Riadenie simulovaného robota iCubSim pomocou architektúry Agent-Space

Andrej Lúčny - Matúš Kopernický

Katedra aplikovanej informatiky, Fakulta matematiky, fyziky a informatiky, Univerzita Komenského
Mlynská dolina, 842 48 Bratislava
lucny@fmph.uniba.sk, matus.kopernicky@microstep-mis.com

Abstrakt

K humanoidnému robotu iCub je voľne dostupný pomerne kvalitný simulátor iCubSim. Podarilo sa nám do neho implementovať riadenie na báze architektúry Agent-Space. Táto architektúra bola navrhnutá dávnejšie ako jedna z možných implementácii Minského predstavy o mysli ako spoločenstve agentov. Z technického pohľadu vychádza z Brooksovej subsumpčnej architektúry, ale spôsob komunikácie modulov systému je realizovaný na spôsob multi-agentového systému so stigmergickou komunikáciou. Funkčnosť riadenia je demonštrovaná mimo iné na príklade ovládania hlavy a očí robota sledujúceho pohyblivú loptičku na stole, ktorý je výsledkom interakcie ôsmich pomerne jednoduchých agentov.

1 Úvod

V roku 2006 založená iniciatíva EU Cognition si stanovila za cieľ združovať výskumníkov v oblasti kognitívnej vedy a vybaviť ich technickými prostriedkami, ktoré by umožnili vzájomnú spoluprácu a hlavne nezávislé overovanie výsledkov. Najvýznamnejším činom tejto iniciatívy bol integrovaný projekt RoboCub v rámci ktorého bol vyvinutý humanoidný robot iCub, ktorým bolo vybavených 14 výskumných pracovísk v EU (a ďalšie postupne pribúdajú). V minulosti boli totiž spravidla výsledky dosiahnuté na nejakom humanoidnom robotovi zložitejšej konštrukcie neoveriteľné, lebo nik iný nedisponoval rovnakým hardvérom. Nasadenie iCub-u túto situáciu zmenilo. Aby bol dosah tejto akcie ešte väčší, bol taktiež vyvinutý open-source simulátor iCubSim, ktorý je - napriek chybám, ktoré má - celkom dobrou aproximáciou reálneho sveta. Myšlienka za tým je tá, že pokiaľ niekto urobí zaujímavý výsledok v simulátore, môže sa kontaktovať s niektorým vlastníkom iCubu a overiť svoje pokusy v reáli.

So simulátorom iCubSim bol uskutočnený celý rad zaujímavých experimentov, viď www.icub.org. Časť z nich smerovala aj k výskumu riadiacich architektúr a to zväčša na báze neurónových sietí, napríklad (Peniak 2013).

Simulátor sa používa tak, že sa naprogramuje aplikácia, ktorá cez TCP ovláda jednotlivé senzory a aktuátory simulovaného iCubu. Tento simulátor sme teda doplnili o aplikáciu, ktorá implementuje našu riadiacu architektúru Agent-Space (Lucny 2004). To znamená, že používateľ môže do nej vložiť svoju sadu agentov, z interakcie ktorých riadenie povstáva v zmysle (Minsky 1986), (Books 1999), (Kelemen 2003). To si žiada taktiež prekryť senzory a aktuátory ovládaním, ktoré je kompatibilné s touto architektúrou.

Túto implementáciu sme potom overili na jednoduchej úlohe sledovania objektu podľa farby. Táto úloha však nie je triviálna natoľko, aby pri nej nevynikli jednotlivé aspekty riadiacej architektúry.

2 Simulátor iCubSim

iCubSim bol vyvinutý v C++. Dá sa inštalovať aj na MS Windows, ale jeho primárna platforma je Linux. My sme použili Ubuntu, kde sa nainštaluje veľmi ľahko:

```
# sudo sh -c 'echo "deb http://www.icub.org/ubuntu trusty contrib/science" > /etc/apt/sources.list.d/icub.list‘
# sudo apt-get update
# sudo apt-get install icub
```

(trusty tu znamená len verziu Ubuntu). Okrem toho je treba v /usr/share/icub ... context/simconfig/ iCubPartsActivation.ini nastaviť objects = on a pre bežný počítač ode_param - timestamp na cca 35 (default 10 vyžaduje výkonný hardware).

Z ďalších prostriedkov potrebujeme kompilátor GNU GCC, cmake a nejaký editor pre C++ alebo IDE ako QtCreator.

Jadrom simulátora je rovnaká softvérová platforma akú používa iCub - Yet Another Robotics Platform (YARP). V prípade simulátora je však miesto senzorov a aktuátorov robota prepojená na analogickú virtuálnu realitu vo fyzikálnom engine Open Dynamics Engine (ODE).

Simulátor iCubSim implementuje virtuálneho robota, ktorý má rovnaké parametre ako robot iCub: je 105cm vysoký, váži 20,3kg a má 53 DoF (degree of freedom), tj. stupňov voľnosti, ktoré zahŕňajú 12 kontrolovateľných DoF na nohách, 3 kontrolovateľné DoF na torze, 32 DoF pre ruky a 6 DoF pre hlavu. Robot taktiež obsahuje rôzne druhy senzorov ako

kamery v očiach, mikrofóny a tlakové senzory (zodpovedajúce hmatu po celom tele).

Prepojenie senzorov a aktuátorov s riadiacou aplikáciou je realizované cez TCP server yarpserver, ktorý implementuje komunikačný protokol YARP. Tento server len sprostredkúva komunikáciu medzi jednotlivými klientami, pričom jedným z nich je simulátor virtuálnej reality na báze OpenGL a ODE s modelom robota iCub a prípadne zopár ďalšími predmetmi (stôl, loptička, ..., atď). yarpserver pracuje na TCP porte 10000 a umožňuje klientským aplikáciám nadviazať spojenie cez niekoľko ďalších portov tesne nad 10000 a komunikovať vzájomne YARP protokolom. Ten si možno predstaviť ako ďalšiu transportnú vrstvu umožňujúcu dátovú výmenu medzi jednotlivými modulmi klientských aplikácií (v prvom rade medzi riadiacou aplikáciou a jednotlivými senzormi a aktuátormi simulovaného robota). Je to ako keby sme nad TCP vybudovali ešte jeden protokol podobný UDP, v rámci ktorého zavádzame vlastné porty, tzv. YARP porty, akurát ich identifikátorom nie je číslo, ale text podobný ceste k súborom na disku - napríklad kamere ľavého oka je priradený YARP port /icubSim/cam/left. Yarpserver sa spustí napríklad:

```
# gnome-terminal -e yarpserver
```

Samotný simulátor sa potom spustí ako ďalší proces:

```
# gnome-terminal -e iCub_SIM
```

YARP protokol umožňuje prepojiť tok dát medzi dvomi YARP portami. Takže keď spustíme aplikáciu na zobrazovanie obrazu a priradíme jej nami definovaný port /camera:

```
# gnome-terminal -e "yarpview /camera"
```

neukazuje nič, kým do nej nepresmerujeme dáta z kamery v yarpserveri, ktorá beží na porte /icubSim/cam/left, ktorý je definovaný v simulátore iCubSim:

```
# yarp connect /icubSim/cam/left /camera
```

Na každý YARP port sa pritom môže pripojiť viacero aplikácií, takže tým, že si obraz z ľavého oka zobrazíme v yarpview, nestrácame možnosť brať jeho hodnotu rovnakým spôsobom do riadiacej aplikácie.

Architektúra riadiacej aplikácie je ponechaná na jej vývojára, ktorý zo svojho programu volá funkcie klientskej knižnice implementujúcej YARP protokol. Akú architektúru zvolí, je teda len na ňom – spravidla píše jednovláknový program synchrónne volajúci spomínanú klientskú knižnicu. Obraz z kamery ľavého oka získame napríklad takto:

```
localPort = "/left/image/in";
imagePort = new  yarp::os::BufferedPort
        <yarp::sig::ImageOf <yarp::sig::PixelRgb>>();
imagePort->open(localPort);

remotePort = "/icubSim/cam/left";
yarp::os::Network::connect(remotePort,localPort);

yarp::sig::ImageOf<yarp::sig::PixelRgb> *image;
image = imagePort->read();
if (image != NULL) process(image);
```

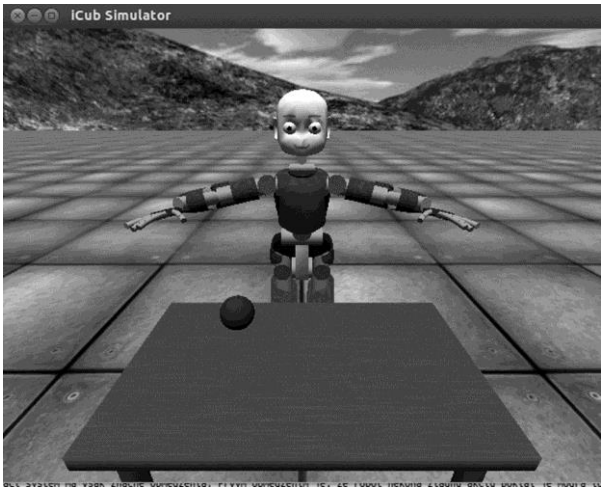
Podobne pošleme príkaz na motor:

```
yarp::dev::PolyDriver driver;
yarp::dev::IPositionControl *pos;
yarp::dev::IVelocityControl *vel;
yarp::dev::IEncoders *encs;
yarp::dev::IControlLimits *iLim;
yarp::sig::Vector speeds;
yarp::sig::Vector accelerations;
string Device = "remote_controlboard";
string LocalPort = "/test/client";
string RemotePort = "/icubSim/head";
yarp::os::Property options;
options.put("device", Device);
options.put("local", LocalPort);
options.put("remote",RemotePort);
driver.open(options);
driver.view(encs);
driver.view(pos);
driver.view(iLim);
// compute speeds & accelerations
pos->setRefAccelerations(accelerations.data());
pos->setRefSpeeds(speeds.data());
```

Keď implementujeme nejaké konkrétne správanie, musíme napísať takú aplikáciu, ktorá zo sekvencie vstupov zo senzorov a propriocepcie počíta v reálnom čase adekvátne výstupy na aktuátory.

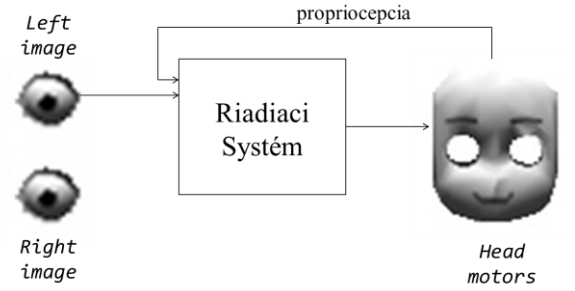
3 Modelová úloha

Za modelovú úlohu sme si zvolili sledovanie loptičky danej farby, upriamenie pozornosti na ňu a jej udržanie v zornom poli robota (obr. 1).



Obr. 1: Virtuálna scéna s iCubom a objektmi

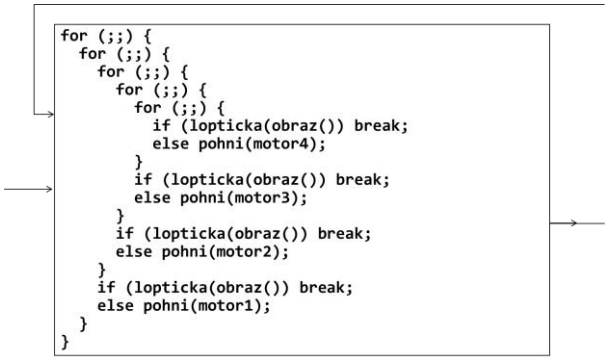
Keď túto úlohu premietneme do vzťahu senzorov a aktuátorov, ide o spracovanie obrazu z oka (berieme do úvahy len jedno) na pohyb štyroch motorov: dva hýbu hlavou zľava doprava a zhora dole a dva hýbu očami zľava doprava a zhora dole (obr. 2).



Obr. 2: Riadiaci systém vo vzťahu k senzorom a aktuátorom.

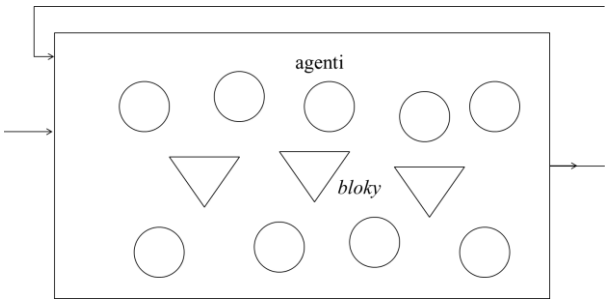
4 Riadiaca architektúra Agent-Space

Pokiaľ by sme sa modelovú úlohu snažili vyriešiť tradičným spôsobom, v jednom vlákne by sme nejako cyklicky hýbali motory, až kým by sa loptička nedostala do správneho zorného poľa. Približný kód takého programu vidíme na obr. 3.

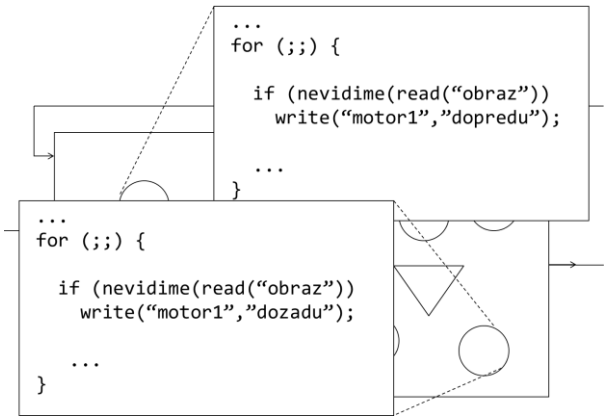


```
for (;;) {
  for (;;) {
    for (;;) {
      for (;;) {
        for (;;) {
          if (lopticka(obraz()) break;
          else pohni(motor4);
        }
        if (lopticka(obraz()) break;
        else pohni(motor3);
      }
      if (lopticka(obraz()) break;
      else pohni(motor2);
    }
    if (lopticka(obraz()) break;
    else pohni(motor1);
  }
}
```

Obr. 3: Tradičný radiaci systém



Obr. 4: Elementy riadenia podľa architektúry Agent-Space



```
...
for (;;) {

  if (nevidime(read("obraz"))
    write("motor1","dopredu");

  ...
}
```

```
...
for (;;) {

  if (nevidime(read("obraz"))
    write("motor1","dozadu");

  ...
}
```

Obr. 5: Povstávanie riadenia z viacerých jednoduchších správaní

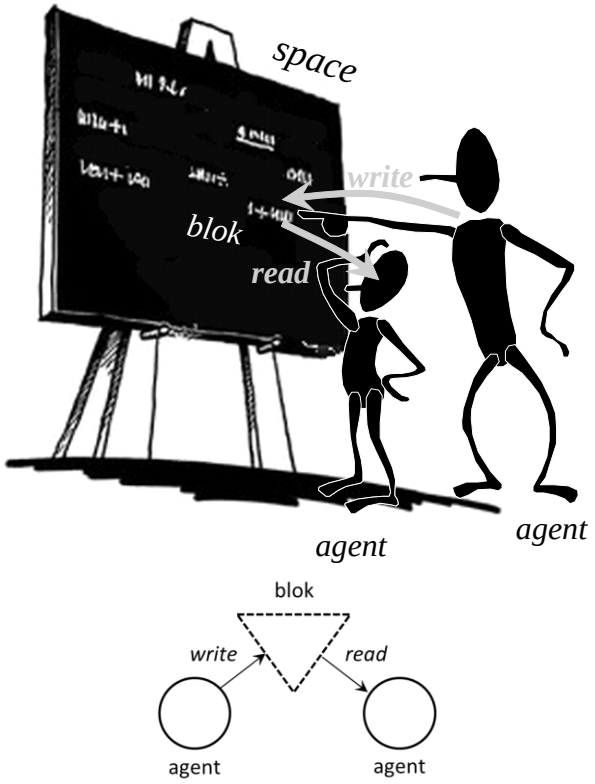
Aby pohyb hlavy a očí nebol príliš mechanický a drevorubačský, prispôsobovali by sme hýbanie motoru vnútornému stavu systému, ktorý by sme premietli do hodnôt globálnych premenných.

Inovácia nášho prístupu spočíva v tom, že riadiaci systém bude mať viac vlákien, medzi ktorými bude prebiehať interakcia, čiže normalizovaná dátová výmena. Každé z týchto vlákien budeme považovať za samostatne konajúceho agenta, schopného interakcie s ostatnými agentmi prostredníctvom tzv. blokov (stigiem) na čiernej tabuli, či nástenke. Implementujeme tak societu agentov, medzi ktorými prebieha tzv. stigmergická komunikácia (obr. 4). Pritom každý jeden agent je z hľadiska riadenia porovnateľný s celým riadiacim programom pri tradičnom riešení, akurát, že jeho kód by mal byť podstatne jednoduchší – čo je zmyslom takejto dekompozície riadenia. Programy agentov sa nemusia vždy len dopĺňať, môžu taktiež súperiť, t.j. pokúšať sa uskutočniť aj úplne protichodné akcie na aktuátoroch (obr. 5).

Každý z týchto agentov má vlastné riadenie nezávislé na ostatných agentoch a vykonáva vlastný kód, pričom tento beží v nekonečnom cykle. Pri každom prechode týmto cyklom (ktorý vyvoláva určitá udalosť a to buď impulz od časovača alebo dostupnosť nových informácií) sa na základe vnímania aktuálnej situácie a ďalších informácií uložených v tzv. vnútornom stave agenta, vypočíta a vykoná náležitá akcia.

Najvýznamnejšou zložkou vnímania je pre každého agenta komunikácia s inými agentmi. Práve tento komunikačný mechanizmus zaraďuje agenta do systému. V použitej architektúre Agent-Space je tento mechanizmus založený výlučne na nepriamej komunikácii a je realizovaný tzv. čiernou tabuľou (nástenkou), ktorú nazývame space (obr. 6). Prostredie dokáže uskladňovať pomenované dáta, tzv. bloky, ktoré agenti dokážu čítať, zapisovať a mazať. Agenti pritom musia poznať ich mená a formát dát v nich zapísaných.

Detaily tejto architektúry, ktoré je potrebné zvážiť, aby tento koncept fungoval naozaj dobre, sú popísané v (Lucny 2004) (Lucny 2012). Tu spomenieme len základnú povahu zápisu blokov, pomocou ktorej sa dá realizovať dátový tok od mnohých producentov ku mnohým konzumentom.



Obr. 6: Komunikácia medzi agentmi cez space a jej symbolické vyobrazenie

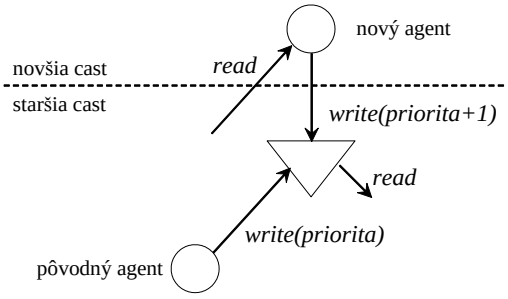
Bloky sa v space vytvárajú prvým zápisom. Čítať ich však možno už predtým a to bez toho, že to vyvolalo nejakú výnimku: pre tento prípad je agent pri čítaní povinný zadefinovať preddefinovanú hodnotu, ktorá sa pri čítaní neexistujúceho alebo prázdneho bloku vráti, ako keby v ňom bola zapísaná.

Blok môže obsahovať iba jedinú hodnotu, takže hodnota zapísaná jednou operáciou zápisu je prepísaná nasledovnou. Hodnota zapísaná v bloku je prepísaná bez ohľadu na to, či ju nejaký agent stihol prečítať alebo nie. Takže ak producent zapíše nasledujúcu hodnotu do bloku prv, ako niektorý konzument stihne blok prečítať, tak tento konzument o túto hodnotu proste príde – z jeho pohľadu nikdy v bloku nebola. Takže ak jeden konzument nie je dostatočne rýchly, aby stihol z bloku prevziať všetky zapisované hodnoty, automaticky tu príde k ich vzorkovaniu, zatiaľ čo druhý ich môže stihnúť prečítať všetky. Vďaka tejto vlastnosti je v systéme ľahké kombinovať pomalé a rýchle moduly.

Pri každom zápise je možné taktiež definovať časovú platnosť zapísanej hodnoty, tj. dobu, po uplynutí ktorej hodnota z bloku automaticky zmizne (čo zariadi space). Pokiaľ časová platnosť pri zápise nie je definovaná, hodnota ostáva v bloku natrvalo, respektíve do najbližšieho prepísania. Časová platnosť je dobrý nástroj na vyjadrenie dočasného charakteru informácie. Navyše podporuje dátový tok od mnohých producentov k mnohým konzumentom. Keď máme napríklad viac metód na výpočet určitej hodnoty, pričom každá vyprodukuje hodnotu len za určitých podmienok, môžeme tieto metódy premeniť na agentov zapisujúcich rovnaký blok. Takto sa konzumenti nemusia starať z akého zdroja hodnota pochádza. Musíme však zariadiť, aby agent, ktorý nič nespočítal, radšej nič do bloku nezapísal, lebo tak by prepísal použiteľnú hodnotu od iného agenta. Potom ale zákonite hrozí, že nik nespočíta nič a v bloku naveky ostatne stará posledná hodnota. Preto v takomto prípade treba zadefinovať jej časovú platnosť.

Okrem časovej platnosti hodnoty, je možné pri zápise hodnoty zadefinovať aj jej prioritu. Toto nám umožňuje preferovať hodnotu od určitého producenta. Priorita zápisu je realizovaná v space tak, že keď sa nejaký agent snaží prepísať v space prioritnejšiu hodnotu, hodnota sa nezmení (pričom on o tom nie je nijako informovaný). Tento mechanizmus je dôležitý, keď sa pri inkrementálnom vývoji snažíme z novo pridaných častí systému ovplyvniť pôvodné (obr. 7), čo je podstata Brooksovej subsumpčnej architektúry (Brooks 1999). Pri agent-space sa novo pridaní agenti miešajú do komunikácie medzi pôvodnými tak, že:

- noví čítajú bloky, cez ktoré si pôvodní vymieňajú informácie (odpočúvanie)
- noví prepisujú hodnotu v týchto blokoch (supresia)
- noví vymazávajú hodnotu v týchto blokoch (inhibícia)



Obr. 7: Supresia starších častí z novších

Vieme teda vyjadriť všetky mechanizmy Brooksovej subsumpcie (odpočúvanie, supresiu a inhibíciu) a dokonca aj komplikovanejšiu koordináciu nových a pôvodných agentov. To nám umožní organizovať komplexnejší systém, ktorého zložité správanie necháme vzniknúť z interakcie jeho jednoduchých častí, ktorých správanie vieme zadefinovať ľahšie.

5 Implementácia Agent-Space do iCubSim

Na implementáciu architektúry Agent-Space do iCubSim je okrem implementácie všeobecného agenta

(implementuje sa ako objekt s vlastným vláknom (kód 1)) a mechanizmu komunikácie medzi agentmi (implementuje sa cez statickú pamäť pomocou synchronizácie vlákien, viď (Lucny 2012)) treba pre každý senzor (vrátane propriocepcie) a aktuátor implementovať blok, pomocou ktorého ho je možné monitorovať či riadiť.

```
#include "agentspace.h"

class MyAgent : public Agent {
 protected:

   void init (string args) {
    //timer_attach(period,period);
    //id=trigger_attach("name");
   }

   void sense_select_act (int id) {
    //value = space_read("name",default);
    //space_write("name",value);
    //space_write("name",value,time_validity);
    //space_write("name",value,time_validity,priority);
   }

 public:
   MyAgent (string args) :
    Agent(args) {};

};
```

Kód 1. Všeobecný kód implementujúci agenta

Pritom sme sa zamerali na podporu takej povahy ovládania senzorov, aby zodpovedala potrebám práce v reálnom čase. Bežne sa totiž pri práci s iCubom i jeho simulátorom ovládajú aktuátory príkazmi typu "zmeň tento kĺb do tejto polohy", zatiaľ čo našim zámerom vyhovuje skôr derivačný pokyn "zmeň smer a rýchlosť pohybu tohto kĺbu takto". Miesto "pohnúť" používame "hýbať" - miesto jedného pokynu aktuátoru, sériu pokynov rozloženú v čase, miesto invokácie, reguláciu. iCubSim nám tu našťastie výrazne pomáha tým, že fyzické limity pohybu kĺbov stráži sám - riadiaci systém si teda môže želať akýkoľvek pohyb, ale ak už je daný kĺb v krajnej polohe, tak sa pohyb neudeje. Vydanie pokynu pre nás pritom znamená zapísanie určitej hodnoty rýchlosti alebo zrýchlenia do určitého bloku v space. Čítaním iného bloku si môžeme overiť skutočnú aktuálnu polohu kĺbu.

Práca so senzormi je o niečo jednoduchšia, lebo stačí zariadiť, aby určitý blok v space obsahoval poslednú hodnotu zo senzora, odkiaľ si ho môžu agenti kedykoľvek prečítať.

Funkčnosť bloku v space, ktorý reprezentuje senzor či aktuátor je najľahšie zabezpečiť agentom, ktorý bude potom súčasťou každej riadiacej aplikácie, ktorá daný senzor či aktuátor používa. Pre našu modelovú úlohu sme potrebovali jednak obraz z kamery oka, čo riešil CamSensAgent (kód 2), jednak ovládanie motorov očí a krku, čo riešil VelocityMotorAgent (kód. 9).

```
void CamSensorAgent::init(string args) {
 remotePort = "/icubSim/cam/left";
 localPort = "/left/image/in";
 blockName = "leftImage";
 startTime = 1000; period = 250;
 imagePort = new  yarp::os::BufferedPort
                <yarp::sig::ImageOf<yarp::sig::PixelRgb>>();
 imagePort->open(localPort);
 yarp::os::Network::connect(remotePort,localPort);
 timer_attach(startTime,period);
}

void CamSensorAgent::sense_select_act(int id) {
 yarp::sig::ImageOf<yarp::sig::PixelRgb> *image
     = imagePort->read();
 if (image!=NULL) space_write(blockName, image);
 else printf("No image\n");
}
```

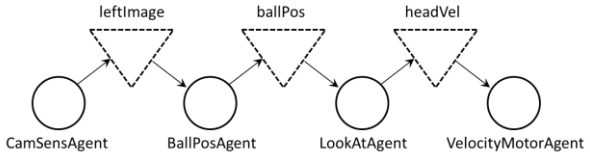
Kód2. Príjem obrazu z kamery do bloku v space

```
void VelocityMotorAgent::init(string args) {
 ...
 string velocityBlock = "head_vel";
 yarp::dev::PolyDriver vel;
 yarp::os::Property options;
 options.put("device", deviceName);
 options.put("local", LocalPort);
 options.put("remote", remotePort);
 vel.open();
 ...
 trigger_attach(velocityBlock);
}

void VelocityMotorAgent::sense_select_act (int id) {
 yarp::sig::Vector commands;
 commands = space_read(velocityBlock, def_commands);
 vel->velocityMove(commands.data());
}
```

Kód 3. Ovládanie motorov podľa bloku v space

6 Implementácia modelovej úlohy

Implementáciu architektúry sme potom otestovali na modelovej úlohe. Pri vývoji jej riešenia sme postupovali v zmysle inkrementálneho vývoja zdola nahor, typického pre Brooksovu sumbsumpčnú architektúru (Brooks 1999).



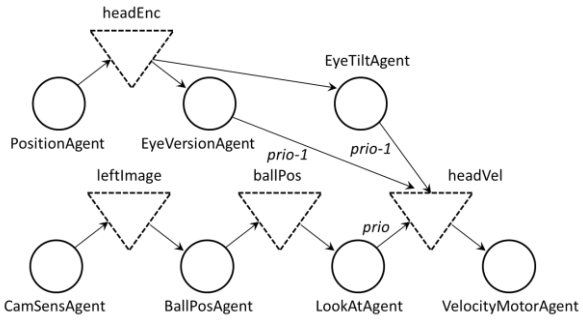
Obr. 8: Vývojová fáza 1

Prvá fáza vývoja rieši základnú situáciu, že robot sledovanú loptičku vidí, ale nemá ju v strede zorného poľa (obr.8). V takom prípade z pozície loptičky na obraze vypočítame vektor rýchlosti. Tu si treba

uvedomiť, že vektor rýchlosti sa vypočíta oveľa ľahšie než výsledná poloha. Tá sa nájde vďaka spätnej väzbe ako poloha v okamihu, kedy je loptička v strede obrazu. Tento prístup môžeme interpretovať ako tzv. stelesnenie (embodiment) - telo robota sa tu podieľa na samotnom výpočte správnej polohy. Riadenie zabezpečujú dvaja agenti: BallPosAgent detekuje polohu loptičky na obraze (kód 4) a LookAtAgent podľa danej polohy voči stredu obrazu určuje smer rýchlosti.

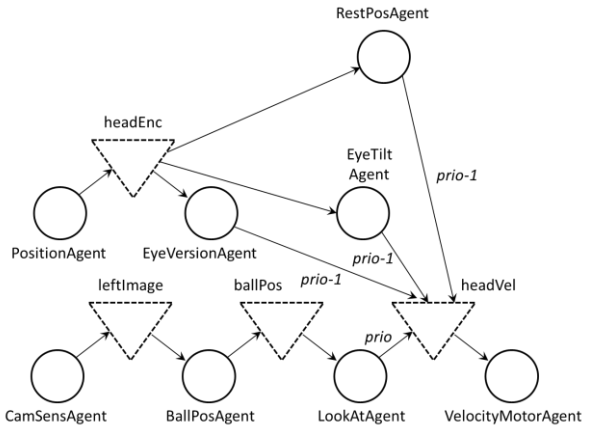
```
void BallPosAgent::init(string args) {
   inBlock = "leftImage";
   outBlock = "ballPos";
   trigger_attach(inBlock,TRIGGER_MATCHING);
}

void BallPosAgent::sense_select_act(int pid) {
  int xMean=0, yMean=0;
  yarp::sig::ImageOf<yarp::sig::PixelRgb> *image;
  image = space_read(conf.inBlock, defimage);
  if (image != NULL) {
    width = image->width();
    height = image->height();
    for (int x = 0; x < width; x++) {
      for (int y = 0; y < height; y++) {
        yarp::sig::PixelRgb& pixel;
        pixel = image->pixel(x,y);
        if (pixel.b > pixel.r*1.2+10
            && pixel.b > pixel.g*1.2+10) {
          xMean += x; yMean += y; count++;
        }
      }
    }
    if (count > 0) {
      xMean /= count; yMean /= count;
      target.x = xMean; target.y = yMean;
      space_write(outBlock, target, 300);
    }
  }
}
```

Kód 4. Agent detekujúci polohu loptičky na obraze



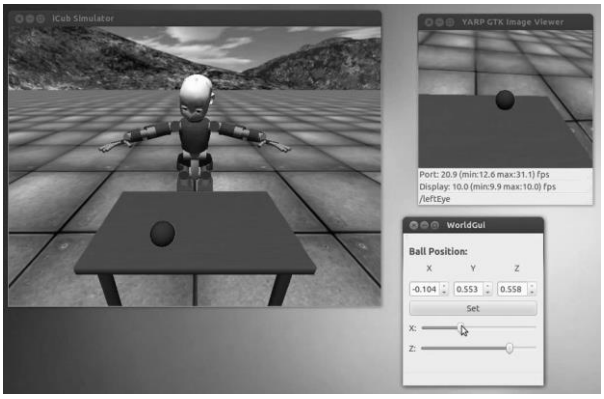
Obr. 9: Vývojová fáza 2

Druhá fáza (obr. 9) rieši situáciu, keď robot loptičku v zornom poli nemá. Vtedy začína gúľať očami, pričom miesto nejakého globálneho algoritmu prehľadávania, používa jednoduchý trik a to, že nezávisle na sebe nimi pohybuje zľava doprava (EyeVersionAgent) a zhora dole (EyeTiltAgent) rôznymi rýchlosťami a smer pohybu mení na základe dosiahnutia krajných polôh, o čom sa dozvie z propriocepcie, ktorú zabezpečuje PositionAgent. Následkom toho je pohyb očí oveľa prirodzenejší, než keby sa robot snažil organizovane prehľadať priestor okolo seba. Pritom sa netreba báť, že by sa loptička ocitla v nejakom mŕtvom priestore, ktorý robot do pohľadu nedostane, hoci by na ňu dovidel. Pokyny od agentov gúľajúcich očami (EyeVersionAgent, EyeTiltAgent) majú menšiu prioritu ako pokyny od sledovania loptičky (LookAtAgent), takže pokiaľ robot loptičku vidí, tak očami negúľa. Na druhej strane tu vidíme, že je dôležité, aby mali pokyny od LookAtAgent obmedzenú časovú platnosť, aby sa gúľanie očí vôbec niekedy prejavilo.



Obr. 10: Vývojová fáza 3

V tretej fáze sme pridali pohyb hlavy, ktorý je potrebný, keď gúľanie očami nestačí. Ten realizuje RestPosAgent, ktorý hýbe hlavou zľava doprava (obr. 10).



Obr. 11: Implementácia motivačnej úlohy

7 Záver

V príspevku sme prezentovali rozšírenie simulátora iCubSim o možnosť tvoriť jeho riadiacu aplikáciu na báze architektúry Agent-Space. Vo vyvinutom

prostriedku sme potom implementovali ako príklad sledovanie loptičky robotom, pričom užívateľ mohol touto loptičkou hýbať, aj ju schovať (obr. 11). Zdrojové kódy ako aj videá prezentujúce výsledok sú na www.agentspace.org/icub.

V budúcnosti sa budeme snažiť jednak trochu tieto kódy očistiť a zdokumentovať, jednať previazať vnímanie robota s externým prostredím, aby videl nielen predmety na svojom stole, ale aj samotného experimentátora, t.j. užívateľa programu.

Literatúra

R. A. Brooks: Cambrian Intelligence, The MIT Press, Cambridge, Mass., 1999.

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

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

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

Lúčny, A.: Multiagentový prístup k modelovaniu mysle - alebo ako sledovať pingpongovú loptičku. In: Umelá inteligencia a kognitívna veda III. (Kvasnička V. ed.), STU, Bratisva, 2011

Lúčny, A.: Otvorená implementácia architektúry Agent-Space. In: Kognice a umělý život XII. (Kvasnička, V. - Wiedermann, J. eds.), Agentura Action M, Praha, 2012, pp. 132-136

Metta, G., Natale, L., Nori, F., Sandini, G., Vernon, D., Fadiga, L., von Hofsten, C., Rosander, K., Lopes, M., Santos-Victor, J., Bernardino, A. a Montesano, L. (2010). The icub humanoid robot: An open-systems platform for research in cognitive development. Neural *Networks*. vol. 23, no. 8-9, pp. 1125–1134.

Minsky, M.: Society of Mind. Simon & Schuster, New York, 1986

Peniak, M. (2013). Aquila 2.0 Software Architecture for Cognitive Robotics. ICDL 2013