Od medzimodulových spojení k nepriamej komunikácii medzi agentami

Andrej Lúčny[1]

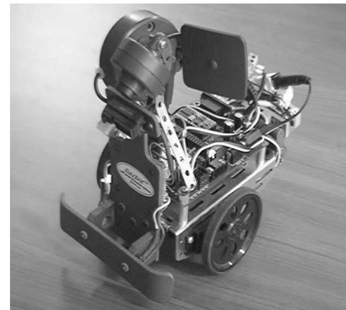
[1]Katedra aplikovanej informatiky, FMFI UK Bratislava
MicroStep-MIS, Čavojského 1, 84104, Bratislava
andy@microstep-mis.com

Abstrakt. Moderný spôsob riadenia robotov sa opiera o modulárny software, v ktorom je vždy výstup z jedného modulu smerovaný do vstupu modulu iného, čím sa naprogramuje ako sa vstupy zo senzorov menia na výstupy do aktuátorov. Tento spôsob je vhodný pre vizuálne a komponentové programovanie avšak z hľadiska prepojenia medzi modulmi iba kopíruje tradičný spôsob usporiadania hardware. Prezentujeme ako sa možno od neho odpútať a aké výhody to prináša. Navrhujeme vlastnú architektúru agent-space a na príklade z kognitívneho videnia porovnávame jej schopnosti oproti tradičným riešeniam.

Kľúčové slová: agent-space, robotika, kognícia, sw. architektúry

1 Úvod

Riadiace systémy robotov sú dnes v drvivej miere realizované softwarovo, ale z ideového hľadiska ich modularita stále slepo napodobňuje hardwarové zapojenie. Pritom je zjavné, že software by mohol dovoliť viac, ako aj to, že dnes používané architektúry nestačia na urobenie rozhodujúceho kroku, ktorý čisto technické spracovanie informácie obohacuje o kognitívnu zložku. V súčasnosti to dobre vidno na posune od *machine vision* ku *cognitive vision*, z ktorého čerpáme (jednoduchý) príklad a demonštrujeme, aké je dôležité zaoberať sa samotnou podstatou tvorby software, aby sme dokázali napredovať v tvorbe riadiacich systémov.



Obr. 1. Mobilný robot vybavený dvojkolesovým podvozkom a kamerou

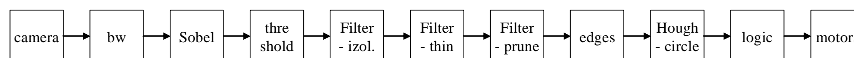
2 Tradičné riešenie

Vezmime si ako príklad mobilného robota vybaveného kamerou a dvojkolesovým podvozkom, ktorý má sledovať pingpongovú loptičku, ktorou pohybujeme (obr. 1). Už aj pri tejto jednoduchej úlohe, riešenie obsahuje dostatočne veľa rôznych algoritmov na to, aby malo zmysel hovoriť o jeho modulárnom usporiadaní. Základná logika je síce takmer atomická:

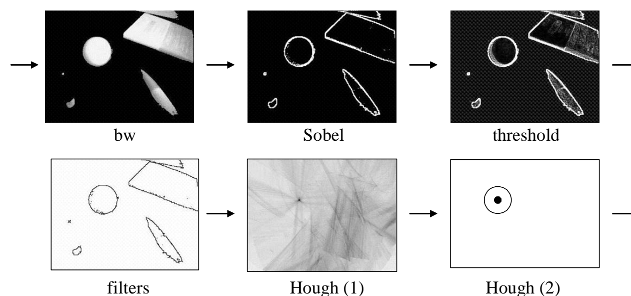
- ak je loptička na obraze moc vpravo, vydávame príkaz na otáčanie sa doprava
- ak moc vľavo, doľava
- ak je loptička moc veľká, cúvame
- ak moc malá, ideme dopredu

ale už samotné rozpoznanie stredu a polomeru vhodného kruhového objektu na obraze, ktorý reprezentuje loptičku pozostáva z viacerých fáz:

- farebný obraz z kamery sa prevedie na čiernobiely
- čiernobiely obraz sa vyhraní, napríklad Sobelovým operátorom
- na základe prahu prevedieme vyhranený obraz na bitovú mapu obsahujúcu hrany
- odstránime izolované body
- hrany stenčovaním premeníme na čiary (*thinning*)
- orežeme ich (*pruning*)
- takto získanú mapu premeníme na množinu úsečiek
- pomocou Houghovej transformácie pre kružnicu získame stredy a polomery kružníc na obraze



Obr. 2. Základ riešenia v tradičnej architektúre



Obr. 3. Medzivýsledky pri spracovaní obrazu

V modernom vývojovom prostriedku[2], programujeme takýto „pipeline“ tým, že na pracovnú plochu postupne umiestňujeme vopred pripravené komponenty, nastavujeme ich parametre a vytvárame spojenia medzi ich vstupmi a výstupmi. Pritom dbáme na zhodu typu údaju, ktorý tieto moduly vysielajú alebo prijímajú.

[2] napríklad iConnect – jedno z komerčných prostredí na programovanie priemyselných robotov

Pritom môžeme použiť jeden komponent viac krát, napríklad odstránenie izolovaných bodov, stenčovanie hrán a ich orezanie sa dá realizovať rovnakým modulom Filter, ktorému ako parameter nastavíme rôzne sady operátorov. (V prípade, že nemáme vhodný komponent k dispozícii, môžeme naskriptovať alebo nakódovať a skompilovať nový). Program potom vyzerá ako ten na obr. 2.

V princípe to funguje a spracovanie prebieha tak ako to vidíme na obr. 3.

3 Problémy pri rozšíreniach

V prípade, že by sme zvolili súčasnú úlohu z priemyselnej robotiky, takéto riešenie by bolo zrejme dostatočné. Na priemyselnej linke sa totiž deje zhruba stále to isté, nemení sa scéna, ktorú vníma kamera, nemení sa ani jej osvetlenie, všetko prebieha v cykle. Avšak v prípade nášho mobilného robota je to inak.

V prvom rade sa musíme vysporiadať s faktom, že pohybom robota sa mení scéna a jej charakteristiky. Z hľadiska načrtnutého spôsobu identifikácie loptičky hrá kľúčovú úlohu napríklad osvetlenie scény, nakoľko od neho závisí vhodný prah pomocou ktorého transformujeme vyhranený obraz (odtiene šedej) na bitovú mapu (biela alebo čierna). Takisto farba a odrazivosť podkladu scény sa môžu meniť, pričom miestami môžu byť podmienky tak zlé, že bežná kamera poskytne obraz na ktorom ani my sami očami nevieme loptičku nájsť.

Tým sa dostávame k ďalšiemu okruhu problémov, ktoré spočívajú v uvedomení si „kognitívnej hĺbky" rozpoznávania loptičky. Musíme vyriešiť problémy ako je splývanie s pozadím a zákryt. Pokiaľ aj loptičku práve nevidíme, dokážeme ju sledovať na základe toho, že vidíme jej časť alebo len na základe toho, že vieme, že sa nemohla vypariť a predvídame, že pokračuje ďalej podobným smerom a rýchlosťou ako doteraz. Musíme ďalej riešiť problém zámeny sledovanej loptičky za iný objekt. V scéne môže byť napríklad viac loptičiek navlas rovnakých ako je tá ktorú sledujeme a vyššie uvedený proces rozpoznania nedáva žiadnu záruku, že si vyberie tú našu, ani že nebude oscilovať medzi viacerými možnosťami.

Navyše sa nedá predpokladať, že by nám bol na niečo užitočný robot, ktorého schopnosti sú obmedzené na sledovanie pingpongovej loptičky. Túto úlohu treba brať skôr ako cvičenie pre tvorbu časti oveľa komplexnejšieho celku. V riešení by sme teda mali vidieť, že je škálovateľné na rozsiahlejšie úlohy. Ako triviálny príklad takého rozšírenia môžeme uvažovať schopnosť robota aktívne vyhľadávať loptičku, čo sa zíde v prípade, keď je na inom mieste scény a na obraze z kamery sa vôbec nenachádza.

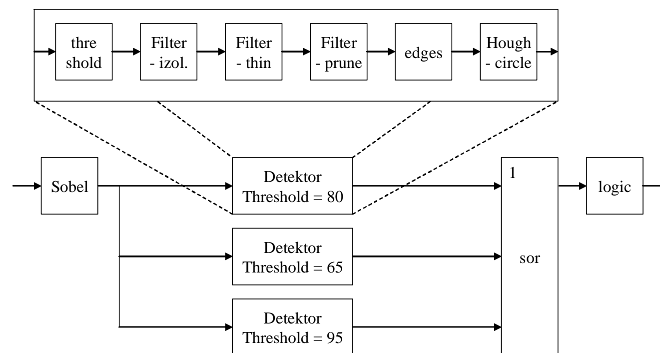
Z okruhov týchto problémov danej úlohy si teraz vyberieme reprezentantov, na ktorých budeme testovať schopnosti tradičnej architektúry:

- zvládnutie rôznej úrovne osvetlenia scény
- zvládnutie viacerých loptičiek v scéne
- zvládnutie sledovania loptičky, ktorá podlieha zákrytu
- zvládnutie aktívneho vyhľadávania loptičky, ktorú nevidno na obraze kamery

Okrem toho dúfame, že táto diskusia navodila dostatočné presvedčenie čitateľa, že i tak jasne definovaná úloha ako je sledovanie loptičky má – pokiaľ sa ide do technických detailov – komplexný charakter. Pod týmto pojmom rozumieme

okolnosť, že sa asi ťažko podarí urobiť systém, ktorý by v sebe zahrňoval všetky potrebné aspekty, na prvý pokus podľa návrhu, ktorý bol urobený na papieri bez toho, že by sa niečo skúšalo s prototypom. Vývoj bude teda prebiehať ako postupnosť modifikácii. Pritom jednotlivé verzie je potrebné na hardwari odskúšať a preto tu nemôže ísť o typický vývoj zhora nadol, ale o vývoj inkrementálny.

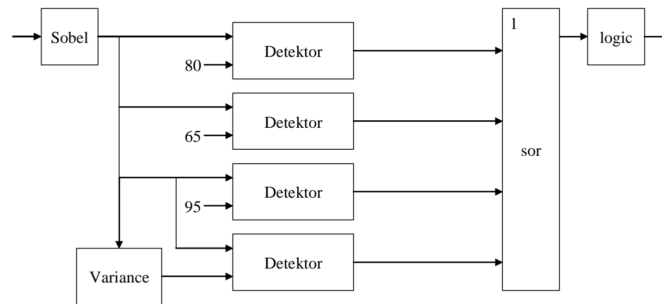
Ako si teraz poradí s vytýčenými rozšíreniami tradičná architektúra? Na zvládnutie rôznej úrovne osvetlenia scény potrebujeme po vyhranení obrazu skúsiť viacero rôznych prahov, prípadne vypočítať vhodný prah rôznymi metódami (napríklad metódou maximálnej variancie), lebo jedna hodnota prahu – hoci šikovne zrátaná – nikdy nezaručí dokonalý výsledok (napríklad keď svetlo svieti zľava a loptička je v tme napravo, maximálna variancia spočítaná pre celý obraz nezaručí, že ju budeme vidieť. Potrebujeme teda v prvom rade sparalelniť fázu od prahovania po rozpoznanie Houghovou transformáciou. Sparalelnenie je ľahké, potiaž spočíva v tom, že rôzne výsledky, ktoré tým vzniknú, musíme nejako zlúčiť dohromady, lebo výsledná logika prijíma len jeden parameter loptičky. Keďže zatiaľ predpokladáme, že loptička je v scéne len jedna, každý z jednotlivých prahov dá buď nič alebo správnu polohu. Takže nám na to stačí vhodný modul *sor* (sorted or), ktorý vráti prvú „nenulovú“ hodnotu. Pritom oceníme možnosť vytvárať nové moduly zložením viacerých existujúcich, čím šetríme námahu a zabezpečujeme si lepšiu modifikovateľnosť (obr. 4).



Obr. 4. Ukážka paralelného spracovania a zložených modulov

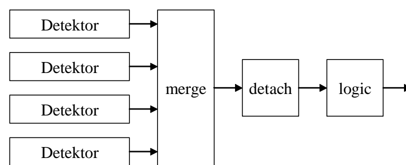
Oveľa viac starostí nám spôsobí zapojenie metódy maximálnej variancie, ktorá poskytuje niečo ako pseudo-optimálny prah. Na riešení na obr. 5 nie je síce na prvý pohľad nič podozrivého, ale v reáli nás bolestne upozorní na problematiku práce takého systému v reálnom čase. Výpočet vhodného prahu je totiž značne pomalý a pri naivnom spôsobe výpočtu odozvy, ktorého takt je daný výlučne frekvenciou snímkovania kamery, sa nám celý systém zadrhne. Hoci ostatné vetvy môžu byť v rozpoznaní loptičky úspešné, modul *sor* sa nedá spustiť, kým mu neprídu všetky nadrôtované vstupy. A jeden z nich veru nepríde, kým sa optimálny prah nezráta. Pritom robot sa v scéne nehýbe až tak prudko aby vôbec malo zmysel počítať optimálny prah pre každý snímok z kamery. Dá sa to samozrejme obabrať tak, že pomalé moduly spustíme v samostatných vláknach a pokiaľ ich výpočet ešte

neskončil, tak príchod vstupnej hodnoty spôsobí len okamžité zopakovanie poslednej výstupnej hodnoty. To už ale rovno môžeme v samostatnom vlákne spustiť každý modul, pamätať poslednú hodnotu na každom výstupe a dať užívateľovi možnosť globálne naplánovať vykonávanie jednotlivých modulov. Toto je vskutku postup ktorý sa používa, pričom plánovač je spravidla do istej miery inteligentný, t.j. užívateľ stanovuje len načasovanie, prioritu, čakanie na určitý vstup a zvyšok zaobstará plánovací program podobný plánovaču procesov v „preemptívnom multitaskingu". Reálne použitie tohto typu konektivity teda zďaleka nie je také priamočiare ako sa môže zo základného konceptu zdať.

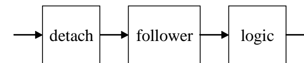


Obr. 5. Ukážka kombinácie pomalých a rýchlych modulov

Prítomnosť viacerých loptičiek v scéne spôsobuje tomuto konceptu ešte vážnejšie problémy. V princípe je totiž problém odviesť z nejakého modulu premenlivé množstvo výstupov, ich počet je pevne daný. A tu by sme potrebovali aby Houghova transformácia poskytla nielen najpravdepodobnejšiu kružnicu, ale hneď niekoľko kružníc. Tradične sa to rieši zavedením balíčkového spracovania dát, t.j. na výstup môžeme poskytnúť balíček nájdených kružníc a na ich zlepovanie využiť moduly určené na spracovanie balíčkov ako sú *merge* (zlepenie viacerých balíčkov do jedného) a *detach* (premenenie balíčka na postupnosť jednotlivých hodnôt) (obr. 6). V princípe však možnosť dynamického vytváranie spojení medzi modulmi chýba.



Obr. 6. Ukážka balíčkového spracovania dát

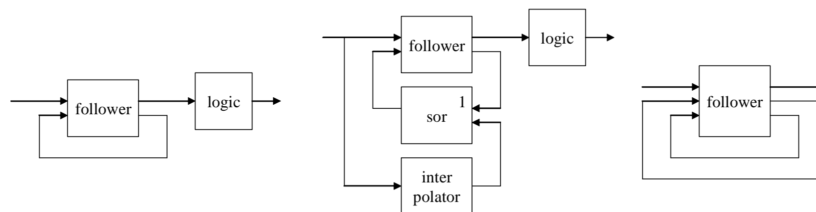


Obr. 7. Vloženie modulu

Na sledovanie jednej loptičky, za prítomnosti viacerých takých v scéne, samozrejme nestačí pustiť do logiky po sebe viac rôznych kandidátov. Musíme jej predradiť modul – nazvime ho *follower* - ktorý z nich vytriedi tých, ktorí zodpovedajú sledovanej loptičke (obr. 7). Tento modul si musí pamätať, ktorého z kandidátov sme sa vlastne rozhodli sledovať a z ďalších prichádzajúcich brať iba tých, ktorí sú s ním

ztotožniteľní, t.j. iba mierne od sledovaného odlišujú. Ak je vhodný kandidát k dispozícii, tak si ním aktualizujeme sledovaného a prepustíme ho na výstup. Inak kandidáta zahodíme.

Ponechať informáciu o sledovanom kandidátovi vnútri modulu nie je práve šikovné. Prídeme na to hneď ako pristúpime k riešeniu zákrytu. Pre prípad úplného zákrytu potrebujeme do logiky púšťať pozíciu loptičky, ktorá nie je získaná zo spracovania aktuálneho obrazu, ale anticipovaná z pohybu loptičky v období keď ju vidno bolo. Potrebujeme teda do sústavy modulov zapojiť modul *interpolator*, ktorý prepúšťa vstup kým mu prichádzajú „nenulové" hodnoty pozície loptičky a dáva na výstup interpoláciu z posledných nenulových hodnôt, kým mu prichádzajú hodnoty nulové. Samozrejme len po určitú dobu, po uplynutí ktorej by už interpolácia bola tak nepresná, že pošle radšej „nulovú" hodnotu. Jednoznačne vychádza, že tento modul musí byť kŕmený hodnotami z *follower*-u, lebo iba tieto sa vzťahujú k pohybu sledovanej loptičky. Lenže ak zaradíme *interpolator* za *follower*, tak sa po opätovnom objavení sa loptičky na obraze, nepodarí *follower*-u loptičku identifikovať s posledným sledovaným kandidátom. Ten sa totiž vyberá z tých, ktorí sú rozpoznaní z obrazu a už riadne ostarel. Modul *follower* treba presvedčiť, že sleduje loptičku aj počas anticipácie. Nedá sa to však urobiť tým, že by sme do vstupov *follower*-u primiešali interpolovanú hodnotu. Tým pádom by sa táto hodnota dostala aj na vstup *interpolator*-u a jeho anticipácia by sa zafixovala na tejto hodnote, čiže úplne by zlyhala. Niet tu inej možnosti ako zvonku manipulovať so zapamätaným kandidátom vo *follower*-i. Museli by sme teda *follower* prerobiť tak, aby mal vstup, ktorým sa dá nastavovať zapamätaný kandidát a *interpolator* by tento vstup musel využívať (obr. 8 vľavo). Iná možnosť je zakaždým tohto kandidáta vypustiť z *followera* a cez oneskorovací modul ho opäť priviesť na jeho vstup. Tým sa zabezpečí, že po ceste je možné ovplyvniť ho hodnotou z *interpolator*-u (obr. 8 v strede). (Toto máme príklad tzv. problému neprístupnosti vnútorného stavu, ktorý sa pri tvorbe modulárnych systémov vyskytuje dosť často).

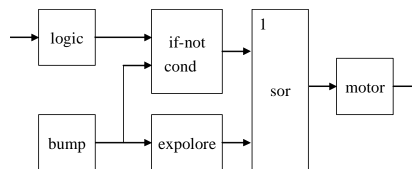


Obr. 8. Ukážka riešenia neprístupnosti vnútorného stavu

Starosti nám pri konštrukcii *follower*-u bude robiť taktiež sledovanie starnutia zapamätaného kandidáta, samozrejme len pokým nezapojíme do činnosti interpoláciu. Čím dlhšie trvá stav keď tento kandidát nie je aktualizovaný, tým naničhodnejšiu informáciu poskytuje. Po čase by bolo vhodné nadobro zabudnúť a zvoliť si na sledovanie inú loptičku, ktorá je k dispozícii. Tradičné architektúry umožňujú sledovať takéto starnutie údajov len na aplikačnej úrovni, t.j. ako ďalší údaj (obr. 8 vpravo). Pritom sa toto starnutie potenciálne týka väčšiny údajov, ktoré sú v systéme vypočítané, (v našom prípade je to predovšetkým snímaný obraz a všetky fázy jeho

spracovania) hoci pri tradičnom spôsobe prepojenia modulov má význam len pre vnútorné údaje v moduloch a pre údaje realizujúce spätné väzby.

Nakoniec prediskutujeme čo je potrebné na rozšírenie tohto systém o aktívne vyhľadávanie loptičky. V prvom rade je nutné pridať celý subsystém realizujúci pohyb robota, ktorý nie je riadený pozíciou guličky, ale realizuje viac-menej náhodný prieskum prostredia a pritom je zabezpečený proti narážaniu do prekážok. To je pomerne ľahké, lebo ako vidno na obrázku 1, náš robot je vybavený aj ľavým a pravým nárazníkom. Tento prieskumný pohyb je však nevyhnutné koordinovať s existujúcim sledovaním guličky. Zdanlivo by na to stačil jeden modul *sor* na výstupe na motory – kým nevidíme loptičku, robíme prieskum, keď ju uvidíme tak ju sledujeme. Lenže za schopnosť sledovať guličku pri zákryte sa platí tým, že ľahko narazíme i v momente keď ju vnútorne vidíme, ale reálne nie. Preto je logický vzťah medzi sledovaním a prieskumom značne zložitejší (obr. 9). Spravidla, čím reálnejší je svet v ktorom sa takýto vzťah odvíja, tým popletenejšia je sústava modulov realizujúcich tento vzťah. O nízkej modifikovateľnosti takéhoto zapojenia nemôže byť pochýb, je sa v ňom proste ťažko vyznať.



Obr. 9. Ukážka kombinácie hierarchicky neusporiadateľných častí

Hoci sme programovanie stanoveného správania robota tradičným spôsobom nakoniec zvládli (koniec koncov teoreticky sa čokoľvek vypočítateľné dá zvládnuť monolytickým programom v strojovom kóde), viac krát tento spôsob povážlivo zaškrípal. Zosumarizujeme si kedy:

- keď mali vstupy do modulu rôznu frekvenciu
- keď boli kombinované pomalé moduly s rýchlymi
- keď bolo treba dynamicky meniť počet vstupov a výstupov
- keď bolo treba časovo ohraničiť platnosť nejakého údaju
- keď bolo treba zabezpečiť netriviálnu koordináciu medzi paralelnými subsystémami

4 Problémy pri rozšíreniach

Kde sú problémy s technológiou tradičnou, otvára sa priestor pre technológie nové. Prvé pokusy na vylepšenie tradičného konceptu badať už v 80-tych rokoch XX. storočia, pričom boli viaceré cesty správne identifikované, nič menej zatiaľ sme sa nestretli s riešením, ktoré by bolo dotiahnuté tak, ako to, čo si teraz predstavíme. Opiera sa o zovšeobecnené agentovo-orientované programovanie. Zovšeobecnené preto, že na rozdiel od analogického pojmu zavedeného Shohamom [10] nepožadujeme, aby agent obsahoval „mentálne“ reprezentácie (hoci môže, niekedy

dokonca na úžitok veci), ale všetky ostatné požiadavky spĺňa. Je to autonómna entita, ktorá opakovane vníma svoje prostredie a na základe toho volí, čo v ňom vykoná. Pritom sa tejto činnosti dá pripísať nejaký cieľ, takže na rozdiel od neurónu z neurónovej siete, o agentovi z multi-agentového systému vieme jasne povedať, akú rolu v ňom hrá. Pri našom nazeraní sa agent však ponáša skôr na stavebný blok mysle, než na myseľ samotnú[3]. Z hľadiska vývoja programovania, možno považovať takýto druh modularity za ďalšiu etapu vývoja programátorských štruktúr od záznamov cez objekty k agentom, kde - na rozdiel od objektov - agenty majú dáta doplnené nielen o kód, ale aj vlastné vlákno jeho riadenia (*program counter*). Okrem toho disponujú nejakým prostriedkom na vzájomnú asynchrónnu komunikáciu (dátovú výmenu).

Vychádzajúc z problémov pri kombinovaní pomalých a rýchlych modulov, je prechod od modulov k agentom dosť prirodzený. Totiž, aby pomalé časti neblokovali rýchle, musíme ich vedieť prerušiť a znovu pustiť, t.j. musia bežať vo vlastnom vlákne. A keď už všetky moduly bežia vo vlastnom vlákne, prečo to nevyužiť na elimináciu špecifického plánovača a vykonávanie modulov prenechať bežnému plánovaču vykonávania vlákien! Ten síce o moduloch nič konkrétne nevie, ale vhodná synchronizácia sa tu dá zabezpečiť tak, že sama povstane z interakcií medzi modulmi vďaka pridaniu mechanizmu vzájomnej dátovej výmeny.

Poďme sa teraz bližšie pozrieť na tento komunikačný mechanizmus. Pokiaľ má riešiť problém s rôznymi frekvenciami vstupov, potrebujeme kód agenta transformujúci vstupy na výstupy spúšťať nielen na základe toho, že sa hodnoty na vstupe aktualizujú (tu budeme hovoriť o trigger-i, spúšťači), ale aj na základe zvoleného časového rytmu, ktorý na vstupoch nezávisí (čo označíme ako timer, časovač). Hodnoty, ktoré majú do určitého modulu vstúpiť, nemusia byť teda spracované hneď ako sú vypočítané a musia teda niekde pretrvať. Ďalej kvôli kombinácii pomalých modulov s rýchlymi je vhodné aby nepretrvávali všetky hodnoty, ale len tie, ktoré modul stíha spracovať. Môžeme prijať zjednodušenie, že pretrvá vždy iba posledná vyprodukovaná hodnota. Ak zariadime, že ani tá nemusí pretrvať naveky, ale automaticky sa po vypršaní určitého času definovaného jej producentom stratí, máme vyriešený aj problém s časovým ohraničením údajov. A ako dosiahneme dynamicky premenlivý počet výstupov? Na to musíme odstúpiť od myšlienky, že spomínané pretrvávajúce hodnoty reprezentujú pevné spojenia. Treba si ich predstaviť skôr ako odkaz na papieriku, ktorý producent podáva a konzument sa pozerá čo je na ňom napísané. Pri takejto predstave nič nebráni tomu, aby producent vydal zo seba viac odkazov, ktorých počet je premenlivý v čase. Konzument potrebuje iba kľúč ako ich všetky referencovať. Komunikovanie pomocou takýchto odkazov nám zároveň rieši problémy s obmedzením sa na časovo posledné hodnoty. Keby ich bolo treba v čase viac, môžeme z nich narobiť viacej časovo posledných hodnôt rozložených v priestore.

Klasické moduly teda v tomto alternatívnom riešení nahradia agenty, ktoré cyklicky vykonávajú výpočet transformujúci určité vstupné údaje na výstupné. Jeden prechod týmto cyklom môže nastať na základe zmeny hodnoty určitého vstupného údaju (*trigger*) alebo na základe uplynutia určitej doby od predchádzajúceho prechodu (*timer*). Komunikácia medzi týmito agentami je obmedzená na nepriamu:

---

[3] čiže filozoficky sa opierame skôr o societný model mysle M. Minského [8], než o Shohamovu interpretáciu, niektorí autori tu používajú aj pojem subagent alebo aj resource

vstupno-výstupné údaje agenty preberajú z a odovzdávajú do odkazov nachádzajúcich sa mimo nich. Tieto odkazy nazývame bloky a detaily spôsobu, akým sa nimi pracuje, sú pre použiteľnosť tohto konceptu rozhodujúce:

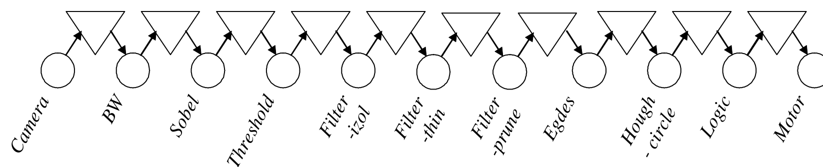
- každý blok má svoje meno, na základe ktorého ho agenty môžu zapisovať alebo čítať. Čítať bloky je možné aj hromadne, na základe masky pozostávajúcej z mena doplneného o značky na porovnávanie regulárnych výrazov (hviezdičková notácia, *regex*)
- blok fyzicky vzniká prvým zápisom, ale čítať sa dá už predtým, t.j. existuje tu niečo ako prázdny blok
- zápisom bloku zaniká predchádzajúca hodnota zapísaná pod rovnakým menom bez ohľadu na to, či ju konzumenti stihli prečítať – konzumenti nemajú žiadnu vedomosť ktorú verziu hodnoty čítajú okrem toho, že je to hodnota najaktuálnejšia
- blok môže mať obmedzenú časovú platnosť, ktorú definuje jeho producent pri jeho zápise

Vďaka týmto vlastnostiam dokážeme prakticky eliminovať potrebu takých modulov ako *sor*, musíme však ešte nejako umožniť určovať prioritu pri viacerých agentoch zapisujúcich do jedného bloku. Pomerne ľahko sa to dá zariadiť tak, že

- pri zápise môže producent definovať prioritu zapisovanej hodnoty, na základe čoho zápisy producentov s nižšou prioritou nebudú mať žiadny efekt, kým platnosť tejto hodnoty nevyprší alebo nie je prepísaná hodnotou s prioritou rovnakou alebo vyššou - producenti pritom opäť nemajú žiadnu vedomosť či svoju hodnotu „pretlačili“ do bloku alebo nie.

Bloky sú teda akýmsi dynamickým pohrobkom pevných spojení medzi modulmi. Nakoľko obrazne hovoríme, že bloky sú v priestore mimo agentov, nazývali sme túto architektúru *agent-space*.

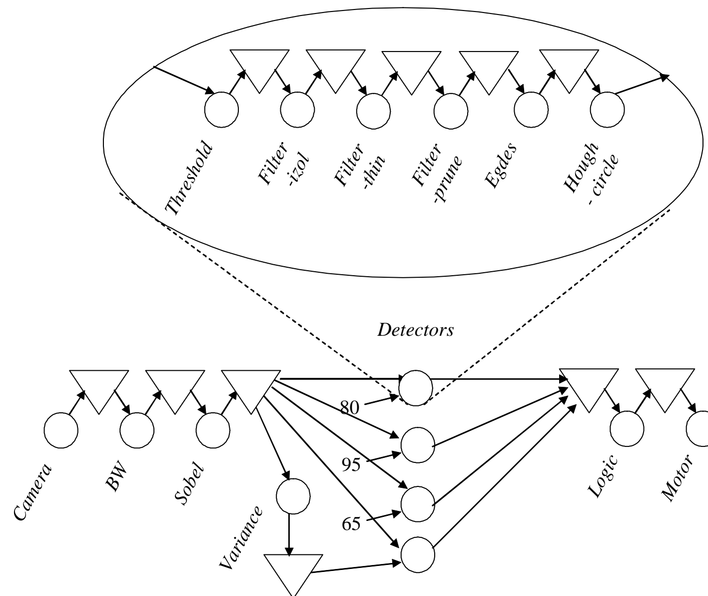
V ďalšom budeme zobrazovať agentov ako ovály, bloky ako hranaté útvary (trojuholníky, štvorce, šesťuholníky), čítanie bude znázorňovať šípka idúca z bloku do agenta, zápis šípka z agenta do bloku. Pre jednoduchosť budeme v schémach abstrahovať od toho, či je agent budený trigger-om (a ktorých blokov sa trigger týka) alebo timer-om či oboma.



Obr. 10. Základ riešenia v architektúre agent-space

Poďme sa teraz pozrieť ako táto architektúra obstojí na zvolenom príklade. V základnom riešení je *pipeline* medzi agentami štrukturálne analogický *pipeline*-u medzi modulmi (obr. 10). Líši sa však operačne. Keby sme „predávkovali“ príliš vysokou frekvenciu vstupných obrázkov tradičné riešenie, tak by sa buď začalo oneskorovať výstupom voči vstupnému obrazu alebo – pri inteligentnejšom plánovači - vzorkovať spracovanie tak, že výpočet cyklicky prebieha celým *pipeline*-om.

Riešenie v agent-space bude v takomto prípade automaticky znižovať frekvenciu aktualizácie blokov v poradí v akom sú zapojené v *pipeline*. Pomalšie moduly tu totiž automaticky pôsobia ako filtre vzorkujúce údaje prechádzajúce cez *pipeline*, na základe toho, že pri dlhšom počítaní výstupu nestihnú prečítať všetky verzie hodnôt na vstupoch. Spôsob preberania vstupno-výstupných hodnôt tu systému poskytuje schopnosť implicitne navzorkovať dátové toky v systéme tak, aby vždy pracoval v reálnom čase.



Obr. 11. Ukážka paralelného spracovania, zložených modulov a kombinácie pomalých a rýchlych modulov v agent-space

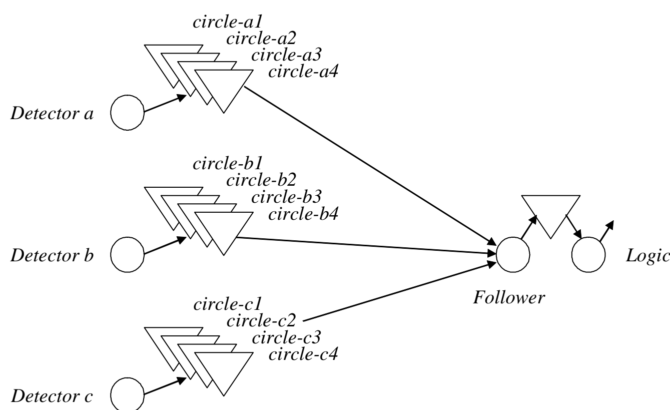
5 Rozšírenia v alternatívnom riešení

A ako to bude vyzerať pri rozšíreniach? Vyskúšanie viacerých prahov na zvládnutie rôznej úrovne osvetlenia scény je z hľadiska sparalelnenia analogické (obr. 11). Vidíme zároveň, že je výhodné mať možnosť urobiť z viacerých agentov jedného, a že konkrétne mená blokov nad ktorými agent pracuje musia byť nastavené až pri jeho použití, aby bol súci ako viacnásobne použiteľný komponent. Zisk je tu hneď dvojaký:

- eliminujeme modul, ktorý slúži na zlúčenie hodnôt produkovaných paralelnými prahmi tým, že ich necháme zapisovať do jedného bloku. V tejto fáze totiž uvažujeme len jednu loptičku v scéne a teda nám vôbec nevadí, ak sa pozícia generovaná jedným prahom, prepíše pozíciou, ktorú detekujeme na základe iného prahu. Podstatné je aby časť, ktorá loptičku nezdetekuje, neprepísala reálnu

pozíciu od iného producenta „nulovou“ hodnotou. Toto pravidlo má, zdá sa, v architektúre agent-space všeobecnú platnosť: ak nevieš čo zapísať, neurob nič (a nechaj platnosť hodnoty v bloku vypršať).

- pomalší modul na výpočet optimálneho prahu nebude brzdiť rýchlejšie pracujúce vetvy. V prípade, že je dosť výpočtovej kapacity, nebude ich brzdiť vôbec, v prípade preťaženia systému len natoľko, koľko je spravodlivé z hľadiska preemptívneho plánovania vlákien.

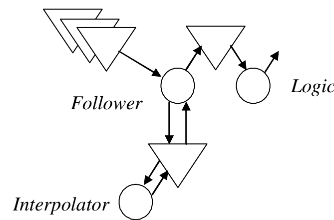


Obr. 12. Ukážka hromadného spracovania blokov v agent-space

Pre prípad viacerých loptičiek v scéne, stačí upraviť agenta realizujúceho Houghovu transformáciu tak, že zapisuje viac blokov a následný agent (vyberajúci, ktorú loptičku sledujeme) bude preberať všetky hodnoty nahromadené rôznymi vetvami (obr. 12). Problém s neprístupnosťou vnútorného stavu obsahujúceho sledovaného kandidáta tu ľahko vyriešime tým, že údaje o sledovanom kandidátovi udržujeme v pomocnom bloku, ktorý pri každom prepočte prečítame, a ak sa ho podarí ztotožniť s niektorým z aktuálnych kandidátov, upravený zapíšeme naspäť. Nepotrebujeme ani oneskorovací obvod, len nesmieme tohto agenta zavesiť na trigger z tohto bloku. Pokiaľ z pomocného bloku nič neprečítame, zvolíme na sledovanie prvého vhodného kandidáta čo príde. Pokiaľ sa nepodarí sledovaného kandidáta ztotožniť s niekým novým pomocný blok necháme bez zmeny, takže po čase jeho platnosť vyprší. To nás automaticky pohne k voľbe nového objektu sledovania. Čas nás tu teda zaujíma len pri zápise pomocného bloku, kedy musíme nadefinovať akú dobu je platný.

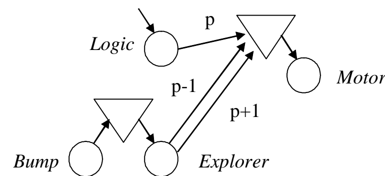
Podobné časové obmedzenie je aj na výstupe z agenta sledovania (*follower*) do logiky a je šikovné urobiť celú vec tak, že tento výstup je aktualizovaný bez ohľadu na to, či sa darí ztotoženie alebo nie. Zákryt potom riešime tým, že pridáme do systému agenta realizujúceho interpoláciu (*interpolator*), ktorý sa výlučne opiera o pomocný blok (obr. 13). Číta z neho údaje, interpoluje ich a predpovedanú polohu zapisuje späť, ale s nižšou prioritou než *follower*. Pokiaľ má *follower* čím blok aktualizovať, tak sa *interpolator* prakticky neprejavuje. Ak však hodnota pochádzajúca od *follower*-a v pomocnom bloku vyprší, objaví sa tam automaticky hodnota od *interpolator*-a (dôležitý technický detail je, že sa musí objaviť hneď, t.j.

blok si musí fyzicky pamätať aj menej prioritné alternatívy, hoci sa pred konzumentmi tvári, že ich nemá). *Follower* je tým uspokojený dostatočne na to, aby túto hodnotu produkoval na výstup do logiky, lebo si namýšľa, že je to hodnota, ktorú si sám pred časom do pomocného bloku zapísal. Kým ju nevie ztotožniť, do práce *interpolator*-a sa nemá dôvod miešať. Interpolátor preto musí dávať pozor na to, aby neinterpoloval príliš dlho. Po čase musí prestať a umožní tak *follower*-u zvoliť si nový objekt sledovania. Môže sa však stať, že sa interpolovanú hodnotu medzitým podarí ztotožniť s detekovaným kandidátom a tým pádom je *interpolator* umlčaný. Vidíme, že pomerne jednoduchým individuálnym správaním sa agentov sa vďaka zložitejšej povahe blokov dá vyjadriť relatívne zložité globálne správanie.



Obr. 13. Ukážka riešenia neprístupnosti vnútorného stavu v agent-space

Aktívne sledovanie loptičky si podobne vieme ľahko predstaviť realizované pomocou priorít. Najjednoduchšie je pridať sústavu agentov, ktorá zapisuje prieskumné pohyby s prioritou nižšou ako je výstup z logiky a pohyby na vyhýbanie sa prekážkam s vyššou prioritou (obr. 14). Tým sa zabezpečí, že ak hrozí náraz, vyhýbame sa prekážkam. Ak však nehrozí náraz a systém loptičku vidí alebo anticipuje, sledujeme loptičku. Inak začneme prieskumné pohyby. Vyťažili sme tu zo schopnosti architektúry kombinovať dve časti systému tak, že raz má prioritu prvá pred druhou a inokedy druhá pred prvou, podľa potreby.



Obr. 14. Ukážka kombinácie hierarchicky neusporiateľných častí v agent-space

Môžeme teda zosumarizovať, že na vybraných problémoch poskytla naša alternatívna architektúra elegantnejšie riešenie. Na základe skúsenosti s jej používaním pre ďalšie systémy, môžeme konštatovať, že tento jav sa netýka len tohto príkladu, hoci pre nedostatok miesta to nemôžeme dokazovať.

6 Záver

V tomto príspevku sme predstavili nami vytvorenú architektúru agent-space, ktorá je alternatívou k tradičnej modularite riadiacich systémov robotov. Upozornili sme na fakt, že dnes úplne samozrejmá čisto softwarová realizácia tohto riadenia umožňuje viac, než len napodobňovanie modularity hardwarových obvodov. Nepriamo sme pritom otvorili také základné otázky programovania ako sú „čo je premenná?“ a „ako jeden kus kódu volá druhý?“. Ukázali sme aký prínos má uvažovať u premennej obmedzenú časovú platnosť a prioritu a aký význam má posunúť sa v myslení od modulov - objektov k modulom – agentom, t.j. od reaktivity k tzv. proaktivite.

Tieto nápady nám nepadli z neba. Za prvú snahu o vymanenie sa z tradičného konceptu je na tomto poli nevyhnutné považovať sumbsumpčnú architektúru od R. Brooksa [1], ktorý pochopil, že sa niečo musí robiť práve so spojeniami medzi modulmi. Nakoľko v tej dobe boli tieto moduly realizované hardwarovo, nemohol s nimi robiť čokoľvek, takže ostal pri vkladaní supresorov a inhibítorov do týchto spojení. Ako správne poukázali Payton a Rosenblat [9], zastavil sa takto na pol ceste, neprekonal napr. problém potreby dynamicky meniť prioritu rôznych častí systému. Zaujímavé je, že už v jeho dobe pritom existoval v paralelnom programovaní koncept nepriamej komunikácie LINDA, z ktorého sme čerpali spôsob komunikácie medzi agentami. Pravda, museli sme ho mierne modifikovať. Obmedzili sme ho na asynchrónny režim a odstránili sme nezmyselnú požiadavku volať pred prvým zápisom primitívu na vytvorenie bloku, ktorá bráni realizovať dátové toky od mnohých producentov k mnohým konzumentom. Použijúc moderný jazyk agentovo orientovaného programovania sme potom tieto staré ale hodnotné myšlienky pretavili do modernej a ucelenej podoby.

Vzhľadom na to, že každý agent realizuje v takomto systéme určitú zafixovanú reakciu (nejaký reflex), môžeme architektúru agent-space považovať za snahu implementovať správanie systému prostredníctvom skupiny reflexov nad vnútornou pamäťou s ohraničenou trvanlivosťou. Tým naznačujeme, že je tu istý vzťah k problému ako povstáva fenotyp z genotypu, nakoľko tu z jednoduchých vecí môžu povstávať zložité. V tomto článku sme pomlčali o biologickej motivácii, ale ona je pre nás veľmi dôležitá. V podstate, ak je vôbec slušné povedať, že príroda programuje svoje výtvory, tak my k tomu dodávame, že používa trochu iné premenné a inak organizovaný kód, než si bežne pri programovaní predstavujeme. S tým asi bude súhlasiť takmer každý, my ale máme určitú konkrétnu predstavu aký charakter tieto premenné a kód majú.

Tento článok sme sa snažili písať samonosne, porovnávajúc agent-space s tradičným prístupom na konkrétnom príklade. Snažili sme sa predstaviť našu architektúru vo svetle prostriedku užitočného pre tvorbu robotických systémov, ktoré majú vysoké nároky na komplexnosť, čo sa pozitívne prejavuje pri rozširovaní ich kognitívnych schopností. Kognitívne schopnosti takýto systém dokáže prejavovať tým, že do svojich vnútorných blokov ťaží z prostredia informáciu, ktorá spôsobuje, že globálneho hľadiska sa lepšie správa. Je to len prvý krok, ale snáď budeme pokračovať.

Referencie

1. Brooks, R.: *Cambrian Intelligence*, The MIT Press, Cambridge, Mass., 1999.
2. Brooks, R.: *Robot – The Future of Flesh and Machines.* Penguin Books, London, 2002
3. Davies, E. R.: *Machine Vision: Theory, Algorithms, Practicalities*., Elsevier, 2005
4. Kelemen, J.: *The Agent Paradigm.* Computing and Informatics, Vol.22. (2003), pp. 513-519
5. Lúčny, A.: *Building Complex Systems with Agent-Space Architecture.* Computing and Informatics, Vol. 23 (2004), pp. 1001-1036
6. Lúčny, A.: *Building Control System of Mobile Robots with Agent-Space Architecture*. CLAWAR/EURON Workshop, Vienna, 2004
7. Lúčny, A.: Architektúra A*gent-Space.* doctoral theses. FMFI UK Bratislava, 2005
8. Minsky, M.: *The Society of Mind.* Simon&Schuster, New York, 1986
9. Rosenblatt, J. K. - Payton, D. W.: *A Fine-Grained Alternative to the Subsumption Architecture for Mobile Robot Control.* In: Proceedings of the IEEE/INNS International Joint Conference on Neural Networks, Washington DC, vol. 2, 1989, pp. 317-324
10. Shoham, Y.: *Agent-oriented programming.* Artificial Intelligence, 60(1):51-92. (1993)

Annotation:

From inter-module wires to indirect communication among agents

This paper criticizes traditional approach to modularity of software-based control systems of robots which just mimics hardware layouts. It proposes a novel architecture for building of control systems which replaces modules by agents and wires by indirect communication among them. The architecture is compared with traditional approach on particular example from domain of cognitive vision. Mechanisms of variables with bounded persistence, priorities and implicit real-time support are underlined.