po inicializačnej fáze vbehne do nekonečného cyklu, v ktorom zaspí, pri zobudení vníma v akej situácii sa ocitol, zvolí si, či a čo v danej situácii podnikne a vykoná to (Obr. 4c). To je samozrejme obrazne povedané[8]: vnímanie tu spočíva v prenose dát do agenta, voľba

[6] Hrozí tu, že viaceré súbežné procesy naraz modifikujú rovnaký dátový priestor a jeho obsah sa tým stáva nekonzistentný

[7] Pokiaľ celý náš systém beží na jednom počítači napĺňame tak obsah sloganu „The computer is the network!"

[8] a to rečou typickou pre oblasť známu ako „multiagentové systémy". Viď napr. [6] [9] [18]

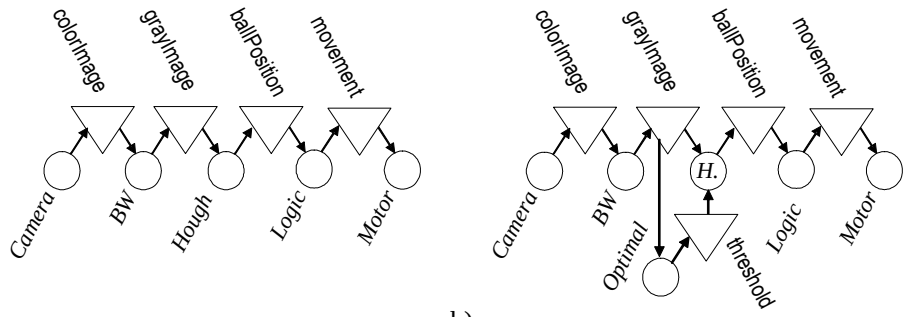
vo výpočte, ktorý tieto dáta spracúva a vykonanie akcií v prenose dát z agenta. Prenos dát pritom môže vo všeobecnosti prebiehať dvomi spôsobmi: priamo, t.j. agent pošle dáta druhému agentovi a nepriamo, t.j. agent umiestni dáta do prostredia medzi agentmi, odkiaľ si ich iní agenti môžu prevziať. My sa sústredíme výhradne na druhý spôsob[9], t.j. na nepriamu komunikáciu, lebo lepšie vyhovuje našim zámerom.



Obr. 4. a) Multiagentový systém. b) Agent. c) Životný cyklus agenta.

Ako teraz pristúpime k tvorbe systému, ktorý riadi nášho robota, čo sleduje pingpongovú loptičku? V prvom rade prerobíme pôvodné riešenie tak, že sekvenciu, ktorá spracúva obraz rozbijeme na niekoľko agentov, ktorí si v analogickej sekvencii budú posúvať dáta (Obr. 5a). Zamýšľame totiž túto sekvenciu postupne vetviť (v prvom kroku pridaním hľadania optimálneho prahu – Obr. 5b) a neskôr dokonca vytvárať v rozrastajúcej sa dátovej výmene cykly.

[9] náš prístup bude založený na vlastnej architektúre, ktorú sme nazvali Agent-Space [13]



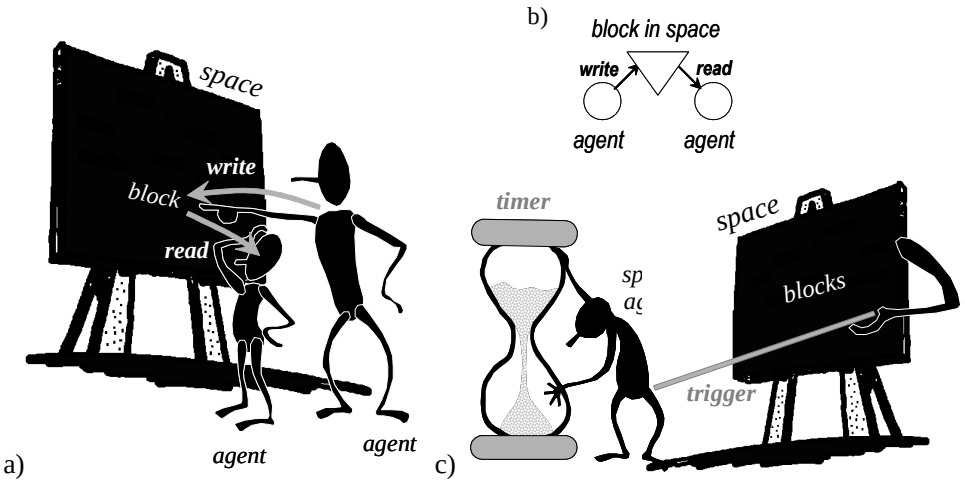
a) b)

Obr. 5. a) Multiagentový systém analogický pôvodnému riešeniu. b) Rozšírenie, pri ktorom pomalý agent *Optimal* nezdržuje rýchlejšieho agenta *Hough* (*H.*). (Agenti sú zobrazení kruhom, komunikované dáta trojuholníkom.)

Z hľadiska implementácie je každý agent realizovaný samostatným vláknom[10]. Ostáva vysvetliť ako sú tieto vlákna synchronizované a akým štandardným mechanizmom si vymieňajú dáta. Ako sme už spomenuli, základným princípom tejto dátovej výmeny je pre nás výlučne nepriama komunikácia, je teda založená na tzv. čiernej tabuli (blackboard)[11], ktorú nazývame – v súlade so zaužívanom názvom v tzv. koordinačnom programovaní – *space*. Táto tabuľa je spoločná pre všetkých agentov a je možné na ňu umiestniť pomenované dáta rôzneho formátu, tzv. bloky. Základnú manipuláciu s blokmi predstavuje na každej takejto tabuli zápis a čítanie, t.j. jeden agent blok zapíše a druhý agent ho prečíta (Obr. 6a,b). Zápis bloku bude samozrejme prebiehať vo fáze vykonávania akcií (*act*), otázka je však na základe čoho sa uskutoční čítanie. Keďže agenti priamo nekomunikujú, čitateľ bloku nevie, ani kto, ani kedy zapísal na tabuľu novú informáciu. Zapisujúci agent sa teda nijako nepričiní o to, aby sa čítajúci agent zobudil (z fázy *sleep*) a blok prečítal (v rámci fázy *sense*). Ostávajú mu preto len dve možnosti. Môže sa chodiť pravidelne pozerať na tabuľu, čo je účinná stratégia, keď vie, že za normálnych okolností sa tam pravidelne objavujú nové dáta. Vtedy používa tzv. časovač (*timer*). S touto stratégiou by vystačil napríklad agent BW, ktorý vie, že kamera poskytuje obrázky raz za 100ms. Keby ju však použili všetci agenti v celej sekvencii, spracovanie obrazu by sa zbytočne oneskorovalo o čas, ktorý uplynie medzi zápisom a čítaním. Preto má čítajúci agent k dispozícii aj ďalší mechanizmus zvaný spúšťač (*trigger*). Spúšťač je realizovaný tabuľou, ktorú na úvod čítajúci agent požiada, aby ho zobudila, keď sa zmení obsah bloku s určitým menom. Keď zapisujúci agent spôsobí, že táto udalosť nastane, tabuľa čítajúceho agenta zobudí a ten si hodnotu daného bloku prečíta (Obr. 6c).

[10] Čo je pochopiteľne menej efektívne riešenie, volíme ho pre jeho schopnosť ďalšieho škálovania, t.j. jeho rozširovania na komplexnejší systém

[11] Ideovo ju možno považovať za derivát tzv. tupple-space z jazyka LINDA[8], navrhnutého pre programovanie paralelných počítačov, ktorý našiel uplatnenie v rámci tzv. koordinačného programovania [4], i v rámci technológie Java Jini spoločnosti Sun ako tzv. Java Space [17].



Obr. 6. a) Nepriama komunikácia medzi agentmi. b) Schéma dátového toku medzi dvomi agentmi. c) Budenie agenta časovačom (timer) a spúšťačom (trigger).

Ostatné aspekty manipulácie s blokmi popíšeme, až keď ich budeme potrebovať (je ich ako uvidíme pomerne dosť a niektoré by pri skutočnej tabuli vyvolávali ešte zázračnejší dojem ako spúšťače – naša tabuľa je však len kus softvéru vo virtuálnom svete a môžeme jej dať také schopnosti ako sa nám hodí). Skúsme sa teraz pozrieť ako bude vyzerať kód agentov realizujúcich riadenie nášho robota (Obr. 5a).

```
class AgentCamera : public Agent {
    protected:
        VideoCapture camera;
        void init (string args) {
            camera.open(0);
            timer_attach(100,100);
        }
        void sense_select_act (int pid) {
            Mat image;
            camera >> image;
            space_write("colorImage",image.clone(),200);
        }
    public:
        AgentCamera (string args) : Agent(args) {};
};
```

Kód 2. Kód agenta, ktorý realizuje príjem obrazu z kamery (C++, OpenCV, AgentSpace)

Prvý z nich – agent *Camera*, bude budený časovačom a v pravidelných taktoch bude preberať obraz z kamery a vkladať ho na tabuľu – *space* (Kód 2). Pritom je zaujímavé všimnúť si niekoľko vecí. V prvom rade, vidíme, že agent je definovaný ako odvodená trieda od triedy *Agent*, ktorá je aplikačne nezávislá a implementuje v sebe všetky

potrebné (a dosť krkolomné) mechanizmy: vlákna, časovače, tabuľu a spúšťače, ktoré by vývojár bez využitia knižnice *AgentSpace* musel do istej miery urobiť sám a to s každou novou aplikáciou. Nadradená trieda *Agent* zriadi vo svojom konštruktore samostatné vlákno (*pthread*), ktoré vykoná metódu *init()* a potom v cykle pomocou synchronizačného zámku (*mutex*) zaspí a po zobudení zavolá metódu *sense_select_act()*. Prekrytím týchto dvoch metód preto vývojár zimplementuje plnohodnotného agenta. Ďalej je zaujímavé si uvedomiť, že do *space* je potrebné vložiť nie ten objekt, s ktorým lokálne v agentovi pracujeme, ale jeho klon. On sám totiž môže byť po vykonaní zápisu modifikovaný a nakoniec zanikne ako náhle riadenie opustí *sense_select_act()*. *AgentSpace* preto automaticky pri zápise klonuje tzv. hlbokým klonovaním všetko, čo vie naklonovať samotná platforma. Existujú však zložité objekty – ako je napríklad obraz – ktoré vnútorne odkazujú na alokované dáta a o ich klonovanie sa musí postarať vývojár. Samozrejme klonovanie, podobne ako ďalšie mechanizmy *AgentSpace*, predstavuje pre systém určitú záťaž. V odôvodnených prípadoch sa dá do Space zapísať aj smerník a synchronizácia prístupov ako aj deštrukcia zapísaného objektu je potom ponechaná na vývojárovi. Podobný problém nastáva pri čítaní zo space, pretože viac agentov môže prečítať ten istý objekt. Vývojár ho preto nesmie modifikovať – a ak by to potreboval, potom ho musí naklonovať.

```
class AgentBW : public Agent {
    protected:
        void init (string args) {
            trigger_attach("colorImage");
        }
        void sense_select_act (int pid) {
            Mat color, gray, blank(240,320,CV_8UC3);
            color = space_read("colorImage",blank);
            cvtColor(color,gray,CV_BGR2GRAY);
            space_write("greyImage",gray.clone(),120);
        }
    public:
        AgentBW (string args) : Agent(args) {};
};
```

Kód 3. Kód agenta, ktorý realizuje transformáciu farebného obrazu na čiernobiely (C++, OpenCV, AgentSpace)

Najjednoduchší z agentov je agent *BW*, ktorý realizuje transformáciu farebného obrazu na čiernobiely (Kód 2.). Mohlo by sa zdať, že tento kód by bolo lepšie včleniť priamo do agenta *Hough*, lenže čiernobiely obraz je dosť častým vstupom pre rôzne metódy spracovania obrazu a s pribúdajúcimi agentmi, by sme ho stále znovu a znovu zbytočne počítali. Na rozdiel od agenta *Camera*, *BW* je budený spúšťačom, t.j. jeho synchronizačný zámok sa odblokuje zápisom do *space* (v našom prípade to bude práve vlákno agenta *Camera*, ktoré zápisom farebného obrazu odblokuje vlákno agenta *BW* a podnieti ho, aby z tohto obrazu vypočítal obraz čiernobiely).

Na prácu agenta *BW* priamo nadväzuje agent *Hough*, ktorý zo *space* prečíta čiernobiely obraz a zapíše do neho pozíciu loptičky (Kód 4.). Tu nastáva dilema

spočívajúca v tom, že agent *Hough* niekedy nájde a inokedy nenájde loptičku. Má teraz na výber dve stratégie. Keby chcel zobudiť agenta *Logic* spúšťačom, t.j. analogicky ako bol budený on sám i agent *BW*, musel by v prípade, že loptičku nenájde zapísať do *space* hodnotu „neznáma“ a *Logic* by sa na to zobudil a vydal by pokyn na zastavenie robota. Alternatívne by mohol hodnotu zo *space* zmazať, pričom vidíme, aké je dôležité, aby aj zmazanie bloku zo *space* vyvolávalo spúšťače zaregistrované na daný blok.

```
class AgentHough : public Agent {
    protected:
        void init (string args) {
            trigger_attach("grayImage");
        }
        void sense_select_act (int pid) {
            Mat blank(240,320,CV_8UC1), gray = space_read(„grayImage“,blank);
            HoughCircles(gray,circles,CV_HOUGH_GRADIENT,2,50,200,100);
            if (circles.size()>0) space_write("ballPosition",circles[0],300);
        }
    public:
        AgentHough (string args) : Agent(args) {};
};
```

Kód 4. Kód agenta, ktorý realizuje Houghovu transformáciu (C++, OpenCV, AgentSpace)

Niežeby to nefungovalo, ale v našom kóde sme použili lepšiu stratégiu spočívajúcu vo využití mechanizmu časovej platnosti[12]. Pozorný čitateľ si iste v našich kódoch pri zápisoch do *space* všimol dosiaľ nekomentovaný tretí parameter, ktorý napríklad pre zápis *grayImage* nadobúda hodnotu 120. Čo to je? Ako sme spomenuli, naša tabuľa je softvérová a z toho dôvodu sa dokáže – keď je to potrebné – správať aj zázračným spôsobom. Okrem spúšťačov, dokáže realizovať aj mechanizmus, ktorý zapisujúcemu agentovi umožní definovať čas, po uplynutí ktorého dáta z tabule bez jeho pričinenia zmiznú. Čiže tých 120 znamená, že keby sme stopli agenta *Camera*, tak ešte 120 milisekúnd budú mať agenti k dispozícii čiernobiely obraz (normálne ho aktualizujeme každých 100 ms), pričom po tento čas sa nebude meniť. Bolo by ale tragédiou, keby to trvalo dlhšie, lebo ak by napríklad v tomto obraze bola loptička vľavo, robot by sa stále točil doľava a nič by mu nepomohlo sa z tohto točenia dostať, keďže v jeho vnútornej reprezentácii by už jeho pohyb nemal na vnímaný obraz žiaden vplyv. Samozrejme o odstránenie tohto obrazu zo *space* by sa mohol postarať aj niektorý z agentov, nič menej to by bola zbytočná komplikácia ich kódu (napríklad keby to bol agent BW, musel by byť budený aj časovačom, aj spúšťačom – čo sa síce dá – vďaka tomu, že *sense_select_act()* dostáva ako parameter identifikátor *pid* (*proxy id)*, porovnateľný z návratovou hodnotu *trigger_attach()* respektíve *timer_attach()* – ale je to zmätok). Ďaleko jednoduchšie je využiť mechanizmus časovej platnosti.

Ako teda tento mechanizmus využije agent *Hough*? Namiesto toho, aby v prípade, že loptičku na obraze nenájde, mazal z tabule svoj predchádzajúci názor, zapíše tam

[12] Prvý krát bol tento mechanizmus navrhnutý pre Java Space ako tzv. „data leasing“ [17]

radšej tento názor s obmedzenou časovou platnosťou a to takou, aby v prípade, že loptičku kontinuálne detekuje, na tabuli jeho názor vždy bol. V našom prípade sme zvolili 300 milisekúnd, čo zodpovedá náročnosti výpočtu Houghovej transformácie.

Z tohto čísla zároveň vidíme, že agent *Hough* nemusí nevyhnutne stihnúť spracovať každý snímok ktorý príde od kamery. V tradičnom riešení (Kód 1) sa to vyrieši tým, že sa frekvencia záberov z kamery prispôsobí výpočtovým nárokom Houghovej transformácie. Tu naopak tento vplyv nemáme a preto ak paralelne pustíme rýchleho agenta, ktorý by loptičku detekoval nie podľa tvaru ale napríklad podľa farby, mohol by pracovať na rovnakej frekvencii ako kamera.

```
class AgentLogic : public Agent {
    protected:
        void init (string args) {
               attach_trigger("ballPosition"); // attach_timer(100,100)
        }
        void sense_select_act (int pid) {
            Vec3f middle(cols/2,2*row/3,optimal);
            Vec3f circle = space_read("ballPosition",middle);
            float x = circle[0];
            float radius = circle[2];
            string motion;
            if (x < cols/2 - tolerance) motion = turnleft;
            else if (x > cols/2 + tolerance) motion = turnright;
            else if (radius < optimal - tolerance2) motion = forward;
            else if (radius > optimal + tolerance2) motion = backward;
            else motion = stop;
            space_write("movement",motion,100);
        }
    public:
        AgentLogic (string args) : Agent(args) {};
};
```

Kód 5. Kód agenta, ktorý realizuje logiku pohybu robota (C++, AgentSpace)

Kód agenta Logic (Kód 5) uvádzame bez definovania niektorých konštánt. Obsahuje jednu zaujímavú fintu. Pri čítaní zo *space* je oveľa lepšie než kontrolovať, či sa hodnotu podarilo prečítať (požadovaná hodnota tam ešte nemusí alebo už nemusí byť), definovať náhradnú hodnotu, ktorú čítanie v takom prípade vráti. To je ten dosiaľ ignorovaný druhý parameter *space_read()*. Čiže miesto toho, aby sme vetvili kód pre prípad, že máme loptičku a nemáme loptičku, budeme v prípade, že loptičku nemáme, predpokladať, že je na správnom mieste. Keďže je tento agent budený spúšťačom, tak v momente kedy sa loptička stratí, agent *Hough* prestane aktualizovať blok *ballPosition*, 300 ms sa nič neudeje, potom platnosť *ballPosition* vyprší, načo sa agent *Logic* zobudí (rovnako ako keby prišla nová hodnota), nadobudne dojem, že loptička je v správnej polohe, vydá príkaz *stop* a nič sa nedeje, až kým *Hough* neuvidí loptičku znovu.

Alternatívne by agent *Logic* mohol byť budený časovačom s nízkou frekvenciou (viď zakomentovaný príkaz v init(), kód 5). V takom prípade by bol

motorický príkaz generovaný kontinuálne, došlo by však k určitému (hoci akceptovateľnému) oneskoreniu jeho vykonania.

Na záver tohto príkladu uvádzame kód agenta *Motor* (Kód 6), ktorý len zoberie príkaz na pohyb a vykoná ho na motoroch tým, že ho zapíše do komunikačného portu. Z neho ide do podvozku robota, kde je prečítaný v palubnom procesore a adekvátne je podľa neho generovaná frekvencia, ktorou sú ovládané príslušné servo motory. Jediný nový prvok tu spočíva vo využití odovzdávania argumentov – odovzdávame takto meno komunikačného portu.

```
class AgentMotor : public Agent {
    protected:
        SerialPort motor;
        void init (string args) {
            motor.open(args,9600/*bps*/);
            attach_trigger("movement");
        }
        void sense_select_act (int pid) {
            string motion = space_read("movement",stop);
            motor << motion;
        }
    public:
        AgentMotor (string args) : Agent(args) {};
};
```

Kód 6. Kód agenta, ktorý realizuje logiku pohybu robota (C++, AgentSpace, SerialPort)

Ostáva všetkých agentov naštartovať. To sa našťastie urobí pomerne ľahko, lebo medzi agentmi nie sú žiadne formálne väzby či závislosti. Jedine po naštartovaní nesmieme hneď ukončiť beh programu, ale musíme ho zablokovať, napríklad na vstup od užívateľa (Kód 7.)

```
int main() {
    AgentCamera Camera("");
    AgentBW BW("");
    AgentHough Hough("");
    AgentLogic Logic("");
    AgentMotor Motor("COM40");
    getch(); // don’t exit
}
```

Kód 7. Príklad naštartovania systému (C++, AgentSpace)

V tejto chvíli máme hotové multiagentové riešenie analogické tradičnému. Zatiaľ sme z toho, že sme pár riadkov kódu nahradili desaťnásobkom nemali žiadnu výhodu. Teraz však taká chvíľa prichádza. Pozrime sa teda na to, ako zrealizujeme rozšírenie systému na Obr. 5b. V prvom rade musíme z kódu vyňať natvrdo nakódované konštanty prahov a umiestniť ich na tabuľu (*space*) a urobiť z nich tak parametre. Je všeobecne poučné, že pre potenciálne rozširovanie systému je vhodné všetky takéto

konštanty už preventívne držať v *space* – hoci to opäť zvyšuje réžiu, keďže ich odtiaľ treba čítať. Preberanie parametrov zo *space*, ktoré sa menia iba zriedka sa síce dá efektívne napísať pomocou spúšťačov (Kód 8a), nič menej oveľa krajší a udržateľnejší kód agentov budeme mať práve vtedy, ak ich budeme čítať pri každom použití (Kód 8b).

```
a)  int cid;
    int ct;
    void init (string args) {
        trigger_attach("grayImage");
        cid = trigger_attach("cannyThreshold");
        ct = 200;
    }
    void sense_select_act (int pid) {
        if (pid == cid) ct = space_read("cannyThreshold",200);
        else {
            Mat blank(240,320,CV_8UC1), gray = space_read("grayImage",blank);
            HoughCircles(gray,circles,CV_HOUGH_GRADIENT,2,50,ct,100);
            if (circles.size()>0) space_write("ballPosition",circles[0],300);
        }
    }

b)  void init (string args) {
        trigger_attach("grayImage");
    }
    void sense_select_act (int pid) {
        int ct = space_read("cannyThreshold",200);
        Mat blank(240,320,CV_8UC1), gray = space_read("grayImage",blank);
        HoughCircles(gray,circles,CV_HOUGH_GRADIENT,2,50,ct,100);
        if (circles.size()>0) space_write("ballPosition",circles[0],300);
    }
```

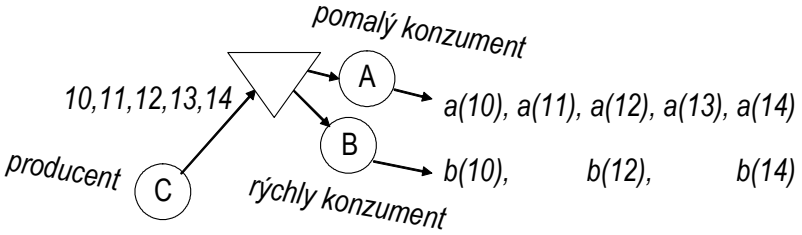
Kód 8. Preberanie parametrov: a) spúšťačmi (reaktívne), b) zakaždým (čisto reaktívne)

Je zaujímavé si všimnúť, že agentov v tomto príklade odlišuje nielen použitie spúšťačov, ale aj vlastnosť týkajúca ich tzv. vnútorného stavu. Kým agent nakódovaný v 8a) obsahuje informáciu, ktorá v ňom trvalejšie prežíva (tj. parameter *ct*), agent z 8b) si nič dlhodobo nepamätá. V zmysle teórie behaviorálnych systémov ide o tzv. čisto reaktívneho agenta [2]. Pre vývojárov takéhoto systému je čistá reaktivita všetkých agentov ideálom: niečím čo sa zriedka podarí dosiahnuť, ale čím bližšie sa k tomu dostaneme, tým lepšie. Vtedy sme si totiž istí, že v žiadnom agentovi nemáme schovanú a nedobytnú informáciu, ktorú by sme pri ďalších rozšíreniach systému potrebovali zmeniť a nebudeme mať na to iné prostriedky, než sa vrátiť k úprave kódov už nakódovaných a odladených agentov – tak ako sme to my museli urobiť v prípade vytiahnutia parametra z agenta *Hough*. V našom prípade šlo o triviálnu úpravu, takže žiadny problém – ale nemusí to byť vždy tak.

Po úprave agenta *Hough* už nebude problém pridať ďalšieho agenta *Optimal* (Obr. 5b), ktorý podobne ako *Hough* preberá čiernobiely obraz, ale nepočíta na ňom jednu transformáciu, ale viacero, pričom skúša nájsť takú hodnotu prahu pri ktorej ešte

nejaké kružnice zdetekované má, ale keby zdvihol prah ešte vyššie, už nemá. Nijako sa pritom neponáhľa. Vie, že nie je ten najdôležitejší agent a tak si dáva aj úmyselné pauzy, nech dopraje strojový čas iným. Na rozdiel od agenta *Hough* sa vôbec netrápi preto, že nestihne prečítať každý obraz, čo vygeneruje agent *BW*. Stačí mu zamudrovať správny prah raz za čas. Robot sa tak pri sledovaní loptičky môže dostať do iných svetelných podmienok v ktorých loptičku stratí hoci sa na ňu pozerá. Vtedy trvá istý čas, kým agent *Optimal* vymyslí nový prah. Je to podobné ako keď človek príde zo svetla do tmy – chvíľu mu to trvá, kým sa začne orientovať.

Čítanie čiernobieleho obrazu, ktorý produkuje agent *BW* agentmi *Hough* a *Optimal* je poučné z hľadiska javu, ktorý nazývame implicitné vzorkovanie (obr. 7). Keď agent *BW* zapisuje do *space* blok *grayImage*, vždy tým prepíše predchádzajúcu hodnotu (a vyvolá deštrukciu objektu, ktorý ju reprezentuje). Vôbec ho nezaujíma, či ho niektorý z agentov stihol prečítať. Agent *Hough* sa snaží čítať podľa možnosti každý, zato *Optimal* ich nestíha prečítať drvivú väčšinu. Je pritom jedno, či budeme agenta *Optimal* budiť časovačom alebo spúšťačom: pokiaľ sa motá v *sense_select_act()*, všetky budenia sa v rámci rovnakého podnetu zlievajú do toho posledného (nezlejú sa len rôzne budenia, takže napríklad ak prichádza veľa podnetov z rôznych spúšťačov, agent dostane od každého práve jedno budenie). Dátový tok medzi agentom *BW* (producentom dát) a agentmi *Hough* a *Optimal* (konzumentami dát) je teda automaticky vzorkovaný na takú frekvenciu, ktorá zodpovedá výpočtovým možnostiam konzumentov. Toto vzorkovanie je inherentnou vlastnosťou *space* a v kóde agentov sa nijako neprejavuje.



Obr. 7. Implicitné vzorkovanie

Implicitné vzorkovanie nie je vhodné vždy a všade. Bolo by napríklad nepríjemné počítať takýmto systémom peniaze. Avšak pre systém, ktorý má pracovať v reálnom čase a nedostáva sa mu na to dostatok výpočtovej kapacity, je implicitné vzorkovanie požehnaním. Takýto systém sa nikdy nemôže zahltiť a akosi sám od seba robí vždy to najlepšie, čo sa dá pri danej výpočtovej kapacite stihnúť. Pritom vývojár nemusí toto prispôsobovanie sa nijako explicitne vyjadrovať vo svojich kódoch – funguje to úplne automaticky. Je zaujímavé si všimnúť, že je tu obrátená logika chápania, čo je to systém reálneho času. Tradičný názor je, že na to by ste mohli mať systém reálneho času, nevyhnutne potrebujete dostatočnú výpočtovú kapacitu. Pri našom prístupe toto obmedzenie prekonávame a schopnosť systému ako-tak pracovať v reálnom čase je daná jeho modulárnym usporiadaním. Budujeme tu systém, ktorý chce byť systémom

reálneho času napriek tomu, že sa mu na to nedostáva dosť výpočtovej kapacity. Má však mechanizmus, ktorým tomuto nedostatku čelí[13].

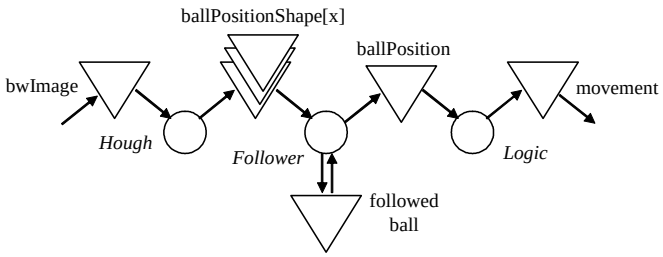
```
a)  void init (string args) {
        trigger_attach("grayImage");
    }
    void sense_select_act (int pid) {
        int ct = space_read("cannyThreshold",200);
        Mat blank(240,320,CV_8UC1), gray = space_read("grayImage",blank);
        HoughCircles(gray,circles,CV_HOUGH_GRADIENT,2,50,200,100);
        for (int i=0; i<circles.size(); i++)
            space_write("ballPositionShape*",circles[0],300);
    }

b)  void init (string args) {
        tid = trigger_attach("ballPosition*",TRIGGER_MATCHING);
    }
    void sense_select_act (int pid) { // pid == tid
        Vec3f last = space_read("followedBall",Vec3f(0,0,0));
        bool nolast = (last[2] == 0);
        SpaceIterator<Vec3f> it;
        for (it=space_read_begin(pid); it!=space_read_end(pid); it++) {
          Vec3f candidate = it->value;
          if (nolast || (fabs(candidate[0]-last[0])<tolerance
                      && fabs(candidate[1]-last[1])<tolerance)) {
             space_write("followedBall",candidate,600);
             space_write("ballPosition",candidate,300);
             break;
          }
       }
    }
```

Kód 9. a) úpravy agenta *Hough*. b) nový agent *Follower*

Keď teraz robot vojde do tmy, na chvíľu ustrnie a potom začne normálne pracovať. Keď však vystúpi z tmy do svetla, nespráva sa až tak pekne, lebo počas doby hľadania nového prahu sa snaží sledovať to veľké množstvo kružníc, ktoré sa mu zrazu objavili pred očami. Podobná situácia vzniká aj bez zmeny svetelných podmienok, pokiaľ do výhľadu robota umiestnime viacero loptičiek. Na vylepšenie tohto správania možno uvažovať ďalšiu modifikáciu systému, ktorá zabezpečuje kontinuitu sledovania jednej loptičky. Nedostatok nášho doterajšieho návrhu spočíva v tom, že agent *Hough* zapíše do *space* vždy len jednu pozíciu: prvú, ktorá sa mu naskytne. Nie je samozrejme problém, aby tam zapísal miesto jednej pozície celý vektor. Ešte lepšie však bude, keď do *space* zapíše jednotlivé pozície ako samostatné bloky. Je to lepšie preto, že tak možno zamiešať jeho názor s názorom ďalších detektorov, ktoré by sme pridali v budúcnosti. Dostávame sa tak k potrebe opäť rozšíriť schopnosti *space*, tentoraz o hromadné manipulácie s blokmi.

[13] Domnievame sa, že v podobnej situácii sa nachádzajú aj živé organizmy a že v ich modulárnej štruktúre by sa dal nájsť podobný mechanizmus [12]



Obr. 8. Úprava pre zameriavanie pozornosti na konkrétnu loptičku

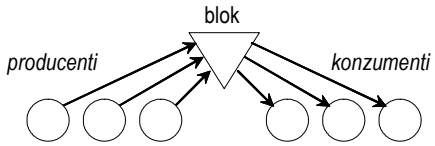
Realizáciu vidíme na kódoch 9a a 9b. Agenta *Hough* (Kód 9a) sme modifikovali tak, aby do *space* zapísal všetky nájdené kružnice. Na to mu slúži hviezdičková konvencia, ktorá spôsobí, že k menu zapisovaného bloku je pripojený index tak, aby jeho meno nebolo v konflikte s aktuálne platnými blokmi s rovnakou maskou. K menu zapisovaných blokov bola taktiež pridaná prípona *Shape* determinujúca pôvodcu bloku, aby v budúcnosti mohli iní agenti zapisovať iných kandidátov, napríklad *ballPositionColor**. Medzi agenta *Hough* a agenta *Logic* vložíme agenta *Follower*, ktorý, bude realizovať zameranie pozornosti robota na jednu loptičku (Kód 9b). Ako vidíme z jeho kódu, čítanie viacerých blokov podľa masky je realizované prostredníctvom iterátora, ktorého agent získa podľa identifikátora jeho spúšťača. Stratégia tohto agenta je pomerne jednoduchá: pokiaľ nesleduje konkrétnu loptičku, tak sa sústredí na prvú, ktorá sa mu naskytne. Potom sa snaží medzi prichádzajúcimi loptičkami nájsť takú, ktorej poloha sa od zvolenej príliš neodlišuje. O tej predpokladá, že ide o jeho loptičku, ktorá sa mu v obraze mierne posunula. Zabúda teda zvolenú pozíciu a sústreďuje sa na novú, jej podobnú. Pokiaľ sa mu dlhší čas nepodarí nájsť podobnú pozíciu, zvolenú pozíciu zabúda a je pripravený akceptovať prvú inú pozíciu, ktorá sa mu naskytne. Zložitejší je spôsob, akým je táto stratégia implementovaná. Nielenže zvolená pozícia sledovanej loptičky je uložená do *space* ako blok *followedBall* – čím sa agent stáva čisto reaktívnym, ale podstatne je tu využitá i časová platnosť tohto bloku.

5 Teoretické pozadie multiagentového prístupu.

Okrem svojich technických predchodcov (LINDA, Java Space, multiagentové systémy) má uvedená architektúra blízko ku Minského societnému modelu mysle [15] a Brooksovej subsumpčnej architektúre [1]. Našim zámerom pri návrhu AgentSpace bolo, aby sme s týmito dvomi prístupmi boli kompatibilní, tj. aby sme im poskytli vhodnú implementačnú platformu. Uvedieme niekoľko podnetov, ktoré z týchto prístupov vychádzajú:

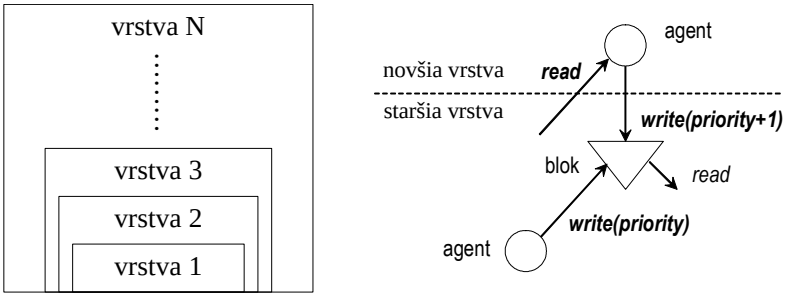
Decentralizácia. Tak, ako požaduje Minsky pre svoj societný model, pri našom prístupe je systém decentralizovaný v tom zmysle, že v ňom neexistuje agent, ktorý by riadil činnosť ostatných agentov. Pokiaľ niektorý z agentov zlyhá, môže to mať síce

neblahý vplyv na globálne správanie systému, ale tento stav nebude generovať žiadnu výnimku, ktorá by zastavila celý systém. Agenti, ktorí nie sú odkázaní na bezprostredné výsledky zlyhaného agenta, sa budú mať ďalej čulo k životu. Pri ich dostatočnom počte a vzájomnom zálohovaní sa dokonca môže stať, že sa zlyhanie jediného agenta v globálnom správaní prejaví len okrajovo, alebo dokonca tvorivo. Táto vlastnosť je daná schopnosťou blokov byť čítaný i zapisovaný viacerými agentmi. Každý blok potenciálne reprezentuje dátový tok many:many, pričom zostava producentov i konzumentov bloku sa môže dynamicky obmieňať (Obr. 9).



Obr. 9. Dátový tok many:many

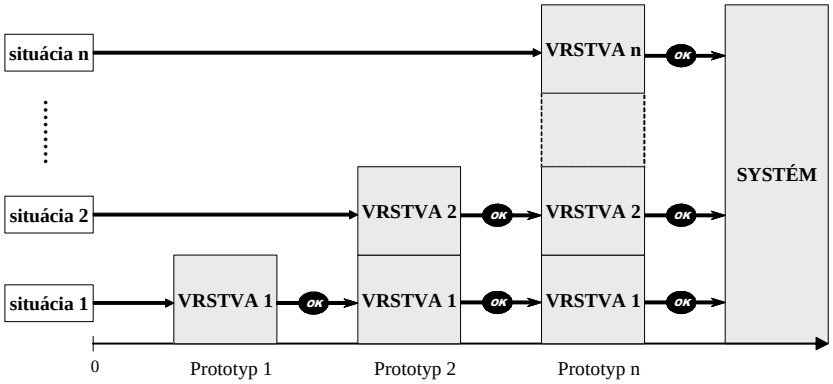
Subsumpcia. Brooks v [1] navrhol podnetný spôsob ako môže vývojár napodobňovať pri vývoji evolúciu bez toho, že by pracoval s nejakou populáciou jedincov. Je založený na fakte, že evolúcia sa odráža vo vnútornom usporiadaní systému, pričom pre zjednodušenie predpokladáme, že v potomkovi nájdeme celú štruktúru jeho predchodcu zaobalenú do nových štrukturálnych vrstiev (Obr. 10 vľavo)



Obr. 10. Subsumpcia a jej realizácia v AgentSpace

Základným predpokladom použitia subsumpcie je vhodná modularita systému, ktorá musí umožniť, aby sa evolučne novšie vrstvy vedeli s koordinovať s evolučne staršími. Pri našom prístupe na to dokážeme využiť zapisovanie bloku viacerými agentmi. Evolučne novší agent môže prepísať hodnotu v evolučne staršom bloku, čím zmení dátový tok medzi dvomi evolučne staršími agentmi. Ten z nich, ktorý blok číta, pritom vôbec nevie rozpoznať, že prečítaná hodnota nepochádza z jeho vrstvy. Takýmto spôsobom sa môže evolučne novšia vrstva votrieť do činnosti evolučne starších vrstiev: prečíta si z nich obsah blokov a prípadne ich poprepisuje tak, aby tieto vrstvy manipulovala k špecifickej činnosti. Pre tento trik je však dobré zaviesť ďalšiu schopnosť nášho *space* a to prioritný zápis, aby novší agent nemusel súperiť so starším

– normálne by totiž záležalo len od okamihu zápisu a čítania, či konzument bloku prečíta hodnotu zapísanú starším alebo novším producentom. Preto zapisovateľ okrem mena, hodnoty a časovej platnosti môže pri zápise definovať aj prioritu zapísanej hodnoty. Zápis hodnoty s menšou prioritou potom *space* ignoruje (Obr. 10 vpravo), pričom zapisujúcemu agentovi sa zdá, že prebehol v poriadku.



Obr. 10. Vývoj situovaného systému pomocou subsumpcie

Situovanosť. Nami vytvorený systém je situovaný v tom zmysle ako požadoval Brooks [2], t.j. do jeho konštrukcie sa premietajú parametre konkrétneho prostredia, pre ktoré systém funguje, ako aj parametre senzorov a motorov systému. Odráža to napríklad množstvo tolerančných konštánt, ako i snaha spojazdniť najprv systém v jednoduchšej situácii a potom túto situáciu stupňovať. Tento prístup je vhodný najmä pre také aplikácie, kde je jasné, čo od systému očakávame, ale nedáva nám to ani najmenšiu nápovedu ako to implementovať. Typickým príkladom takej aplikácie je autonómne vozidlo: vieme povedať kedy jazdí správne a kedy zráža chodcov na prechode, ale to len málo osvetľuje, aká by mala byť jeho riadiaca štruktúra. Situáciu v ktorej tento systém má fungovať, preto na úvod výrazne zjednodušíme a postupne budeme stupňovať, pričom na ďalší kvalitatívny stupeň sa budeme posúvať pomocou subsumpcie (Obr. 11). V jednoduchšej situácii necháme systém konať prostredníctvom evolučne starších vrstiev. V zložitejšej situácii sa však ozvú novšie vrstvy, votrú sa do činnosti starších, vymažú čo je nesprávne, prepíšu čo sa dá opraviť a sami pridajú svoj diel akcií tak, aby odozva celého systému na danú situáciu bola adekvátna.

6 Záver

Multiagentový prístup k programovaniu riadiacich štruktúr robotov a iných systémov umelej inteligencie je zaujímavou inšpiráciou na prenos našich predstáv o fungovaní mysle do technickej praxe. Vychádza z už pomerne starých myšlienok o modulárnom usporiadaní a postupnom vývine mysle [7] [15], ktoré vyjadruje novým spôsobom a to prostriedkami, ktoré boli vyvinuté v oblasti multiagentových systémov [18]

a v koordinačnom programovaní [3]. Na príklade konkrétnej architektúry AgentSpace [13], ktorá je výsledkom nášho dlhoročného snaženia, sme ukázali, ako sa takéto systémy programujú a aké výhody a nevýhody to prináša. Hlavnou výhodou je tu schopnosť postupného rozširovania vyvíjaného systému a podpora jeho práce v reálnom čase. Nevýhodou sú zvýšené nároky na výpočtový výkon. Každopádne, pokiaľ chce niekto vybudovať komplexný systém, ktorý sa čo len vzdialene podobá v istých ohľadoch mysli, musí byť na takéto investovanie pripravený. Našou architektúrou sme implementovali niekoľko akademických i komerčných systémov a jej formu už považujeme za ustálenú.

S ohľadom na zložitosť prezentovaného materiálu sme pri našom výklade použili metodologický prístup, ktorý na konkrétnej aplikácii vysvetľuje jednotlivé mechanizmy tak, aby bolo jasné, prečo sme ich navrhli práve tak, ako sme ich navrhli. Dúfame, že sme nás postup zvolili správne.

Poďakovanie: Táto kapitola vznikla za podpory grantovej agentúry VEGA v rámci grantovej úlohy VEGA 1/0280/08.

Literatúra

[1] Brooks R.: *Intelligence without representation*. Artificial Intelligence 47, 1991, pp. 139-159

[2] Brooks, R.: *Cambrian Intelligence*, The MIT Press, Cambridge, Mass., 1999

[3] Brooks, R.: *Robot – The Future of Flesh and Machines.* Penguin Books, London, 2002

[4] Ciancarini P., Rossi D.: Jada: *Coordination and Communication for Java Agents.* In Mobile Object Systems: Towards the Programmable Internet (Vitek J., Tschudin C. eds.), LNCS Volume 1222, 213-228, Springer-Verlag, Berlin, 1997

[5] Davies, E. R.: *Machine Vision: Theory, Algorithms, Practicalities.*, Elsevier, 2005

[6] Doran J.: *Distributed AI and its Applications. InL Advanced Topics in Artificial Intelliegence* (Mařík V., Štepánková O., Trappl R., eds.), Springer-Verlag, Berlin, 1992

[7] Fodor J.: *The Modularity of Mind*. The MIT Press, Cambridge, Mass., 1983

[8] Gelernter D.: *Generative Communication in LINDA.* ACM on Transactions on Programming Languages and Systems, Volume 7(1), 80-112, 1985

[9] Jennings N.: *On agent-based software engineering*. Artificial Intelligence 117, 2000, pp. 277-296

[10] Kelemen, J.: *Multiagent symbol systems and behavior-based robots*. Applied Artificial Intelligence 7, (1993), 419-432.

[11] Kelemen, J.: *The Agent Paradigm.* Computing and Informatics, Vol.22. (2003), pp. 513-519

[12] Lúčny, A.: *Reaktívny model inteligentného systému.* In: Kognitívne vedy II (Kvasnička, V. – Pospíchal, J., eds.), CHTF STU Bratislava, 1999, 75-84

[13] Lúčny, A.: *Building Complex Systems with Agent-Space Architecture.* Computing and Informatics, Vol. 23 (2004), pp. 1001-1036

[14] Lúčny, A.: *Od medzimodulových spojení k nepriamej komunikácii medzi agentami.* Znalosti, VŠE Ostrava, 2007.

[15] Minsky, M.: *Society of Mind.* Simon & Schuster, New York, 1986

[16] Sonka, M. – Hlaváč, V. – Boyle, R.: *Image processing, analysis and machive vision*. 3rd edition, Thomson Learning, Toronto, 2007

[17] Waldo J.: *Mobile Code, Coordination and Changing Networks.* Concoord, Lipari, 2001

[18] Wooldridge M.: *Introduction to MultiAgent Systems*. John Wiley & Sons, 2002.