UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA APLIKOVANEJ INFORMATIKY

Ronald Weiss

Vizuálne programovanie riadiaceho systému pre kolóniu robotov

Diplomová práca

Školiteľ: RNDr. Andrej Lúčny PhD.

Bratislava 2007

Týmto prehlasujem, že som diplomovú prácu vypracoval samostatne, a všetku použitú literatúru uvádzam v zozname.

Bratislava, Apríl 2007 Ronald Weiss

Ďakujem svojmu školiteľovi RNDr. Andrejovi Lúčnemu PhD. za jeho cenné rady a pripomienky pri písaní tejto práce. Ďakujem tiež Mgr. Pavlovi Petrovičovi za rýchle riešenie technických problémov v robotickom laboratóriu, na ktoré som pri práci narazil.

Abstrakt

Ronald Weiss: *Vizuálne programovanie riadiaceho systému pre kolóniu robotov*. Diplomová práca. Univerzita Komenského v Bratislave, Fakulta Matematiky Fyziky, a Informatiky. Školiteľ: RNDr. Andrej Lúčny PhD. Bratislava 2007.

Táto diplomová práca popisuje návrh a implementáciu programovateľného riadiaceho systému pre kolóniu mobilných robotov. Systém beží na počítači, ktorý má priamy prístup k robotom, a programovať sa dá cez internet s pomocou webového rozhrania. Jedná sa o multiagentový systém, a jeho programovanie spočíva vo výbere a nastavení agentov, ktoré majú bežať. Programovanie cez webové rozhranie je vizuálne, užívateľ nemusí poznať žiadny zložitý programovací jazyk, správanie robotov si jednoducho "nakliká". Multiagentový riadiaci systém je postavený na netradičnej softvérovej architektúre Agent-Space, používajúcej výlučne nepriamu komunikáciu medzi agentmi, čo mu dáva niektoré zaujímavé vlastnosti, odlišujúce ho od iných podobných systémov.

Kľúčové slová: mobilné roboty, riadiaci systém, vizuálne programovanie, Agent-Space

Obsah

1 Úvod **7**

2 Prehľad **9**

2.1 Multiagentové systémy 9

2.2 Subsumpčná architektúra 10

2.3 Architektúra Agent-Space 12

2.4 Platforma *Java* a jej výhody 15

3 Požiadavky **16**

3.1 Popis robotického laboratória 16

3.2 Programovanie robotov 16

4 Hrubý návrh systému **18**

4.1 Riadiaci systém 18

4.2 Agenty 19

4.3 Webové rozhranie 20

5 Detailný návrh a implementácia **22**

5.1 Agent-Space 22

5.2 Riadiaci systém 25

5.3 XML reprezentácia objektov 26

5.4 Základné operácie s robotmi 27

5.5 Jednoduché správania robotov 30

5.6 Webové rozhranie 32

5.7 Bezpečnosť 34

6 Experimenty **36**

6.1 Obchádzanie pohyblivej prekážky 36

6.2 Napodobňovanie druhého robota 41

7 Záver **42**

A Návod na použitie **43**

B Stručný prehľad pripravených agentov **44**

C Robot *Robotnačka* **47**

D Simulátor **49**

Zoznam obrázkov

3.1 Robotické laboratórium 17

5.1 Činnosť agentov 24
5.2 Získavanie pozície robota 29
5.3 Pohyb robotom k cieľovému miestu 31
5.4 Editor vo webovom rozhraní 33

6.1 Agenty a bloky počas experimentu 6.1 38
6.2 Experiment 6.1 #1 40
6.3 Experiment 6.1 #2 40
6.4 Experiment 6.1 #3 40

C.1 Robotnačka 48

Kapitola 1

Úvod

Význam robotiky je v posledných rokoch na vzostupe. Roboty už nie sú žiadna fikcia, vykonávajú za ľudí čoraz viac úloh, na ktoré sami nestačíme. Či už to sú priemyselné roboty, ktoré vystriedali ľudí pri linkách vo fabrikách, alebo roboty pomáhajúce pri skúmaní vesmíru či podmorských hlbín, alebo hoci roboty nahrádzajúce chirurga na operačnom sále. Posúvajú hranice našich možností stále ďalej.

Fakulta matematiky, fyziky a informatiky UK a Fakulta elektrotechniky a informatiky STU majú spoločné robotické laboratórium, v ktorom je v spoločnom výbehu niekoľko korytnačích robotov, ktoré sa dajú ovládať z diaľky cez internet. V laboratóriu je aj kamera, takže užívateľ môže roboty sledovať v priamom prenose (viď obrázok 3.1). Toto laboratórium slúži na experimentovanie s riadením mobilných robotov.

Cieľom tejto práce je vybudovať vizuálne prostredie pre vysokoúrovňové programovanie robotov v tomto laboratóriu. Toto prostredie bude voľne prístupné cez internet. Užívateľ bude mať k dispozícii sadu pripravených jednoduchých správaní, z ktorých si môže vyskladať zložitejšie správanie, ktoré chce sledovať. Takto by sa malo experimentovanie s týmito robotmi výrazne uľahčiť.

Druhým cieľom práce je vyskúšať založiť vizuálne programovanie na netradičnej architektúre Agent-Space (viď sekciu 2.3). Už totiž existujú komerčné systémy, umožňujúce vizuálne programovanie riadiaceho systému pre roboty[1]. Jedná sa spravidla o modulárny softvér, v ktorom sa priamo pospájajú vstupy a výstupy jednotlivých modulov, nastavia sa parametre jednotlivých modulov, a tak sa nadefinuje ako sa vstupy zo senzorov pretransformujú na výstupy pre aktuátory. Toto je bežná schéma, ktorá ale iba kopíruje zapájanie hardvéru. V čisto softvérovej implementácii riadiaceho systému si však môžme dovoliť viac. Systém vybudovaný v tejto práci sa vďaka použitiu spomínanej architektúry Agent-Space dokáže od tejto zaužívanej schémy odpútať. Bude rovnako založený na skladaní pripravených modulov, ale rozdiel bude v komunikácii medzi nimi. Namiesto priameho spájania modulov použije nepriamu komunikáciu,

[1] napríklad Microsoft Robotics Studio, alebo iConnect od firmy Micro-Epsilon

modulom len nadefinujeme odkiaľ brať vstup a kam posielať výstup. Týmto sa zbavíme obmedzení, ktoré vyplývajú z priameho spájania vstupov a výstupov. To je napríklad nutnosť mať ku každému výstupu pripojený práve jeden vstup, a naopak. Nepriamou komunikáciou získame možnosť mať viac čitateľov a/alebo zapisovateľov na jedno spojenie, bez toho aby o tom komunikujúce moduly museli vedieť. Ďalšie obmedzenie, ktorého sa zbavíme, je nutnosť nejakej vzájomnej synchronizácie behu jednotlivých modulov. V našom systéme budú jednotlivé moduly bežať úplne nezávisle na ostatných.

Spomínaná tradičná schéma priameho spájania modulov je vhodná na tvorbu jednoduchých systémov, typu *"pipeline*", prípadne s viacerými vetvami. To je ideálne pre programovanie priemyselných robotov s jednoduchým priamočiarym správaním. Schéma s nepriamou komunikáciou je naproti tomu vhodná na tvorbu komplikovaných zapojení, kde moduly bežia paralelne, pričom môžu navzájom spolupracovať, potláčať sa, alebo si konkurovať. V takomto systéme sa mnohokrát vynorí aj užitočné správanie, ktoré vývojár pôvodne ani nezamýšľal (tento jav sa volá *emergencia* [9]).

Schéma s priamym spájaním modulov je vyslovene vhodná na vizuálne komponentové programovanie. Zdá sa, že schému s nepriamou komunikáciou doteraz nikto na tento účel nepoužil [10]. Táto práca si teda dáva za cieľ ukázať, že táto schéma je na vizuálne programovanie rovnako vhodná, a pritom ponúka jasné výhody oproti tradičnému riešeniu.

Kapitola 2

Prehľad

V tejto kapitole popíšem existujúce technológie, na ktorých som postavil systém vybudovaný v tejto práci. Základom je softvérová architektúra Agent-Space, na ktorej je systém založený, a z ktorej vlastne čerpá svoje špecifické vlastnosti. Kým sa však k tejto architektúre dostanem, najskôr treba spomenúť čo sú multiagentové systémy, pre stavbu ktorých je určená, a tiež ešte predstavím predchodcu Agent-Space, subsumpčnú architektúru. V tomto prehľade čerpám najmä z [9] a [8].

Môj systém používa platformu Java, a v tejto kapitole ešte stručne popíšem jej špecifické vlastnosti pre ktoré sme ju zvolili.

2.1 Multiagentové systémy

Jednoduchá definícia multiagentového systému je, že je to decentralizovaný systém zložený z množiny nezávislých agentov, ktorý vďaka kooperácii týchto agentov dosahuje nejaký výsledok, ktorý by bolo ťažšie dosiahnuť jednoliatym (resp. centrálne riadeným) systémom.

Samozrejme vyvstáva otázka, čo je to agent. Definície agenta v rôznej literatúre sa dosť líšia, ale pre potreby tejto práce budeme pod agentom rozumieť autonómnu stavebnú jednotku softvéru, teda nejaký kód, ktorý pracuje samostatne, nepotrebuje pre svoj chod iných agentov, ale môže (a väčšinou musí ak má byť užitočný) s nimi komunikovať.

Agenty môžme rozdeliť podľa zložitosti kódu, ktorý vykonávajú na silné a slabé [7]. Silné agenty obsahujú nejakú inteligentnú súčiastku, kým slabé agenty obsahujú iba nejaký jednoduchý kód. Systém poskladaný len zo slabých agentov čerpá inteligenciu z vhodného prepojenia a komunikácie medzi agentmi. Pre potreby tejto práce sú zaujímavejšie práve slabé agenty.

Pre agenta je typický kód typu *sense-select-act*. To znamená že pracuje v nekonečnom cykle, a v každej iterácii najskôr načíta vstupy z prostredia, potom vyberie nasledujúcu akciu, a vykoná ju. Podľa výberu akcie sa agenty delia na deliberatívne a reaktívne. Deliberatívne agenty majú

svoj cieľ (explicitne popísaný v nejakom reprezentačnom jazyku), a na základe tohto cieľa a vstupov z prostredia sa rozhodujú, čo ďalej. Reaktívne agenty, na druhej strane, jednoducho reagujú na vstup, nemajú žiaden explicitný cieľ (majú však implicitný cieľ - dá sa povedať čo robia). Toto delenie sa výrazne kryje s delením na slabé a silné agenty. Reaktívne agenty sú spravidla slabé, a deliberatívne sú silné.

Agenty môžeme chápať ako spôsob, akým sa dá nejaká entita reálneho sveta reprezentovať v počítači. V princípe takéto spôsoby existujú tri: entity pasívne, reaktívne, a proaktívne. Pasívna entita sú nejaké dáta, s ktorými môžme ľubovoľne manipulovať, ale nemôžme od nej nič očakávať. V bežných programovacích jazykoch sa označuje ako záznam (*record*) alebo štruktúra (*struct*). Reaktívna entita, ktorej sa hovorí objekt, so sebou nesie okrem dát aj vykonateľný kód, ktorého vykonanie však treba zvonku aktivovať. Agent je proaktívna entita, ktorá vykonáva svoj kód samostatne, bez potreby externého podnetu. Tak ako programovaniu s objektmi sa hovorí objektovo-orientované programovanie, tak programovanie s agentmi označujeme ako agentovo-orientované programovanie [13].

Multiagentové systémy môžeme ďalej deliť podľa implementácie komunikácie medzi agentmi. Rôzne agenty môžu byť na rôznych počítačoch, a komunikovať cez sieť. Alebo môžu byť na jednom počítači. Buď ako samostatné procesy, a na komunikáciu používať nejakú formu IPC, alebo ako vlákna v rámci jedného procesu, a komunikovať priamo cez zdieľanú pamäť.

Multiagentový systém, ktorý vybuduje táto práca, bude zložený výlučne z reaktívnych agentov. Bude implementovaný v jazyku Java, celý v rámci jediného virtuálneho stroja, teda v jedinom procese. Každý agent bude mať svoje vlákno (*thread*), a komunikovať budú cez zdieľaný pasívny objekt *Space* (viac v sekcii 2.3).

2.2 Subsumpčná architektúra

Subsumpčnú architektúru táto práca priamo nepoužije, ale je natoľko dôležitá v tejto oblasti, že o nej treba napísať. Architektúra Agent-Space, ktorú použijeme, je ňou vlastne tiež inšpirovaná, a preberá jej hlavné myšlienky.

Subsumpčná architektúra je šitá špeciálne na budovanie systémov na ovládanie mobilných robotov. Jej autorom je R. Brooks [2, 4], ktorý ju vymyslel ako reakciu na neúspechy metód tradičnej umelej inteligencie v tejto oblasti.

Hlavný rozdiel medzi subsumpčnou architektúrou a tradičnými prístupmi je v type dekompozície systému na moduly. Kým štandardný postup je rozdeliť systém podľa funkcie, teda do jedného modulu dať kód vykonávajúci podobné funkcie, v subsumpčnej architektúre sa systém delí podľa aktivity. To znamená, že v jednom module je kód potrebný pre funkčnosť nejakej časti správania systému.

Ak systém rozdelíme tradičným spôsobom, jeden z modulov (tzv. kognitívny modul) bude zodpovedný za inferenciu na poznatkoch. Na všetkých poznatkoch, zo všetkých častí systému. To je práve problém, pretože pri rozsiahlejších systémoch narazíme na bariéry výpočtovej zložitosti. Vstupných dát bude veľa, a kognitívny modul bude zbytočne brať do úvahy stále všetko. Je problém spoľahlivo vybrať len tie vstupy ktoré sú dôležité pre danú aktivitu. Rovnako môže byť ťažké podľa výstupu z kognitívneho modulu rozhodnúť o ďalších akciách.

Subsumpčná architektúra, na rozdiel od tradičných metód, nemá samostatne vyčlenený kognitívny modul, a každý modul, venujúci sa svojej jednoduchej aktivite, si vykonáva svoju vlastnú kogníciu, ak treba. Na malej množine vstupov, s malým množstvom výsledných akcií, v reálnom čase.

Brooks sformuloval tieto pojmy [3], na ktorých svoju subsumpčnú architektúru založil:

Situovanosť Pri tvorbe systému počítame so špecifickými podmienkami prostredia, v ktorom bude fungovať. Nepokúšame sa budovať abstraktnú reprezentáciu sveta.

Stelesnenosť Robot má telo, ktoré je v priamej interakcii s prostredím. Akcie majú priamy vplyv na percepciu. Systém preto vyvíjame priamo s prototypom v ruke, a hneď ho testujeme v ostrých podmienkach.

Interakcia Využívame dynamiku prostredia. Interakcia s prostredím sa prejavuje ako inteligentné správanie.

Emergencia Správanie systému vyplýva z interakcie modulov. Je možné implicitné vynorenie správania, ktoré vývojár ani nezamýšľal.

Systémy na báze subsumpčnej architektúry sú vyvíjané inkrementálne, po vrstvách. Postupuje sa pritom zdola nahor, pričom vyššie vrstvy môžu využívať (monitorovať aj ovplyvňovať) nižšie. Jednotlivé vrstvy zodpovedajú jednotlivým aktivitám, a zároveň fázam vývoja. Vrstvy sa skladajú z modulov. Moduly sú konečné automaty, rozšírené o pamäť (vstupné, výstupné a vnútorné registre) a časovač. Na impulzy od časovača sa spustí výpočet, a vstup sa pretransformuje na výstup. Moduly (ich vstupy a výstupy) sú pospájané vedeniami. Vyššie vrstvy môžu zasahovať do nižších tak, že sa pripoja na vedenie v nižšej vrstve, a môžu buď odchytávať prechádzajúce dáta, alebo ich dokonca meniť. Dá sa povedať, že vyššie vrstvy "zahrňujú" tie nižšie, z čoho pochádza názov subsumpčnej architektúry.

Existujú tri mechanizmy subsumpcie:

Duplikácia Pasívne odpočúvanie vedenia.

Supresia Nahradenie pôvodných dát inými, na nejaký čas.

Inhibícia Zastavenie šírenia dát, na nejaký čas.

Prvý robot, implementovaný subsumpčnou architektúrou, bol robot Allen, ktorý prehľadáva kancelárske priestory na jednom poschodí. Je to veľmi jednoduchý okrúhly robot, so sonarmi po obvode, ktoré mu umožňujú detekovať prekážky vo všetkých smeroch, a s dvomi nezávisle poháňanými kolesami, vďaka ktorým sa môže pohybovať rovno, po ľubovoľnej kružnici, či točiť sa namieste (rovnako ako roboty, ktorým sa venuje táto práca). Implementácia v subsumpčnej architektúre je veľmi elegantná a jednoduchá, a robot skutočne ako-tak prehľadáva oblasť dokonca bez toho, aby si musel budovať nejakú mapu. Systém sa skladá z troch vrstiev. Prvá má na starosti vyhýbanie sa prekážkam. Po jej implementovaní sa Allen pohybuje vpred, a ak sa priblíži k nejakej prekážke, otáča sa smerom od nej. Takže sa stále hýbe, ale rýchlo sa zacyklí. Druhá vrstva pridá bezcieľne náhodné potulovanie. Na náhodné zatáčanie pritom využíva prvú vrstvu, a to tak, že občas použije supresiu, aby nafingovala nejakú prekážku. S touto druhou vrstvou sa už robot nezacyklí, ale pravdepodobne bude stále kľučkovať na malom priestore. Napríklad nikdy neprejde cez dvere. To rieši tretia vrstva, ktorá pomocou sonarov hľadá smery, ktorými sa dá ísť čo najďalej rovno bez prekážky. Potom využije druhú vrstvu, znova za pomoci supresie, aby robot akože "náhodne" išiel práve týmto nájdeným smerom. S týmito troma vrstvami už robot časom prehľadá celý priestor, napriek tomu že si ani nebuduje mapu (hoci budovaním mapy sa dá prehľadávanie samozrejme zefektívniť).

2.3 Architektúra Agent-Space

Architektúra Agent-Space (jej autorom je A. Lúčny [9]) je silne inšpirovaná subsumpčnou architektúrou. Prenáša jej myšlienky z poľa mobilnej robotiky do poľa všeobecného programovania systémov reálneho času. S touto architektúrou sa vraciame k agentovej terminológii, ktorú subsumpčná architektúra priamo nepoužíva, hoci jej moduly by sme tiež bez rozpakov mohli nazvať agentmi.

Systém na báze architektúry Agent-Space sa skladá z množiny reaktívnych agentov, ktoré spolu komunikujú výlučne nepriamo, pomocou štruktúry "*Space*" (po slovensky prostredie).

Space je objekt, ktorý obsahuje pomenované dáta, tzv. bloky, ktoré môžu agenty čítať aj zapisovať (stačí im poznať meno bloku). Blok fyzicky vznikne pri prvom zápise, čítať sa však dá aj predtým (vráti napríklad nejakú špeciálnu hodnotu). Toto je dôležitá vlastnosť, ktorá predchádza synchronizačným problémom. Bloky majú svoju dobu platnosti (môže byť nekonečno), po ubehnutí ktorej sa z prostredia odstránia. Túto dobu platnosti nastavujú agenty pri zapisovaní. Forma dát ukladaných v blokoch nie je nijako daná, space ich iba ukladá, a nepotrebuje im rozumieť. Je na návrhárovi systému, aby si zapisujúce a čítajúce agenty rozumeli.

Na space môžme nazerať ako na server, ktorý agentom poskytuje služby. Základné služby sú samozrejme čítanie a zapisovanie blokov. Existujú však aj ďalšie štandardné služby prostredia. Napríklad mazanie blokov. To, rovnako ako zápis, môže mať ako parameter prioritu, ktorá

dovoľuje uprednostňovanie niektorých agentov pred inými. Ďalšou možnou službou prostredia je registrácia *triggrov*, ktorými sa agenty môžu nechať upozorniť na zmeny v prostredí (zápisy a mazanie blokov). Keď sa zmení nejaký blok v prostredí, upozornia sa všetky agenty, ktoré si zaregistrovali trigger na tento blok. Tiež často používanou službou prostredia sú masové operácie s blokmi na základe nejakej masky.

Space je štandardný objekt v tejto architektúre, ktorý majú všetky systémy v podstate rovnaký. Nevykonáva žiadny iný kód, okrem práve popísaných služieb. Aplikačný kód realizujú reaktívne agenty (viď sekciu 2.1). Teda malé jednoduché autonómne programy, vykonávajúce nekonečný cyklus *sense-select-act*. Agent sa zobudí zo spánku na pravidelne generovaný impulz od časovača, alebo na trigger z prostredia, prečíta potrebné dáta z prostredia (prípadne senzorické vstupy), vykoná jednoduchý výpočet (žiadna inteligentná súčiastka), výsledky zapíše do prostredia (alebo pošle aktuátorom), a znova zaspí. Keďže agenty sú reaktívne, inteligencia systému je skrytá vo vhodnej interakcii medzi nimi.

Interakcia so senzormi a aktuátormi (v robotickej terminológii, ale môžme pod tým rozumieť aj hardvérové zariadenia či užívateľa) prebieha v architektúre Agent-Space tak, že každé zariadenie (senzor čí aktuátor) má asociovaného agenta, ktorého jediná práca je čítať hodnotu zo senzoru a zapisovať ju do nejakého bloku, alebo čítať hodnotu z nejakého bloku, a posielať ju na aktuátor. Takto sú zariadenia prístupné všetkým agentom, a všetky agenty môžu monitorovať vstup aj výstup.

Ak agent nemá žiadny vnútorný stav (t.j. premenné, ktoré nie sú lokálne vzhľadom na jednu iteráciu sense-select-act), hovoríme že je rýdzo (alebo čisto) reaktívny. Rýdza reaktivita agentov má svoje výhody. Napríklad tú, že nič nie je skryté pred inými agentmi. Agenty teda môžu ľahko monitorovať, ale aj ovplyvňovať činnosť iných agentov. Rýdzo reaktívneho agenta tiež môžme bez obáv kedykoľvek reštartnúť, a on bude normálne pokračovať vo svojej práci, akoby sa nič nestalo.

Systémy implementované architektúrou Agent-Space majú tieto zaujímavé vlastnosti:

- *Nemožnosť deadlockov* - Space je server, agenty sú klienty, nikto nie je server i klient zároveň, takže deadlock sa nedá dosiahnuť
- *Zotaviteľnosť z chýb* - Ak sa nejaký agent vinou chyby zrúti, môžme ho jednoducho reštartovať (napríklad nejakým ďalším agentom, ktorého úloha je kontrolovať bežiace agenty, a reštartovať zrútené).
- *Konfigurovateľnosť* - Konfiguráciu systému možno za jazdy meniť vďaka možnosti reštartnúť jednotlivé agenty. Čistá reaktivita je tu opäť veľmi užitočná. Jednak uľahčuje reštarty, ale zároveň dovoľuje zmeniť konfiguráciu dokonca bez reštartov (celý stav je v prostredí).

- *Jednoduchý štart systému* - Vďaka možnosti čítať blok predtým ako bol zapísaný, sa netreba starať o poradie, v akom agenty naštartujeme.
- *Dátový tok many:many* - A to úplne transparentne. Čitateľov bloku vôbec nemusí zaujímať, koľko zapisovateľov doňho píše, a rovnako naopak, zapisovateľovi je jedno, koľko čitateľov blok číta.
- *Jednoduché zálohovanie* - Informácia môže prichádzať z viacerých zdrojov, je to umožnené dátovým tokom many:many.
- *Implicitné vzorkovanie* - Automatické riešenie problému rýchleho zapisovateľa a pomalého čitateľa, niektoré dáta sa jednoducho stratia.
- *Decentralizácia* - Žiadny agent nemá výhradné postavenie. Všetky sú autonómne.
- *Modifikovateľnosť systému* - Systémy možno jednoducho modifikovať pridávaním nových agentov a blokov, bez dopadu na funkčnosť pôvodných agentov.

Samozrejme, ako všetko, aj Agent-Space má nielen výhody, ale nájdu sa aj nevýhody:

- *Hrozí nekonzistentnosť dát* - Vzniká pomalým propagovaním informácií. Často to však nevadí, a spravidla s tým netreba počítať. Napríklad keď sa hodnoty v blokoch menia spojito.
- *Hrozí strata dát* - Hodnota v bloku môže byť prepísaná skôr ako ju niekto prečíta. Táto zdanlivá nevýhoda je však často naopak výhodou, spôsobuje totiž implicitné vzorkovanie.
- *Zbytočná práca* - Opak implicitného vzorkovania, agent môže z blokov opakovane čítať rovnaké hodnoty, a spracovávať ich.
- *Plytvanie výkonom* - Optimálne využitie výkonu nie je v Agent-Space na prvom mieste. Je to cena za eleganciu a jednoduchosť, ktorými sa komplexné riešenia v Agent-Space vyznačujú.

Agent-Space vychádza zo subsumpčnej architektúry, a je vlastne jej vylepšením. Dokáže ju priamočiaro simulovať. Moduly zo subsumpčnej architektúry simuluje agentmi, a komunikačné spojenia blokmi. Duplikáciu simuluje jednoducho čítaním jedného bloku viacerými agentmi, supresiu a inhibíciu prepísaním resp. vymazaním bloku s vyššou prioritou. Jediná nepríjemná situácia nastáva, ak je duplikácia a supresia (alebo inhibícia) na tom istom vedení. To je pre Agent-Space problém, dá sa však obísť zdvojením bloku, a pridaním kopírovacieho agenta.

2.4 Platforma *Java* a jej výhody

Názov Java označuje technológiu na tvorbu multiplatformných aplikácií od firmy Sun Microsystems. Základom je moderný objektovo-orientovaný programovací jazyk, rovnako nazývaný Java. Aplikácie napísané v Jave sa nekompilujú priamo do strojového kódu pre konkrétny procesor, ale do prechodnej formy (takzvaný *bytecode*), ktorú potom interpretuje virtuálny stroj (JVM - *Java Virtual Machine*). Implementácie JVM existujú pre všetky majoritné platformy (Windows, Linux, Mac), takže aplikácie napísané v Jave môžeme používať takmer všade. Presne o tom hovorí motto Java technológie "*write once, run anywhere*".

Okrem svojej hlavnej výhody, multiplatformnosti, ktorá ale pre implementáciu riadiaceho systému pre naše robotické laboratórium až taká dôležitá nie je, má Java ešte množstvo ďalších pozitív. Sú to hlavne:

- *reflekčný model* - dynamický prístup k dátam a metódam objektov podľa mena, počas behu programu
- *security framework* - bezpečnostný model umožňujúci rôznym častiam kódu prideliť rôzne oprávnenia
- prirodzená podpora vlákien (*thread*), a medzivláknovej synchronizácie
- *garbage collection* - model správy pamäte, v ktorom sa už nereferencované objekty automaticky uvoľňujú
- štandardná knižnica priamo podporuje mnoho pokročilých funkcií (napr. sieťová komunikácia, ...)

Kvôli všetkým týmto výhodám bola Java jasnou voľbou pre náš riadiaci systém. Keďže systém je programovateľný vzdialene cez web, obzvlášť sa nám hodí reflekčný model, ktorý umožňuje jednoduchú serializáciu a následnú rekonštrukciu objektov. Rovnako veľmi dôležitý je pre nás bezpečnostný model. Umožňuje dovoliť záujemcom o používanie nášho laboratória doprogramovať si do nášho systému vlastné funkcie, a pritom sa nevystaviť bezpečnostnému riziku.

Java sa používa tiež na programovanie webových aplikácií, takzvaných *applet*ov. V tomto prípade sa program (applet) stiahne zo servera do prehliadača klienta, a tam sa spustí. To sa nám tiež hodí do tejto práce. Takže v Jave je napísaný nielen samotný riadiaci systém, ale aj užívateľské webové rozhranie k nemu, ktoré s ním komunikuje.

Viac o Jave sa dá nájsť v hociktorej z množstva kníh o nej, napríklad v [6].

Kapitola 3

Požiadavky

Kým začnem popisovať systém vytvorený touto prácou, treba najprv špecifikovať čo už bolo v našom robotickom laboratóriu urobené, a čo k tomu má pridať táto práca. Najskôr stručne popíšem, ako samotné laboratórium vyzerá, a potom ako sa s robotmi v ňom dalo pracovať doteraz, a čo bude nové.

3.1 Popis robotického laboratória

Robotické laboratórium pozostáva zo stola, na ktorom sa pohybujú roboty, kamery upevnenej na konštrukcii nad stolom, a riadiaceho počítača (viď obrázok 3.1). Riadiaci počítač komunikuje s robotmi bezdrôtovo, s pomocou technológie Bluetooth. Je pripojený k internetu, a prostredníctvom webového serveru umožňuje roboty ovládať komukoľvek cez internet.

Na stole v laboratóriu sú väčšinou dva až štyri roboty typu *Robotnačka* (viď dodatok *C*). Sú to okrúhle roboty, s dvoma nezávisle poháňanými kolesami, a tretím podporným. Vedia sa pohybovať priamo, alebo po akejkoľvek kružnici. Roboty môžu mať aj pero, ktorým vedia kresliť na stôl, kade chodia. Jeden z robotov má navyše chápadlo, ktorým dokáže zdvihnúť a prenášať malé predmety. Všetky roboty majú ešte šesť senzorov po obvode, ktorými vedia detekovať čiary na stole.

Viac o robotickom laboratóriu sa dá dočítať v [12].

3.2 Programovanie robotov

K robotom v laboratóriu je naprogramovaná knižnica, ktorá umožňuje ich jednoduché nízkoúrovňové ovládanie. To zahŕňa "korytnačie" pohyby (vpred, vzad, otáčanie), ovládanie príslušenstva (pero hore/dole, chápadlo zovrieť/uvoľniť a zdvihnúť/dať dole), a zisťovanie stavu (stav batérie, údaje zo senzorov). Nad touto knižnicou je naprogramované webové rozhranie,

Obrázok 3.1: Robotické laboratórium

ktoré umožňuje roboty ovládať cez internet. Toto rozhranie je len "tenká vrstva" nad knižnicou. Má tie isté funkcie, len ich sprístupňuje v grafickom prostredí.

Táto práca má vybudovať systém pre vysokoúrovňové "behaviorálne" programovanie robotov v laboratóriu. To bude spočívať v skladaní zložitejšieho správania robotov z pripravených hotových modulov, realizujúcich jednoduché činnosti. Súčasťou systému bude ďalšie grafické webové rozhranie, takže aj toto vysokoúrovňové programovanie robotov bude prístupné pre každého cez internet. Užívateľ bude môcť prepínať medzi starým rozhraním, s ktorým môže roboty manuálne ovládať, a novým, kde bude môcť pre roboty naprogramovať zložitejšie správanie.

Kapitola 4

Hrubý návrh systému

Systém vybudovaný v tejto práci som nazval RLCS, čo je skratka z anglického názvu *Robotic Lab Control System*. Vo zvyšku práce práce budem používať toto meno.

Systém RLCS sa skladá z dvoch hlavných častí:

1. samotný riadiaci systém, ktorí beží na serveri, a má priamy prístup k robotom
2. webové rozhranie na programovanie riadiaceho systému

V tejto kapitole popíšem hlavné myšlienky stavby riadiaceho systému, ako aj používateľského rozhrania. Detailnejší popis nasleduje v ďalšej kapitole.

4.1 Riadiaci systém

Riadiaci systém RLCS je multiagentový systém, čiže sa skladá z agentov. Tieto agenty zabezpečujú všetku funkčnosť systému, takže ak nebežia žiadne agenty, systém nerobí nič. Užívateľ má nad systémom takmer úplnú kontrolu, čo znamená že takmer všetky agenty v systéme sú spustené užívateľom. Píšem takmer, lebo predsa len bežia v systéme 2 základné agenty ktoré zabezpečujú jeho beh, a na ktoré užívateľ nemá vplyv: prvý je špeciálny agent *Scheduler* ktorý má na starosti spúšťanie a kontrolu ostatných agentov, a druhý je agent komunikujúci s používateľským rozhraním.

Systém RLCS beží na serveri stále, a čaká na spojenia od klientov (webového rozhrania). V jednom spojení klient pošle konfiguráciu, čiže sadu agentov a ich parametrov, na čo RLCS zareaguje stopnutím všetkých bežiacich agentov, a naštartovaním nových, podľa konfigurácie od klienta. Táto operácia je atomická, takže ak je klientov viac, nepomiešajú sa. Ale, samozrejme, ak je klientov viac, tak zostane v platnosti iba konfigurácia od toho z nich, ktorý sa pripojil ako posledný.

Prijímanie konfigurácie od rozhrania je štandardný spôsob fungovania RLCS, avšak je možné úvodnú konfiguráciu načítať aj zo súboru, takže v prípade potreby je možné RLCS používať aj bez grafického rozhrania.

Ak som napísal, že riadiaci systém má priamy systém k robotom, neznamená to nutne, že musí bežať na počítači priamo v robotickom laboratóriu. K robotom sa totiž dá pripojiť aj vzdialene cez sieť, prostredníctvom počítača v laboratóriu. Je však vhodné, aby spojenie medzi riadiacim systémom a robotmi (a tiež kamerou, ku ktorej sa tiež dá pripojiť cez sieť) bolo rýchle.

Samotný riadiaci systém je teda pomerne jednoduchý, a správanie robotov závisí výlučne na užívateľom spustených agentoch.

4.2 Agenty

K systému RLCS je samozrejme naimplementovaná aj sada agentov, aby sa s ním vôbec niečo dalo robiť. Vyskladaním z týchto agentov sa dajú s robotmi v našom laboratóriu dosiahnuť niektoré celkom zaujímavé pohybové správania.

Pri vývoji systému som sa držal postupu, ktorý zaviedol R. Brooks pre svoju subsumpčnú architektúru (viď sekciu 2.2). Vyvíjal som ho po vrstvách, zdola nahor. Ako uvidíme, pre architektúru Agent-Space je tento postup rovnako vhodný. Na začiatok som teda pripravil agenty starajúce sa o činnosti, ktoré sú pre každú prácu s laboratóriom nevyhnutné:

- spojenie s robotom (umožňuje posielať robotovi príkazy)
- pohyb robotom
- získavanie obrazu z hornej kamery
- vysielanie obrazu (do webového rozhrania)

Aby sme mohli dosiahnuť nejaké zaujímavejšie správanie, potrebujeme na obrázku z kamery identifikovať jednotlivé roboty, aby sme poznali ich pozície. Toto je nevyhnutné, ak si máme vystačiť s hornou kamerou ako jediným senzorom. Na to slúži ďalšia sada agentov, ktoré sa starajú o:

- detekciu robotov na obrázku z kamery
- priradenie zdetekovaných pozícií jednotlivým robotom
- častejšie aktualizovanie pozície pomocou odometrie

Keď máme pozíciu robotov, sme schopní na základe nej implementovať ďalších agentov:

- pohyb ku cieľovému bodu
- detekcia hroziacej kolízie
- obchádzanie prekážok

Nad týmto už môžeme konečne implementovať agentov realizujúcich nejaké zaujímavé správania:

- náhodné potulovanie
- jazda po preddefinovanej ceste
- prenasledovanie iného robota
- napodobňovanie pohybov iného robota

Prvé tri z uvedených správaní zahŕňajú aj obchádzanie prekážok (ak sú spustené príslušné agenty).

Pripravených je aj niekoľko agentov na spracovanie obrazu, či už rôzne transformácie obrazu, alebo dokreslovanie značiek do obrazu. Do webového rozhrania sa potom dá vysielať aj spracovaný obraz, namiesto priamo obrazu z kamery.

Ďalej je pripravených ešte niekoľko všeobecných agentov na prácu s prostredím (kopírovač blokov), alebo na prácu s operačným systémom (spúšťač externých aplikácií).

4.3 Webové rozhranie

Grafické webové rozhranie slúži na programovanie systému RLCS. Užívateľ si v ňom "nakliká" akých agentov treba na serveri spustiť, a ponastavuje im parametre. Keď dá užívateľ príkaz na odoslanie konfigurácie, applet sa pripojí k serveru, a pošle systému novú konfiguráciu, ktorá ihneď vstúpi do platnosti.

Parametre agentov sú najmä mená vstupných a výstupných blokov. Tak sa nastavuje komunikácia medzi agentmi. Môžu to ale byť v princípe ľubovoľné parametre, ktoré nadefinuje vývojár ktorý vytvoril daného agenta.

Okrem pridávania agentov po jednom, sa vo webovom rozhraní dajú použiť aj takzvané *správania*. Sú to preddefinované množiny agentov, s nastaviteľnými parametrami, ktoré sa automaticky propagujú do parametrov jednotlivých agentov. Je to nejaká obdoba funkcií (procedúr, rutín, ...) v bežnom programovaní.

Alternatívne, celá konfigurácia sa dá priamo editovať ako text. Používa sa vlastný formát, založený na XML. Tento XML text je vlastne serializácia konfiguračného objektu, ktorý obsahuje zoznam agentov, spolu s ich konfiguráciami. Serializáciou myslím transformáciu objektu do postupnosti bajtov, z ktorej je možné daný objekt spätne zrekonštruovať. Je to priamo to, čo sa pošle na server, kde sa z toho zrekonštruuje konfiguračný objekt, podľa ktorého sa pospúšťajú agenty.

Webové rozhranie zároveň zobrazuje popisy všetkých agentov, parametrov, a správaní, takže slúži tiež ako manuál k systému.

Kapitola 5

Detailný návrh a implementácia

5.1 Agent-Space

Základ celého systému je implementácia architektúry Agent-Space. V jazyku Java sa to dá pomerne jednoducho a priamočiaro. Agentov ľahko implementujeme vďaka vláknam (*thread*), ktoré Java prirodzene podporuje, a jednoduchý space sa dá tiež ľahko implementovať ako mapa. My však máme predsa len niektoré špecifické požiadavky, takže sa nemôžme uspokojiť s takouto najjednoduchšou implementáciou.

5.1.1 Space

Prostredie implementujeme ako mapu blokov, s kľúčmi typu String. Objekty ukladané v prostredí zaobaľujeme do objektov triedy *Block*, ktoré okrem samotného uloženého objektu obsahujú ešte ďalšie potrebné informácie, ako sú čas zápisu, doba platnosti, priorita zápisu, a zoznam agentov, ktoré si zaregistrovali *trigger* na tento blok.

Základné služby poskytované prostredím sú *read* a *write*, tj. čítanie a zápis. Čítanie bloku normálne vráti priamo uložený objekt, ale máme aj špeciálnu metódu, ktorá vráti celý zaobaľujúci blok, z ktorého sa dajú zistiť aj meta-informácie, ako napríklad čas zápisu bloku. Čítanie bloku, v ktorom nie je nič uložené vráti *null*. To znamená že môžme čítať bloky aj predtým ako do nich niekto niečo uloží. Pri zápise sa udáva doba platnosti a priorita. Keď uplynie doba platnosti, blok sa z prostredia odstráni. Zápisy do blokov, v ktorých už je niečo s vyššou prioritou, sú ignorované.

Ďalšou službou prostredia je odstraňovanie blokov, ktoré je implementované jednoducho ako zápis hodnoty *null*, čiže preň normálne funguje priorita a doba platnosti, rovnako ako pre bežný zápis.

Agenty si môžu v prostredí zaregistrovať *trigger* na konkrétne bloky. To znamená, že budú dostávať notifikáciu o každej zmene v danom bloku. Ako táto notifikácia prebieha, popíšem

v ďalšej časti, kde budem písať o implementácii agentov. Pre každý blok si prostredie pamätá zoznam agentov, ktoré si naňho trigger zaregistrovali, a pri každej zmene tohto bloku jednoducho notifikuje všetky agenty z tohto zoznamu.

Objekty typu *Block* sa do mapy vkladajú pri operáciách *write*, *delete*, *attach-trigger*, ale iba ak daný blok v prostredí zatiaľ neexistuje. Ak už existuje, iba sa aktualizuje už existujúci objekt. Bloky sa fyzicky odstraňujú z prostredia, ak im vyprší platnosť, a nie sú na ne zaregistrované žiadne triggre.

Aby bolo možné notifikovať agenty pri vypršaní platnosti bloku, prostredie musí mať vlastné vlákno, nie je to teda celkom pasívny objekt.

Všetky metódy triedy *Space* sú synchronizované, čiže všetky operácie sú atomické. Paralelne bežiace agenty preto môžu bez obáv služby prostredia využívať.

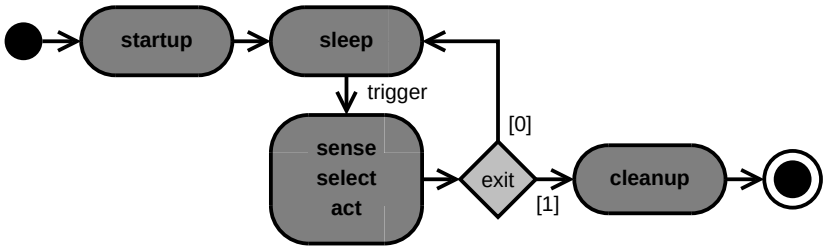
Veľmi dôležitá vlastnosť mojej implementácie prostredia je, že sa v ňom ukladajú iba referencie na objekty, a nie ich kópie. To má niekoľko výhod. Najzjavnejšou výhodou je úspora na zbytočnom kopírovaní, čo je v napríklad v prípade bitmapových obrázkov dosť znateľný rozdiel. Ďalší plus je, že môžme v prostredí ukladať akékoľvek objekty, aj také z ktorých sa nedá vytvoriť plnohodnotná kópia (nie sú klonovateľné). Agenty teda môžu cez prostredie medzi sebou zdieľať objekty. Toto so sebou ale samozrejme nesie riziko konfliktu, v prípade súčasného prístupu viacerých agentov k jednému objektu. V prostredí však ukladáme hlavne nemeniteľné (*immutable*) objekty, pri ktorých sa niet čoho báť. A v tých niekoľkých prípadoch, keď v prostredí ukladáme meniteľné objekty, ich spravidla vyslovene potrebujeme zdieľať, takže si na konflikty prirodzene dávame pozor. Ukladanie kópií namiesto referencií by bolo výhodné iba v prípade meniteľných malých klonovateľných objektov, ktoré nechceme zdieľať, a to je veľmi úzky výber zo všetkých prípadov. A ak taký prípad predsa nastane, môžme si kópiu daného objektu explicitne vyrobiť sami.

5.1.2 Agent

Agent je najdôležitejšia stavebná jednotka celého systému. Všetky agenty sú odvodené zo základnej triedy *Agent*, ktorá implementuje cyklus *sense-select-act*, a tiež niekoľko ďalších základných funkcií.

Keď sa agent naštartuje (metódou *start*), spustí sa nové vlákno (*thread*), v ktorom sa spí, až kým sa z iného vlákna nezavolá metóda *trigger*. Potom sa zavolá abstraktná metóda *senseSelectAct*, ktorú musí každá odvodená agentová trieda prekryť (*override*), a znova sa spí, až po ďalší trigger, a tak ďalej. Činnosť agenta je teda definovaná implementáciou metódy *senseSelectAct*, a tým ako často a za akých okolností sa prebúdza (po externom podnete volaním metódy *trigger*). Prebúdzanie agenta je normálne realizované dvoma spôsobmi. Úplne štandardný spôsob je, že si zaregistruje časovač, ktorý ho potom budí v pravidelných intervaloch. Druhý spôsob je

trigger z prostredia. Agent si v prostredí zaregistruje trigger na nejaké konkrétne bloky, a potom po každej zmene v niektorom z týchto blokov, prostredie agenta zobudí. Trieda *Agent* má metódy pre jednoduchú registráciu ako časovača, tak aj triggrov v prostredí.



Obrázok 5.1: Činnosť agentov

Takže najjednoduchší spôsob ako implementovať nového agenta, je odvodiť novú triedu zo základnej triedy *Agent*, prekryť metódu *senseSelectAct*, a v konštruktore zaregistrovať časovač a potrebné triggre.

Niekedy potrebujeme vedieť, či nás zobudil časovač alebo trigger z prostredia, a prípadne ktorého bloku sa trigger týka. Na toto máme implementovaný jednoduchý mechanizmus. Trieda *Agent* má dva druhy metódy *senseSelectAct*: jednu bez parametrov, a druhú z jedným parametrom typu *long*. Odvodená trieda musí prekryť práva jednu z týchto metód. Ak ju nezaujíma, čo vyvolalo prebudenie agenta, stačí pre jednoduchosť prekryť metódu bez parametrov. Ak chce vedieť čo agenta zobudilo, prekryje metódu s parametrom, z ktorého sa to potom môže dozvedieť. Pri registrácii časovača a triggrov sa zadáva číselný kód, ktorý sa použije ako parameter pre metódu *senseSelectAct*, keď situácia nastane. Nastať môže aj viac takýchto situácií naraz, vtedy sa kódy skombinujú binárnou operáciou *or*. Vhodné kódy sú preto také, ktoré majú práve jeden bit 1, a ostatné 0. Čiže mocniny čísla 2. Keďže typ *long* je 64 bitový, jeden agent dokáže takto rozpoznať 64 rôznych udalostí.

5.1.3 Konfigurácia agentov

Väčšina agentov má nejaké nastaviteľné parametre. Mohli by sa samozrejme nastavovať jednoducho parametrami v konštruktore, alebo trebárs špeciálnymi metódami. My ale chceme agenty konfigurovať cez webové užívateľské rozhranie, takže potrebujeme konfiguráciu nejakým spôsobom štandardizovať. V triede *Agent* je definovaná vnorená trieda *Agent.Cfg*, obsahujúca parametre spoločné pre všetky agenty. Aby bola odvodená agentová trieda konfigurovateľná cez naše webové rozhranie, musí tiež obsahovať vnorenú konfiguračnú triedu, odvodenú z *Agent.Cfg*. Táto konfiguračná trieda môže obsahovať všetky parametre, ktoré agent potrebuje. Agentová trieda musí mať konštruktor, ktorý má jediný parameter, a síce práve tento konfiguračný objekt.

Keď z webového rozhrania príde nová konfigurácia, je to vlastne kolekcia práve takýchto konfiguračných objektov pre jednotlivé agenty. Vďaka tomu, že konfiguračná trieda je vnorenou triedou agentovej triedy, tak s pomocou reflekčného modelu Javy vieme pre každý konfiguračný objekt zistiť, o ktorú agentovú triedu sa jedná. Alternatívne, ak z nejakého dôvodu vývojár nechce aby konfiguračná trieda bola vnorenou triedou agentovej triedy, môže v konfiguračnej triede prekryť metódu *createAgent* triedy *Agent.Cfg*, z ktorej musí vrátiť novo vytvoreného agenta správneho typu.

Základná konfiguračná trieda *Agent.Cfg* obsahuje parametre pre nastavenie časovača a triggrov. Nastavenie časovača je jednoducho perióda v milisekundách. Nastavenie triggrov je komplikovanejšie. V najjednoduchšej forme je to iba zoznam blokov, ale dajú sa špecifikovať aj rôzne kódy pre jednotlivé bloky (použijú sa ako parameter pre *senseSelectAct*), a ako meno bloku sa dá použiť odkaz na iný parameter (*field*) konfiguračného objektu.

5.2 Riadiaci systém

Ako už bolo spomenuté v kapitole 4.1, samotný riadiaci systém tvoria dva špeciálne agenty, na beh ktorých nemá užívateľ žiadny vplyv.

Špeciálny agent *Scheduler* spravuje ostatné bežiace agenty. V systéme je jediná inštancia tohto agenta. Jeho hlavná úloha je po externom podnete (zavolaní metódy *restart*) stopnúť práve bežiace agenty, a spustiť nové, konfiguráciu ktorých dostane ako parameter. *Scheduler* v skutočnosti nestopne bežiaceho agenta ak netreba (teda ak aj v novej konfigurácii je rovnaký). To je výhodné pri agentoch, ktoré majú nejaký vnútorný stav. Najmä ak tento stav je napríklad otvorené sieťové spojenie, alebo niečo iné čo nie je triviálne obnoviť. Takže ak užívateľ cez webové rozhranie zmení nejaké parametre, reštartnú sa iba tie agenty, ktorých sa zmeny týkajú. *Scheduler* si tiež udržuje v pamäti aktuálnu konfiguráciu, aby bola k dispozícii pre webové rozhranie.

Keď *Scheduler* zastavuje práve bežiace agenty, dá im šancu skončiť korektne. To znamená, že im iba dá signál že majú skončiť, a počká kým sa ukončia samé. Ak agent do nejakého času neskončí, *Scheduler* ho považuje za "uviaznutý", a ukončí ho nasilu. Agenty nemusia korektné ukončenie nijak ošetrovať, spraví to za ne základná trieda *Agent*. Ak ale chcú pri skončení niečo vykonať, môžu prekryť metódu *cleanup*.

Hlavný kód *Scheduler*a je teda mimo cyklu sense-select-act, a je volaný zvonku. *Scheduler* ale má aj svoj sense-select-act cyklus, v ktorom kontroluje bežiace agenty, či neskončili. Pôvodný plán bol v tomto prípade agenty prípadne reštartnúť, ale to sa ukázalo ako nepotrebné, nakoľko základná trieda *Agent* ošetruje všetky výnimky v *senseSelectAct*, a teda agenty nepadajú. Takže *Scheduler* iba zaloguje skončenie behu agenta, považujúc ho za plánované. V prípade potreby je však triviálne reštartovanie agentov dorobiť.

Druhý špeciálny agent zabezpečuje spojenie s webovým rozhraním. Tento agent sa volá *Main*, pretože jeho kód sa vykonáva v hlavnom vlákne aplikácie. Je to neštandardný agent, ktorý ani nepoužíva cyklus sense-select-act. To sú však už implementačné detaily.

Komunikácia s webovým rozhraním je jednoduchá. Webové rozhranie nadviaže spojenie, a pošle príkaz, ukončený znakom *newline*. Ďalšia komunikácia závisí od príkazu. Hlavné príkazy sú *GETCFG* a *PUTCFG*. Pri *GETCFG* riadiaci systém jednoducho pošle rozhraniu aktuálnu konfiguráciu bežiacich agentov, získanú od *Scheduler*a, v XML formáte. Pri *PUTCFG* sa príjme nová konfigurácia (v rovnakom XML formáte) a naštartujú sa nové agenty. Ak sa to podarí, pošle sa potvrdenie *OK*, v prípade chyby sa pošle *ERR*. Ešte existuje aj ďalší príkaz *READSPACE*, ktorý slúži na zobrazovanie blokov z prostredia vo webovom rozhraní. Pri tomto príkaze sa prijme ešte meno bloku, a pošle sa obsah daného bloku, zase v XML formáte.

5.3 XML reprezentácia objektov

Na komunikáciu používame špeciálny XML formát, ktorý mapuje objekty z Javy. Je to istá forma serializácie, s jednou veľkou výhodou oproti štandardnej binárnej serializácii Javy. Tou výhodou je priama čitateľnosť a editovateľnosť tohto textového formátu.

Tento XML formát je v princípe jednoduchý. Primitívne typy ukladá ako text. Pre polia, kolekcie, a mapy má špeciálne konštrukty. Ostatné objekty ukladá ako XML uzly obsahujúce elementy pre všetky svoje verejné atribúty (*public field*).

Keďže sa do XML prenášajú iba verejné dáta, nie je to univerzálna serializácia. Objekty ktoré chceme transformovať obojsmerne (rekonštruovať z XML pôvodné objekty), sú navrhnuté tak aby sa do tohto XML dali bezstratovo transformovať. Niekedy je užitočná aj jednosmerná transformácia do XML, bez možnosti rekonštruovať pôvodný objekt. Na to máme ešte jeden mechanizmus. Objekty implementujúce interface *CustomXmlMappable* si vytvárajú náhradné objekty, ktoré sú do XML transformované namiesto nich. Tieto náhradné objekty sú spravidla špeciálnej triedy, podobnej ako pôvodná, len so všetkými atribútmi deklarovanými ako *public*, a obsahujú kópiu pôvodného objektu. Toto umožňuje zobrazovať obsah blokov z prostredia vo webovom rozhraní, bez toho aby sme v triedach ukladaných objektov museli deklarovať všetky atribúty ako *public*. Toto nám umožňuje v prostredí ukladať nemodifikovateľné (*immutable*) objekty, a zároveň ich zobrazovať v XML reprezentácii vo webovom rozhraní. Vďaka tejto technike tiež dokážeme v XML formáte zobrazovať objekty zložitejších tried zo štandardnej knižnice Javy. Stačí z takýchto tried odvodiť nové, a implementovať pre ne interface *CustomXmlMappable*.

Ukážka 1 XML reprezentácia jednoduchej konfigurácie agenta

```
<x xclass="sk.robotics.rlcs.agents.space.CopierAgent$Cfg">
  <blockSource>source</blockSource>
  <blockTarget>target</blockTarget>
  <copyNull>false</copyNull>
  <enabled>true</enabled>
  <logLevel>0</logLevel>
  <triggers>$blockSource</triggers>
</x>
```

5.4 Základné operácie s robotmi

5.4.1 Komunikácia s robotmi

Na komunikáciu s robotmi v našom laboratóriu je pripravená knižnica s podporou pre viaceré programovanie jazyky (autor P. Petrovič [11, 12]). Pre Javu je k dispozícii trieda *Robot*, pomocou ktorej sa dá pripojiť k jednému robotu (cez Bluetooth spojenie), posielať mu príkazy, a zisťovať od neho stav. Roboty rozpoznávajú sadu pohybových príkazov:

- pohybové - daný počet krokov vpred, vzad, alebo točením na mieste doľava alebo doprava
- rýchlostné - nezávislé nastavenie rýchlosti otáčania oboch kolies
- kombinované - daný počet krokov ľubovoľnou rýchlosťou

Existujú ešte aj príkazy na ovládanie prídavného hardvéru (chápadlo, pero), tie ale pre potreby tejto práce nie sú relevantné. Čo sa týka zisťovania stavu robota, vieme si od neho vypýtať tieto hodnoty:

- stav batérie
- hodnoty z pozemných senzorov (nepoužívame)
- odometria (relatívna pozícia počítaná podľa pohybov)

Odometriu si nepočíta priamo robot, ale knižnica cez ktorú sa k nemu pripájame.

Spojenie s robotmi v systéme RLCS majú na starosti agenty triedy *ConnectionAgent*, pre každého robota jeden. Jeho činnosť spočíva iba v nadväzovaní a udržiavaní spojenia s robotom, pohyby už majú na starosti iné agenty. To znamená že agenty musia zdieľať komunikačný *Robot* objekt, čo sa dá vďaka tomu, že v prostredí sa ukladajú iba referencie na objekty. Samozrejme pri tom treba dávať pozor, a synchronizovať všetky prístupy k tomuto objektu. Toto

je trochu nešťastné, a je to aj proti zásadám Agent-Space, ale rozhodol som sa pre túto variantu, pretože cez tento objekt sa realizuje viacero činností, ktoré sú príliš nezávislé na to aby ich mal na starosti jediný agent.

Komunikačný objekt vytvorí práve *ConnectionAgent*, ktorý ho po úspešnom nadviazaní spojenia uloží do prostredia, aby ho mohli použiť iné agenty, ktoré chcú robotom hýbať, alebo zisťovať jeho stav. Ďalšia činnosť tohto agenta spočíva v kontrolovaní, či je spojenie stále živé. Po prerušení spojenia *ConnectionAgent* komunikačný objekt z prostredia zase odstráni, a skúša sa pripojiť znova.

5.4.2 Detekcia robotov

Keďže zatiaľ ako jediný senzor používame hornú kameru, implementácia žiadneho aspoň trochu zmysluplného správania sa nezaobíde bez detekcie pozície robota z kamerového obrázku. Detekcia pozostáva z dvoch častí: za prvé samotné rozpoznanie pozícií robotov na obrázku, a za druhé identifikácia robotov, teda priradenie rozpoznaných pozícií jednotlivým robotom (resp. spojeniam cez ktoré robotom posielame príkazy).

Prvú časť, teda rozpoznávanie z obrázku momentálne nerealizujeme priamo agentmi v systéme RLCS. Na počítači v našom robotickom laboratóriu totiž okrem RLCS beží niekoľko ďalších aplikácii, a jedna z nich už toto rozpoznávanie robí. Takže namiesto duplikovania tejto práce, a zbytočného zaťažovania procesora, máme iba agenta ktorý načítava výstup z tejto aplikácie. Týmto sa samozrejme ušetrí veľa výpočtov, ale na druhej strane máme nad rozpoznávaním málo kontroly.

Existuje samozrejme alternatíva, a síce rozpoznávanie priamo v systéme RLCS sadou agentov. Tieto agenty by mohli tvoriť *"pipeline*", ktorá by pracovala nasledovne:

1. kalibrácia obrázku - odstránenie zakrivenia spôsobeného objektívom kamery
2. detekcia hrán (Sobelov operátor [14])
3. konverzia obrázku na binárny čierno-biely (prahová transformácia)
4. detekcia kružníc (Houghova transformácia [15])
5. identifikácia pozícií robotov podľa nájdených kružníc

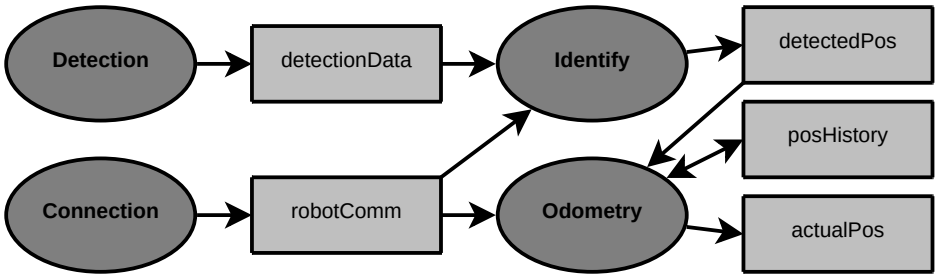
Pre kroky 2 a 3 už aj máme pripravené hotové agenty. Zostáva dorobiť agenty pre zvyšné kroky, to už je ale mimo rámec tejto práce, keďže detekované pozície robotov máme z iného zdroja. Čo sa týka kroku 5, roboty sú okrúhle, a sú na nich nalepené malé tiež okrúhle terčíky kvôli rozpoznávaniu smeru natočenia, takže podľa zdetekovaných kružníc naozaj dokážeme zistiť pozíciu a aj smer robotov.

Druhá časť detekcie robotov spočíva v identifikácii konkrétnych robotov medzi zdetekovanými pozíciami. Na to slúži agent *IdentifyAgent*, ktorý pracuje nasledovne:

1. ak zostáva jediná neidentifikovaná pozícia, použije tú
2. ak už robot má staršiu zdetekovanú pozíciu, a medzi novými je nejaká jej podobná (málo vzdialená), použije tú
3. ak robotom môžeme hýbať, otočí ho o 90 stupňov, a medzi novými zdetekovanými pozíciami podľa toho nájde tú správnu

Pravidlo 1 umožňuje systému pracovať aj s jedným robotom ktorým nemôže hýbať. To nám umožní hýbať jedným s robotov manuálne, a nechať RLCS ovládať ostatné roboty. Takto sa dajú robiť rôzne zaujímavé experimenty.

Zostáva však ešte jeden problém. Obrázok z kamery dostávame s nejakým oneskorením, a samotná detekcia tiež nejaký čas trvá. To znamená, že nikdy nepoznáme aktuálnu pozíciu robotov, ale iba pozíciu z nejakého okamihu v minulosti. Toto čiastočne riešime tým, že pomocou odometrie si pamätáme históriu relatívnych pohybov (stará sa o to *OdometryAgent*). Podľa tejto histórie a zdetekovanej pozície z minulosti si potom vypočítame aktuálnu pozíciu. Tento prístup samozrejme nefunguje stopercentne, ale ako aproximácia má celkom dobré výsledky.



Obrázok 5.2: Získavanie pozície robota
Agenty sú znázornené elipsami, bloky v prostredí obdĺžnikmi.
Smer šípok ukazuje, či agent blok číta alebo doňho zapisuje.

5.4.3 Pohyb robotmi

Posielanie pohybových príkazov robotom má na starosti agent *MovementAgent*. Pre každého robota ovládaného systémom RLCS by mala existovať jedna inštancia tohto agenta. Jeho hlavná činnosť spočíva v tom, že číta z prostredia aký pohyb by mal robot vykonávať (tj. rýchlosť a smer točenia oboch kolies), a posiela robotovi zodpovedajúce príkazy. Agenty, ktoré chcú hýbať robotmi, teda musia zapísať požadovaný pohyb do prostredia. Pomocou vhodného nastavenia parametrov zápisu do prostredia (priorita, validita) môžu rôzne agenty hýbuce robotmi medzi sebou súperiť, spolupracovať, alebo jeden druhého potláčať.

MovementAgent zároveň kontroluje hroziace kolízie, a robota v prípade nebezpečenstva nárazu zastaví. To je veľmi dôležité, pretože roboty by sa nárazmi mohli poškodiť. V detekovaní hroziacich kolízií *MovementAgent*ovi pomáha ďalší agent. Je to *CollisionAgent*, ktorý v pravidelných intervaloch ukladá do prostredia objekt, obsahujúci informácie o okolitých prekážkach pre daného robota. Tento objekt má metódy pre zisťovanie vzdialeností najbližších prekážok v ľubovoľnom smere od robota. Jedna inštancia *CollisionAgent*a by mala byť v systéme pre každého robota, dokonca aj pre tých, ktorých RLCS neovláda. Informácie o hroziacich kolíziách totiž využíva aj *OdometryAgent* (viď časť 5.4.2), ktorý sa snaží detekovať prípadné prešmykovanie kolies robota.

Agenty realizujúce pohyby robotov na vyššej úrovni budú popísané v nasledujúcej sekcii 5.5.

5.5 Jednoduché správania robotov

V tejto sekcii popíšem agenty realizujúce jednoduché pohybové správania robotov. S týmito agentmi je už možné uskutočniť niektoré zaujímavé experimenty (viď kapitolu 6).

5.5.1 Pohyb po predpísanej trase

Implementácia pohybu po predpísanej trase je založená na agentovi *ApproachAgent*, ktorý robotom hýbe smerom k aktuálnemu cieľu, vyčítanému z prostredia. Točí robotom, kým ho nenasmeruje správnym smerom, a potom ním hýbe vpred až kým sa nedostane na cieľové miesto. Priamy prístup k tomuto miestu samozrejme nemusí byť voľný, v ceste môže zavadzať nejaká prekážka. *ApproachAgent* sa o to ale nestará. Ak je v ceste prekážka, robot sa jednoducho zastaví (postará sa o to *MovementAgent*, viď časť 5.4.3).

Nad týmto je postavený ďalší agent *PathAgent*. Ten zadáva aktuálny cieľ cesty (ukladá ho do prostredia). Keď sa robot k cieľu dostane, *PathAgent* zadá nový cieľ. Trasa, teda postupnosť bodov, sa zadáva v konfigurácii tohto agenta.

Toto veľmi jednoduché správanie je zaujímavé najmä ak sa spojí s obchádzaním prekážok. Vtedy môžme jedného robota nechať krúžiť po nejakej predpísanej trase, a druhého ovládať ručne, pričom budeme tomu prvému schválne zavadzať, a sledovať ako sa správa. Viac o obchádzaní prekážok je v časti 5.5.3.

5.5.2 Náhodné potulovanie

Ďalšie základné správanie je náhodné potulovanie robota. Jeho implementáciu som postavil tiež nad agentom *ApproachAgent*, ktorý hýbe robotom za cieľom. Agent, ktorý sa stará o náhodné potulovanie sa volá *RoamAgent*. Pracuje podobne ako *PathAgent*, s tým rozdielom že po

dosiahnutí cieľa sa ďalší vyberie náhodne. Problém nastane ak sa vyberie cieľové miesto ku ktorému nie je voľný prístup, robot sa potom k nemu nikdy nedostane, a tým sa jeho potulky skončia. Preto *RoamAgent* obsahuje ešte poistku, ktorá tomu zabráni. Ak sa robot príliš dlho nevie dostať na cieľové miesto, vyberie sa náhodne nový cieľ.

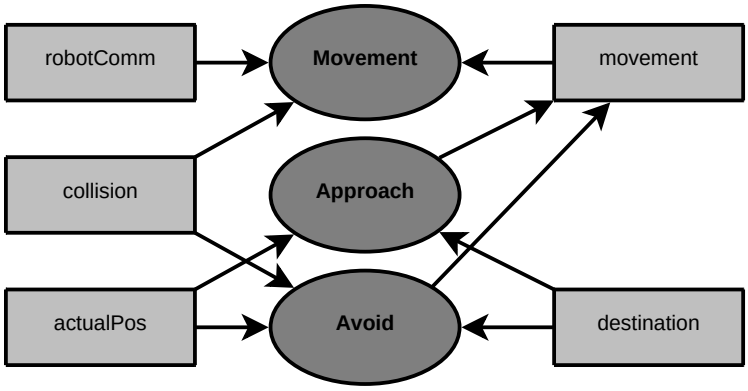
Náhodné potulovanie sa rovnako ako pohyb po predpísanej trase dá vylepšiť vyhýbaním sa prekážkam (viď časť 5.5.3). Pohyb robota je potom plynulejší, "nezadrháva" sa pred každou prekážkou.

Využití náhodného potulovanie je niekoľko. Napríklad, je to jednoduchý spôsob prehľadávania scény. Ďalšie použitie môže byť, ak chceme aby robot nepredvídateľne zavadzal inému pri jeho zložitejšom správaní.

5.5.3 Obchádzanie prekážok

Obchádzanie prekážok musí byť súčasťou každého aspoň trochu rozumného pohybového správania. Mení celkom jednoduché a primitívne správania na zložitejšie, ťažko predvídateľné, a preto zaujímavé.

Väčšina pohybových správaní, ktoré som v systéme RLCS implementoval, je založená na agentovi *ApproachAgent*, ktorý sa stará o to aby robot išiel priamym smerom k aktuálnemu cieľu. Ak je však v ceste prekážka, *ApproachAgent* nevie čo ďalej, a robot sa zastaví. O obchádzanie prekážok sa stará *AvoidAgent*, ktorý detekuje situáciu keď robot zastaví kvôli prekážke, a vtedy ho začne ovládať sám.



Obrázok 5.3: Pohyb robotom k cieľovému miestu
Bloky na ľavej strane sú zapísané inými agentmi. Agenty z vyšších vrstiev môžu pohybovať robotom k cieľu zapisovaním do bloku *destination*.

AvoidAgent sa podľa tvaru prekážky pred robotom rozhodne, z ktorej strany ju obísť, a potom ide zvoleným smerom okolo nej. Pokračuje až kým nastane situácia, že robot má voľný kus cesty smerom k aktuálnemu cieľu. Vtedy prenechá riadenie robota zase *ApproachAgent*ovi.

AvoidAgent používa vyššiu prioritu zápisu požadovaného pohybu do prostredia ako *ApproachAgent*, čo zaručuje že nenastane konflikt. Ten by inak nastal, pretože akonáhle *AvoidAgent*

pohne robotom aspoň kúsok preč od prekážky, *ApproachAgent* by sa opäť snažil hýbať robotom smerom k cieľu.

5.5.4 Sledovanie druhého robota a napodobňovanie pohybov

S pomocou *ApproachAgent*a, ktorý pohybuje robotom smerom k danému cieľu, môžeme veľmi jednoducho dosiahnuť aj ďalšie zaujímavé správanie. Aby jeden robot sledoval druhého, stačí nám ako aktuálny cieľ prvého robota používať aktuálnu pozíciu druhého. Ak k tomu pridáme ešte aj obchádzanie prekážok (v podobe *AvoidAgent*a), vznikne z toho naozaj celkom zaujímavé správanie, a pritom veľmi jednoduché.

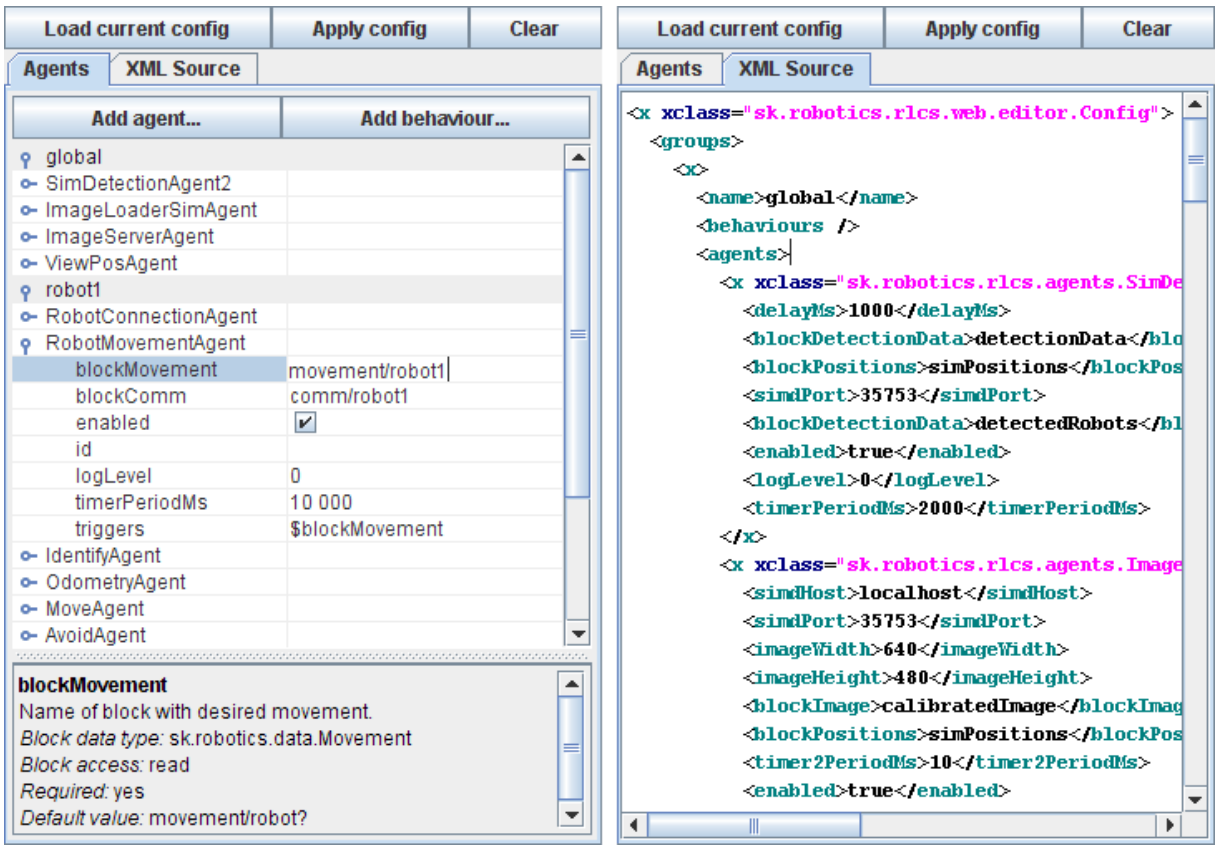
Pripravený je aj agent realizujúci kopírovanie pohybov druhého robota. To je podstatne zložitejšie, ako jednoduché prenasledovanie. Kopírovanie pohybov realizuje *ImitateAgent*, ktorý na to využíva históriu pohybov robota, ktorú generuje *OdometryAgent* (viď časť 5.4.2). Podľa tejto histórie zisťuje aké boli posledné pohyby druhého robota, a opakuje ich po ňom. Treba ešte dodať, že reč je o napodobňovaní pohybov robota, ktorého nemusí ovládať systém RLCS, ale môže to byť aj iná aplikácia alebo manuálne užívateľ. Ak by sme totiž chceli napodobňovať robota ktorého ovláda RLCS, stačilo by kopírovať blok s aktuálnym pohybovým príkazom pre napodobňovaného robota.

5.6 Webové rozhranie

Ako už bolo spomenuté, konfigurácia systému RLCS je tvorená množinou konfiguračných objektov jednotlivých agentov. Webové rozhranie používa reflekčný model Javy na analýzu týchto objektov, a na ich následné zobrazenie a upravovanie. Pre jeho funkciu teda nie je potrebný žiadny ďalší popis agentov, stačia samotné konfiguračné objekty. To je výhodné, lebo nie je potrebné udržiavať duplicitné informácie o agentoch. Máme len zdrojový kód agentov, a z jeho zkompilovanej podoby si už webové rozhranie všetko potrebné vyčíta.

Webové rozhranie ponúka aj používanie celých správaní, čo sú preddefinované skupiny agentov. Správania sú popísané v XML súbore, ktorý sa pri štarte appletu načíta. V tomto súbore je pre každé správanie definovaná sada agentov, aj s parametrami. Ako parametre sa v ňom môžu používať premenné, označené ako *%meno-premennej%*. Webové rozhranie nemá prístup priamo k agentom v správaniach, ale dajú sa cezeň meniť práve tieto premenné. Rovnako ako parametre pri normálnych agentoch. Premenné sa potom automaticky propagujú do parametrov agentov v správaní.

Pri editovaní konfigurácie, a tiež pri pridávaní nových agentov a správaní, sa zobrazuje aj popis jednotlivých agentov, správaní, a ich parametrov. Pre správania sú tieto popisy samozrejme v XML súbore s definíciami správaní. Popisy agentov a ich parametrov sa načítajú z ďalšieho



Obrázok 5.4: Editor vo webovom rozhraní

Vľavo editor agentov, vpravo XML editor.

XML súboru. Tento však nemusíme udržiavať ručne. Generuje sa automaticky zo zdrojového kódu agentov. Konkrétne z *Javadoc* komentárov v zdrojovom kóde. Pomocou nástrojov patriacich k Jave vytiahneme tieto komentáre do XML súboru, ktorý sa následne pomocou XSLT pretransformujeme do formátu, ktorému rozumie naše webové rozhranie. Tento súbor zároveň slúži aj ako zoznam všetkých možných použiteľných agentov, ktoré budú na výber pri pridávaní nových agentov do konfigurácie.

Webový applet prejde všetky parametre konfiguračných objektov v aktuálnej konfigurácii, a zobrazí ich v prehľadnom editore. Užívateľ môže tieto parametre meniť, a pridávať nových agentov, alebo správania. Keď je konfigurácia pripravená, jednoducho jedným klikom sa pošle na server systému RLCS, a ihneď sa uvedie do platnosti.

K editoru agentov a parametrov ponúka webové rozhranie aj alternatívu. Je ňou XML editor, kde môžme priamo editovať XML reprezentáciu konfigurácie. Teda priamo to čo sa pošle systému RLCS. XML editor podporuje zvýrazňovanie syntaxe, a bežné funkcie ako copy/paste, alebo undo/redo. Kvôli bezpečnostným obmedzeniam Java appletov nie je možné konfiguráciu priamo uložiť alebo načítať zo súboru na lokálnom disku, avšak pomocou XML editora a copy/paste sa to dá ľahko obísť. V budúcnosti plánujem tento nedostatok ešte nejak napraviť, pretože ukladanie konfigurácie na lokálnom disku je veľmi užitočná funkcia, ktorú by bolo dobré priamo podporovať.

Applet pracuje iba s konfiguračnými objektmi agentov, a nie so samotnými agentmi. To znamená, že do webového prehliadača užívateľa sa nemusí sťahovať kód agentov, stačia ich konfiguračné objekty. A to je nezanedbateľná výhoda, lebo konfiguračné objekty sú malé, spravidla neobsahujú žiaden kód.

5.7 Bezpečnosť

Systém RLCS umožňuje komukoľvek zvonku cez internet zasahovať do behu počítača v robotickom laboratóriu. Preto nesmieme zabúdať ani na bezpečnosť. Pekný príklad nebezpečnosti systému je agent *ExecuterAgent*, ktorý umožňuje spúšťať externé aplikácie. Potrebujeme ho na komunikáciu s inými systémami bežiacimi na našom serveri. Obmedziť tohto agenta len na niekoľko potrebných externých aplikácií by samozrejme bolo jednoduché, ale takýchto agentov môže byť časom viac, a je lepšie vyriešiť to všeobecnejším bezpečnostným modelom.

Systém RLCS má ambíciu umožniť experimentovanie s našim robotickým laboratóriom pre užívateľov z celého sveta. Chceme im dať aj možnosť doprogramovať si vlastných agentov. S bezpečnostným modelom, ktorý som implementoval, administrátor ani nemusí nejak veľmi kontrolovať kód nových agentov, ktorých nám prípadní záujemcovia pošlú.

Bezpečnostný model je založený na oprávneniach, ktoré sa explicitne prideľujú jednotlivým agentom. Tieto sú definované v XML súbore, ktorý systém RLCS načíta pri spustení. Ak agent nemá žiadne oprávnenie, nemá prístup k žiadnym dôležitým zdrojom. Nemôže čítať z disku ani naň zapisovať, nemôže nadväzovať sieťové spojenia, nemôže zisťovať údaje o systéme, a nemôže ani používať reflekčný model na prístup k iným ako verejným premenným a metódam objektov. Jednoducho, nemôže robiť nič, čo je potenciálne nebezpečné. Môže však narábať s prostredím, čo by v spolupráci s už existujúcimi agentmi malo stačiť na akékoľvek správanie. Ak by cudzí agent predsa len potreboval nejaké špeciálne oprávnenia, môžme mu ich po zvážení udeliť.

Kontrolu oprávnení nám umožňuje bezpečnostný model Javy. Pri štarte si RLCS zaregistruje svoj vlastný *SecurityManager*, a Java potom volá jeho metódu *checkPermission* pri každom prístupe k dôležitým zdrojom. Podľa vlákna, z ktorého je táto metóda zavolaná, zistíme o akého agenta sa jedná, a skontrolujeme či má dané oprávnenie. Agenty, ktoré na to majú oprávnenie, môžu aj spúšťať ďalšie vlákna. Tieto potom majú rovnaké oprávnenia, ako materský agent.

Kontrola oprávnení sa robí pre všetky agenty, aj pre tie ktoré som napísal sám. Niektoré z nich potrebujú špeciálne oprávnenia. Tie sú im pridelené v spomínanom XML súbore. Jedná sa o povolenie pripojenia k robotom, komunikácie s inými systémami na serveri, a spojenie s webovým rozhraním.

Jednotlivé oprávnenia môžu byť dosť podrobné. Agentom môžme povoliť prístup k celému lokálnemu disku, alebo iba ku konkrétnym súborom (podporované sú aj masky). Môžme im

povoliť akékoľvek sieťové spojenie, alebo iba s konkrétnymi adresátmi. Môžme im povoliť zistiť jednotlivé parametre operačného systému, bez toho aby sme im dali prístup k ostatným.

Kapitola 6

Experimenty

V tejto kapitole popíšem ako sa dajú s implementovanými agentmi v systéme RLCS zrealizovať niektoré zaujímavé experimenty.

6.1 Obchádzanie pohyblivej prekážky

Hoci pri programovaní agentov pre pohyb robotom a obchádzanie prekážok som myslel len na obchádzanie statických prekážok (a aj som to počas vývoja testoval iba na statických prekážkach), ukázalo sa, že rovnaké agenty celkom slušne fungujú aj pre pohyblivé prekážky. Takýto experiment môžme zrealizovať napríklad tak, že jedného robota necháme pohybovať sa po nejakej predpísanej dráhe (použijeme na to *PathAgent*a), a druhého robota budeme ovládať manuálne, a budeme ním schválne zavadzať tomu prvému.

Pre dosiahnutie takejto konfigurácie vo webovom rozhraní RLCS postupujeme nasledovne:

1. Pridáme správanie *Image and detection*, pre všetky parametre necháme predvolené hodnoty. (Ak experiment realizujeme so simulátorom, namiesto tohto správania vyberieme *Simulated image and detection*).

2. Pridáme správanie *Robot common active*, nastavíme parameter *robotNum* na 1, a parameter *robot* na robota ktorý sa má pohybovať po predpísanej trase, a obchádzať prekážky (na výber máme 3 skutočné a 2 simulované roboty).

3. Pridáme správanie *Robot common passive*, nastavíme parameter *robotNum* na 2, a parameter *robot* na robota ktorého chceme ovládať manuálne.

4. Pridáme agenta *beh.PathAgent*, a nastavíme mu parametre nasledovne:

Parameter	Hodnota
blockActualPos	actualPos/robot1
blockDestination	destination/robot1
path	150:600 1350:600

Prvé správanie zabezpečí načítavanie obrazu z kamery, a zdetekovaných pozícií robotov, a tiež vysielanie obrazu do webového rozhrania. Druhé a tretie správanie pridajú základné agenty pre prácu s robotmi, jedno pre robota ktorého ovláda RLCS, a druhé pre robota ovládaného externe. Agent *PathAgent* realizuje pohyb robota po predpísanej dráhe, pričom spolupracuje s agentmi z druhého správania.

XML reprezentáciu tejto konfigurácie vidíme v ukážke 2. Na obrázku 6.1 je zase diagram agentov a blokov v prostredí, ktoré sa pri tejto konfigurácii používajú.

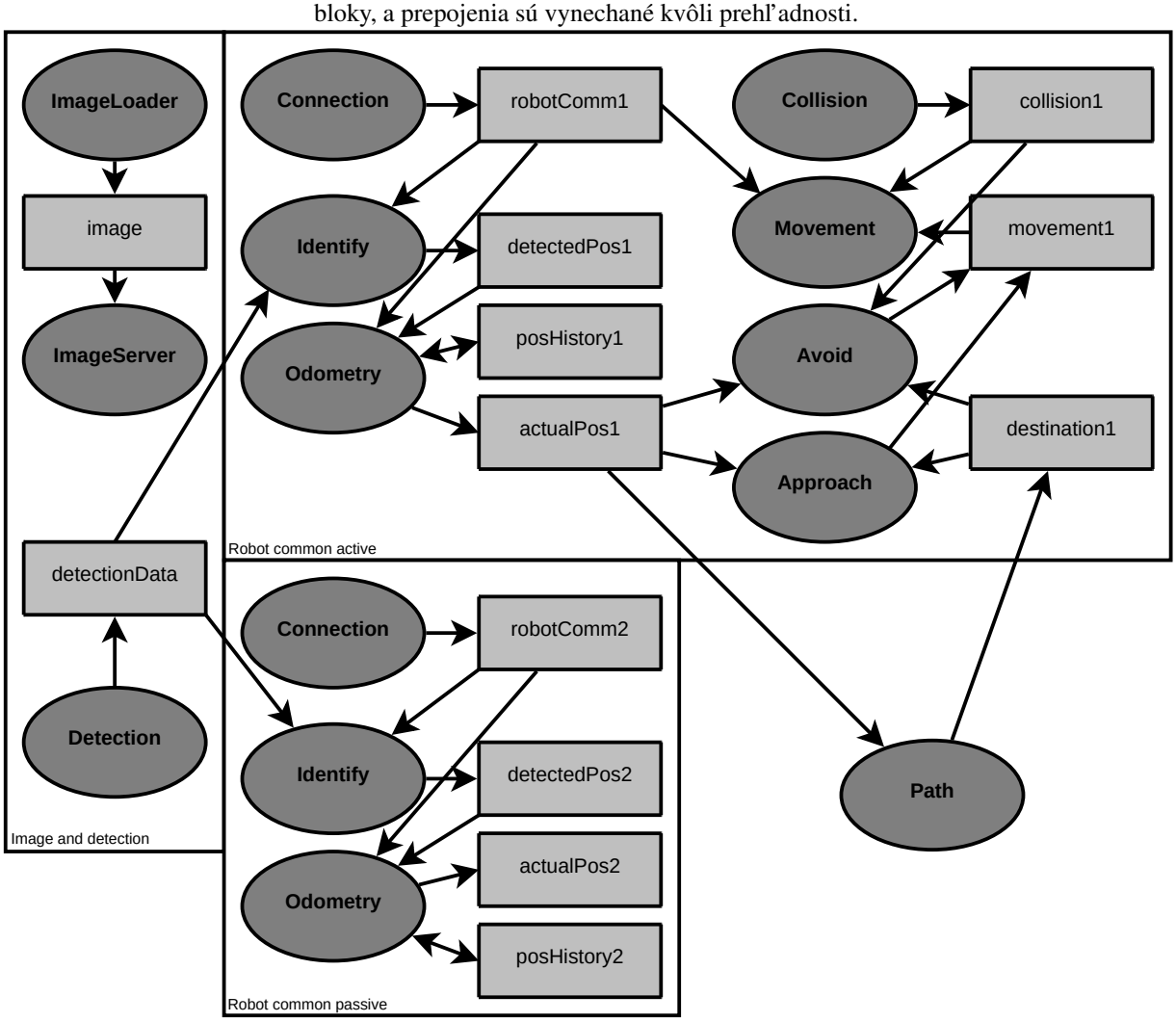
Ukážka 2 XML reprezentácia konfigurácie experimentu 6.1

```
<x xclass="sk.robotics.rlcs.web.editor.Config">
  <groups>
    <x name="">
      <behaviours>
        <x name="Image and detection">
          <params />
        </x>
        <x name="Robot common active">
          <params>
            <x key="robotNum"><value xclass="Integer">1</value></x>
            <x key="robot"><value xclass="...">SIM1</value></x>
          </params>
        </x>
        <x name="Robot common passive">
          <params>
            <x key="robotNum"><value xclass="Integer">2</value></x>
            <x key="robot"><value xclass="...">SIM2</value></x>
          </params>
        </x>
      </behaviours>
      <agents>
        <x xclass="sk.robotics.rlcs.agents.beh.PathAgent$Cfg">
          <path>150:600 1350:600</path>
          <blockActualPos>actualPos/robot1</blockActualPos>
          <blockDestination>destination/robot1</blockDestination>
        </x>
      </agents>
    </x>
  </groups>
</x>
```

Treba ešte dodať, že táto konfigurácia bude fungovať iba ak sú na stole práve dva roboty. Ak by ich tam bolo viac, pre každého ďalšieho by sme museli pridať jedno správanie *Robot common active*. To preto, aby systém RLCS dokázal identifikovať robota ktorým nemôže hýbať (toho

Obrázok 6.1: Agenty a bloky počas experimentu 6.1

Agenty sú opäť znázornené elipsami, bloky obdĺžnikmi, a smer šípok ukazuje či agent blok číta alebo zapisuje. Veľké obdĺžniky znázorňujú preddefinované správania. Diagram je ešte trochu zjednodušený, niektoré agenty, bloky, a prepojenia sú vynechané kvôli prehľadnosti.



ktorým chceme zavadzať druhému). To je totiž možné iba vtedy, ak je tento robot posledný neidentifikovaný na stole.

Konfiguráciu teda odošleme na server, kde automaticky vstúpi do platnosti. Po úspešnom zdetekovaní pozícií robotov sa začne jeden z nich pohybovať po predpísanej dráhe, čo je v tomto prípade z jednej strany stolu laboratória na druhú, a späť, stále dokola.

Na pravej strane rozhrania si môžme prepínať medzi priamym prenosom z kamery, a spracovaným obrazom z RLCS. V spracovanom obraze vidíme zdetekované pozície robotov, a tiež aktuálny cieľ pohybu robota, ktorý sa pohybuje po predpísanej dráhe, ale ak chceme mať plynulejší (menej sekaný) obraz, lepší je priamy prenos z kamery.

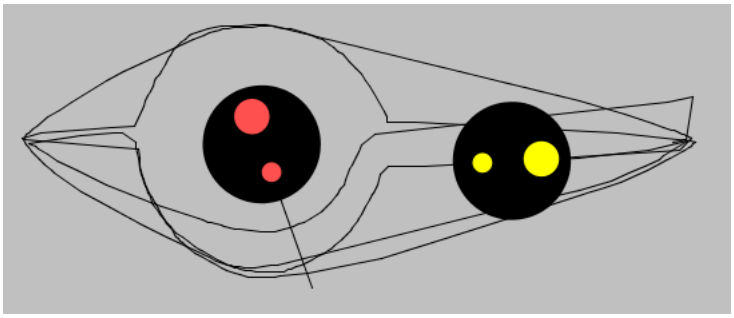
Následne môžme ľavú stranu rozhrania prepnúť na manuálne ovládanie robota, pripojiť sa k druhému robotovi, a začať ním hýbať. Ak sa tomu prvému robotovi jednoducho postavíme do cesty, a nebudeme sa hýbať, bez problémov nás obíde. Takže začneme pokusy s pohyblivou

prekážkou. Uvidíme, že obchádzanie už nie je také hladké ako predtým, ale väčšinou predsa funguje. Veľmi to záleží na tom, akým smerom sa prekážka hýbe, relatívne k smeru pohybu obchádzajúceho robota. Problém nastáva, ak sa prekážajúci robot hýbe smerom k obchádzajúcemu. Vtedy niekedy dochádza k nárazom, napriek snahe agentov zabrániť im. Kvôli nárazom a hlavne prešmykovaniu kolies potom zlyháva odometria, a tým pádom má RLCS na chvíľu nesprávnu pozíciu robota. To je samozrejme veľmi nepríjemné, lebo počas toho času môže RLCS riadiť robota do ďalších kolízií. Tento nepríjemný stav trvá, až kým RLCS problém neodhalí, čo sa väčšinou stane po ďalšom získaní zdetekovaných pozícií z kamerového obrázku. Niekedy to kvôli oneskoreniu detekcie môže trvať ešte o chvíľu dlhšie, v najhoršom prípade to býva niekoľko sekúnd. Keď RLCS (konkrétne *IdentifyAgent*) problém odhalí, zastaví robota a nanovo ho identifikuje (viď sekciu 5.4.2).

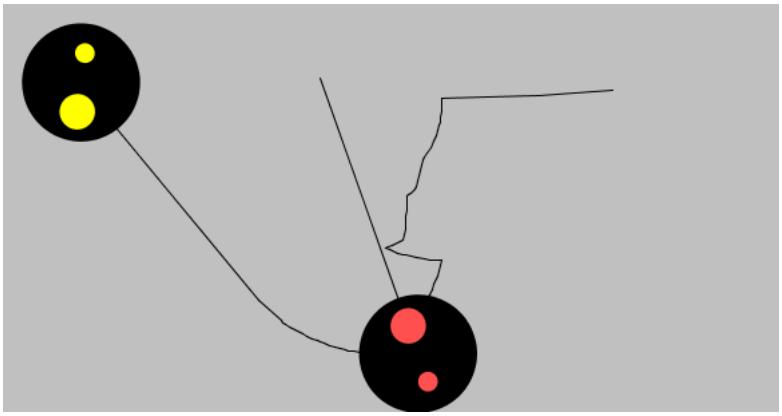
Problém so stratou synchronizácie detekovanej pozície robotov so skutočnou pozíciou niekedy nastáva aj bez nárazov a prešmykovania. Avšak paradoxne, tento problém môže byť niekedy aj výhodou, vnáša totiž do pohybu ďalší nedeterministický prvok, čo môže robotom pomôcť dostať sa zo situácií, z ktorých by sa za bezchybného priebehu dostať nevedeli. Svedkami takejto situácie môžme byť aj pri experimente o ktorom píšem v tejto kapitole. Ak sa totiž so zavadzajúcim robotom postavíme druhému robotovi priamo do cesty, počkáme ktorým smerom sa nás rozhodne obchádzať, a v tom momente sa začneme hýbať tým smerom, kolmo na jeho pôvodnú dráhu, tak to dopadne tak, že roboty pôjdu paralelne vedľa seba, až na koniec stola. Tu môže obchádzajúcemu robotovi problém z detekciou alebo odometriou prísť vhod ako náhodný impulz.

Takže aby som to zhrnul, obchádzanie pohyblivých prekážok funguje celkom dobre, s tým že obchádzajúci robot sa občas o zavadzajúceho zasekne, ale po čase sa z toho väčšinou sám dostane. Niekedy sa to bohužiaľ nezaobíde bez malých nárazov, ale tomu sa pri problematickej detekcii pozícií dá ťažko úplne vyhnúť. Ešte raz zdôrazním, že obchádzanie pohyblivých prekážok realizujú agenty, ktoré na to neboli navrhnuté (boli navrhnuté na obchádzanie statických prekážok). Tento jav sa niekedy označuje ako *brikoláž* [9].

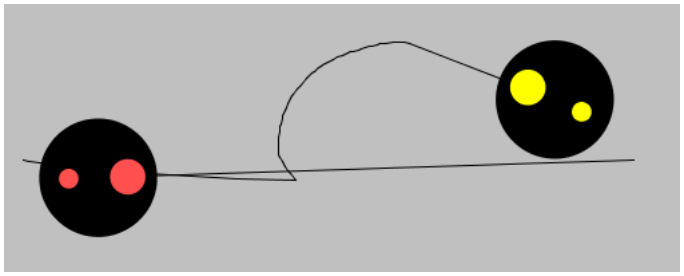
Obrázky 6.2 až 6.4 ukazujú tento experiment realizovaný v simulátore robotického laboratória (viď dodatok D). Je na nich pekne vidieť, po akej trase sa roboty pohybovali.



Obrázok 6.2: Experiment 6.1 #1
Zavadzajúci robot stojí v strede a nehýbe sa, druhý robot chodí z jednej strany na druhú a bez problémov ho obchádza.



Obrázok 6.3: Experiment 6.1 #2
Zavadzajúci robot sa hýbal tak aby sa obchádzajúci cez neho nedostal. Tomu sa však, hoci ťažko, nejak podarilo prešmyknúť sa popri stene, a potom už bez problémov pokračoval k svojmu cieľu.



Obrázok 6.4: Experiment 6.1 #3
V tomto prípade išli roboty priamo proti sebe. Obchádzajúci robot si s tým poradil, napriek úvodným problémom po zrážke.

6.2 Napodobňovanie druhého robota

Ďalší zaujímavý experiment, ktorý môžme s implementovanými agentmi zrealizovať, je napodobňovanie pohybov jedného robota druhým. Práve toto robí pripravený agent *ImitateAgent*. Nestará sa o vzájomnú pozíciu daných robotov, iba po druhom robotovi opakuje pohyby. Ak teda chceme s robotmi urobiť nejaký zaujímavý synchronizovaný "tanec", musíme roboty vopred dostať do požadovanej vzájomnej polohy. Na to by mohol ľahko slúžiť ďalší agent, ktorý by mal väčšiu prioritu ako *ImitateAgent*, a jeho úlohou by bolo dostať robota do požadovanej polohy. Takýto experiment však môžme bez problémov zrealizovať aj tak, že roboty najskôr manuálne nastavíme do vhodnej vzájomnej pozície.

Konfigurácia vo webovom rozhraní RLCS pre tento experiment je podobná ako pri predchádzajúcom experimente, s jediným rozdielom. Namiesto *PathAgent*a pre robota ovládaného systémom RLCS použijeme *ImitateAgent*a. Tomu nastavíme parametre nasledovne:

Parameter	**Hodnota**
blockTargetHistory	posHistory/robot2
blockActualPos	actualPos/robot1
blockMovement	movement/robot1

ImitateAgent je zatiaľ implementovaný veľmi jednoducho, používa iba priamy pohyb vpred, vzad, a otáčanie robota na mieste. Čiže ak budeme s ručne ovládaným robotom robiť zložitejšie pohyby, napodobňovanie nebude perfektné. Pri bežných pohyboch robotom však vidíme, že napodobňovanie funguje celkom dobre. Ak má napodobňujúci robot dosť miesta, pekne kopíruje pohyby druhého robota, s malým oneskorením, ktoré je nastaviteľné ako parameter agenta (meno parametra je *delayMs*). Predvolená hodnota tohto oneskorenia je pol sekundy (500 milisekúnd).

Kapitola 7

Záver

V tejto práci som vybudoval programovateľný riadiaci systém pre roboty v robotickom laboratóriu. Systém je interaktívny, správanie robotov sa dá za behu meniť, a výsledok vidíme v priamom prenose. Je prístupný pre kohokoľvek cez grafické webové rozhranie, a programovanie v ňom je vizuálne, netreba sa učiť žiadny zložitý programovací jazyk. Systém spolu s robotmi sú prístupné 24 hodín denne.

Hlavným cieľom práce bolo vytvoriť nástroj na podporu experimentovania s robotickým laboratóriom. Ako vidno v kapitole 6, tento cieľ sa celkom podarilo naplniť. Samozrejme, aby bol vytvorený systém reálne používaný a užitočný, bude treba na ňom ďalej pracovať, hlavne dopĺňať ho ďalšími agentmi.

Systém som založil na architektúre používajúcej výlučne nepriamu komunikáciu medzi modulmi. Tým som ukázal, že takáto schéma je tiež vhodná na vizuálne programovanie, rovnako ako štandardne používaná schéma s priamym spájaním vstupov a výstupov jednotlivých modulov. To bolo druhým cieľom tejto práce, ktorý je teda tiež splnený.

Vizuálne prostredie, ktoré som k riadiacemu systému spravil, je síce pomerne jednoduché, ale ak to ide takto, vieme si ľahko predstaviť aj sofistikovanejšie prostredie, kde by sa moduly "naťahali" (*drag and drop*) na plochu ako v iných moderných vizuálnych prostrediach, len namiesto spájania medzi sebou by sa spájali s objektmi reprezentujúcimi bloky v prostredí. Toto by mohol byť dobrý námet na prípadné pokračovanie mojej práce.

Ďalší námet na pokračovanie je samozrejme dopĺňanie nových agentov do systému. Zdokonaliť prácu s obrazom a detekciu robotov, pridať zložitejšie správania, zapojiť do správaní aj prídavný hardvér robotov, možnosti sú naozaj veľké.

Systém, ktorý som vytvoril, by mal slúžiť ako experimentálna platforma pre potreby prípadných ďalších diplomových (alebo iných) prác, zaoberajúcich sa robotikou.

Dodatok A

Návod na použitie

Systém RLCS nie je bežná aplikácia, ktorá by sa dala jednoducho otestovať na hociakom počítači. Možné to síce je, buď s použitím simulátora (viď dodatok D), alebo dokonca s ovládaním skutočných robotov v robotickom laboratóriu cez internet, ale postup je pomerne komplikovaný, a existuje jednoduchšia alternatíva. Tou je použiť stále bežiaci systém na počítači priamo v robotickom laboratóriu, ktorý je prístupný cez internet.

K rozhraniu robotického laboratória sa dá dostať na tejto adrese:

`http://virtuallab.kar.elf.stuba.sk/`

Tam vidíme, aké roboty sú dostupné. Dostupné sú skoro vždy dva simulované roboty v simulovanej aréne, a ak nemáme smolu, tak aj aspoň jeden skutočný robot. Pre vstup do laboratória klikneme na obrázok jedného z robotov. Ak chceme pracovať so skutočnými robotmi, tak na skutočného robota, ak so simulovanými, tak na simulovaného. To na ktorého robota klikneme určuje ktorého robota budeme môcť ovládať ručne (samozrejme neskôr sa to dá prepnúť). RLCS má prístup vždy ku všetkým robotom.

Po kliknutí na jedného z robotov sa už dostaneme do rozhrania robotického laboratória. Na jeho ľavej strane sa dá prepínať medzi apletom pre ručné ovládanie robota (*Simple control*), ktorý je zobrazený aj na začiatku, a rozhraním systému RLCS (*RLCS control*). Na pravej strane sa prepína medzi pohľadmi z rôznych kamier (v prípade simulátora je iba pohľad z hora), a niekoľkými ďalšími funkciami. Jednou z týchto ďalších funkcií je aj spracovaný obrazový výstup z RLCS (*processed*), ktorý ale funguje iba ak v RLCS beží *ImageServerAgent*.

Keď ľavú stranu prepneme na rozhranie RLCS, môžeme už pridávať a konfigurovať agenty alebo správania. Môžme použiť napríklad postup z experimentu 6.1. Prehľad všetkých agentov, ktoré sú dispozícii, je v nasledujúcom dodatku B. Keď dokončíme konfiguráciu, odošleme ju kliknutím na *Apply*.

Dodatok B

Stručný prehľad pripravených agentov

V tomto dodatku je iba stručný popis všetkých pripravených agentov pre systém RLCS. Podrobnejší popis (ale v angličtine), vrátane popisu parametrov, je v *JavaDoc* dokumentácii k systému RLCS. Rovnaký popis, vytiahnutý priamo z *Javadoc* dokumentácie, je k dispozícii aj priamo vo webovom rozhraní RLCS.

Agenty sú rozdelené do kategórií (balíkov), podľa toho čo robia. To trochu zaváňa porušením pravidla o dekompozícii podľa aktivity (viď sekciu 2.2). Toto však nie je dekompozícia kódu na moduly, ale iba rozdelenie jednotlivých modulov do skupín, čisto z dôvodu prehľadnosti pre používateľov. Tieto kategórie sú (v abecednom poradí):

beh Agenty implementujúce pohybové správania (angl. *behavior* - odtiaľ meno balíku). Sú to agenty vyššej úrovne, ich činnosť spravidla závisí na agentoch z iných balíkov (hlavne z balíku *robot*).

camera Agenty pracujúce s hornou kamerou a výstupom z nej.

image Agenty na prácu s obrázkami. Sú to hlavne grafické filtre, ale tiež agenty realizujúce čítanie a posielanie obrázkov.

os Agenty na prácu s objektmi operačného systému.

robot Agenty pre základné operácie s robotmi.

sim Agenty komunikujúce zo simulátorom. Tieto agenty sú spravidla alternatívami k iným agentom, pre prípad že pracujeme so simulátorom, a nie s reálnymi robotmi.

space Agenty realizujúce operácie s prostredím.

Nasleduje zoznam jednotlivých agentov (opäť v abecednom poradí). Meno agenta vždy začína menom balíku (kategórie), v ktorom sa agent nachádza.

beh.FollowAgent Prenasleduje jedným robotom druhého.

beh.ImitateAgent Napodobňuje pohyby jedného robota druhým.

beh.PathAgent Hýbe robotom po preddefinovanej trase.

beh.RoamAgent Náhodné potulovanie robota.

camera.ImageTransformAgent Aplikuje na obraz z kamery afínnu transformáciu vypočítanú podľa zdetekovaných referenčných bodov. Po transformácii zostane na obrázku iba samotný stôl, okraje sú orezané.

camera.ReadDetectionAgent Číta pozície zdetekovaných robotov zo súboru na disku, alebo cez HTTP protokol.

camera.ViewPosAgent Dokresľuje do obrázku pozície robotov a aktuálne cieľové body. Umožňuje vývojárovi sledovať funkčnosť detekcie.

image.ImageFilterAgent Základný agent pre všetky obrázkové filtre, implementuje spoločnú funkcionalitu. Dá sa použiť aj sám o sebe, na konverziu obrázku do iného farebného formátu.

image.ImageLoaderAgent Načítava premenlivý obrázok zo súboru do prostredia.

image.ImageServerAgent Vysiela stream obrázkov z prostredia klientom (webovému rozhraniu). Podporuje viacero klientov súčasne.

image.ImageStreamAgent Spúšťaný *ImageServerAgent*om pre každého pripojeného klienta.

image.ConvolutionAgent Filter aplikujúci na obrázok konvolúciu s preddefinovanou maticou.

image.GrayScaleAgent Filter prevádzajúci farebný obrázok na "stupne šedej".

image.InvertAgent Filter realizujúci farebnú inverzia obrázku.

image.SobelAgent Filter aplikujúci na obrázok Sobelov operátor. Používa sa na detekciu hrán v obrázku.

image.TransformAgent Filter aplikujúci na obrázok preddefinovanú afínnu transformáciu.

image.TresholdAgent Filter prevádzajúci obrázok na dvojfarebný čiernobiely, s nastaviteľným prahom.

os.ExecuterAgent Spúšťa externé programy.

os.FileLoaderAgent Načítava obsah daného súboru do prostredia.

os.FileToucherAgent Upravuje čas poslednej modifikácie súborov.

robot.ApproachAgent Hýbe robotom k cieľovému bodu.

robot.AvoidAgent Zabezpečuje obchádzanie prekážok. Kooperuje s *ApproachAgent*om.

robot.CollisionAgent Detekuje hroziace kolízie. Pre každého robota ovládaného RLCS je jeden potrebný.

robot.ConnectionAgent Agent starajúci sa o nadviazanie a udržiavanie spojenia s robotmi.

robot.IdentifyAgent Identifikuje konkrétneho robota v zozname pozícií robotov zdetekovaných z kamerového obrazu.

robot.MovementAgent Posiela robotovi pohybové príkazy.

robot.OdometryAgent Získava pozíciu robota na základe odometrie.

sim.ImageLoaderSimAgent Generuje simulovaný kamerový obrázok.

sim.SimDetectionAgent Získava pozície robotov zo simulátora.

space.CopierAgent Kopíruje bloky v prostredí.

Vo webovom rozhraní k systému RLCS sú k dispozícii aj tzv. správania, čo sú preddefinované skupiny agentov. Pripravené sú zatiaľ nasledovné správania:

Image and detection Načítanie obrazu z kamery, spracovanie obrazu, vysielanie spracovaného obrazu pre klientov, detekcia robotov.

Simulated image and detection Alternatíva k *Image and detection* ak pracujeme so simulovaným laboratóriom.

Robot common active Spojenie s robotom, detekcia robota, základné pohybové agenty (Approach, Avoid), detekcia kolízií, pre roboty ktorými systém môže hýbať.

Robot common passive Spojenie s robotom, detekcia robota, detekcia kolízií, pre roboty ktorými systém nemôže hýbať.

Dodatok C

Robot *Robotnačka*

V tomto dodatku popíšem roboty v našom robotickom laboratóriu, s ktorými systém RLCS pracuje. Ide o roboty typu *Robotnačka*, navrhnuté a vyrobené firmou Microstep–MIS.

Robotnačka je mobilný kresliaci robot, ovládaný bezdrôtovo z počítača. Primárne je určený na podporu výučby programovania v spojení s programovacími prostrediami, ktoré používajú tzv. korytnačiu grafiku (Imagine Logo, Comenius Logo). Robotnačka je však dobre použiteľná aj ako experimentálna platforma, a vyhovuje aj potrebám tejto práce.

Robotnačka je trojkolka s diferenčným pohonom dvoch kolies, a tretím podporným koliečkom. Obsahuje jednoduchý 8-bitový mikropočítač, ktorý riadi motory, a tiež prídavný hardvér. Základná výbava Robotnačky je pohyblivý (hore - dole) držiak na pero, a šesť IR senzorov po obvode, schopných detekovať okraj stolu, ale aj čiary alebo značky na stole. Ďalšia voliteľná výbava môže byť chápadlo s dvomi stupňami voľnosti, schopné zdvíhať a prenášať malé predmety, alebo bezdrôtová sieťová kamera. Komunikácia s externým počítačom prebieha pomocou Bluetooth modulu.

K Robotnačke je naprogramovaná softvérová knižnica, umožňujúce jej ovládanie z rôznych programovacích jazykov. Táto knižnica umožňuje ovládanie robota klasickými korytnačími príkazmi ako "vpred *x* krokov", "otočiť o *x* krokov", a podobne. Navyše podporuje tzv. rýchlostné príkazy, ktorými nastavíme rýchlosť otáčania kolies, pre každé koleso zvlášť. Knižnica tiež umožňuje ovládanie ďalšieho hardvéru (pera a chápadla), a získavanie stavu (IR senzory, batéria).

Robotnačky obsahujú nabíjateľnú batériu, ktorá napája všetok hardvér, vrátane príslušenstva. Na jedno nabitie Robotnačka vydrží fungovať asi dve hodiny. Robotnačka má zaintegrovanú aj nabíjačku na batériu, stačí jej pripojiť sa k zdroju napätia. Na stole v robotickom laboratóriu je pre každú robotnačku nabíjacie miesto, kam jej stačí prísť, jej nabíjačka sa pripojí, a batéria sa dobíja.

Podrobnejší popis Robotnačky vrátane detailnej technickej špecifikácie sa dá nájsť v [5]. Viac informácií o robotickom laboratóriu, vrátane ďalšieho popisu Robotnačky, je v [12].

Systém RLCS zatiaľ zďaleka nevyužíva všetky možnosti Robotnačky. Zatiaľ vôbec nepracuje s jej výbavou (pero, chápadlo, senzory), využíva iba jej schopnosť pohybovať sa. A aj pri pohyboch je RLCS nenáročný, vôbec nepotrebuje veľkú presnosť, ktorou Robotnačka disponuje. Znamená to, že systém by mal len s malými úpravami dokázať pracovať aj s inými typmi robotov.

Obrázok C.1: Robotnačka

Dodatok D

Simulátor

Podla pravidiel, ktoré zaviedol už R. Brooks (viď sekciu 2.2), je lepšie vyvíjať systém priamo s prototypom robota, a nie s nejakým simulátorom. Avšak z praktických dôvodov je často vhodné mať k dispozícii aj simulátor, pretože testovanie so simulátorom je spravidla oveľa jednoduchšie ako so skutočnými robotmi.

Ja som v rámci tejto práce naprogramoval simulátor robotického laboratória, ktorý je užitočný nielen pre testovanie počas vývoja systému RLCS. Je na počítači v laboratóriu stále spustený, a pri práci s laboratóriom cez internet sa namiesto skutočných robotov dá pracovať aj so simulovanými.

Vyvinutý simulátor simuluje skupinu robotnačiek na ploche. Softvér komunikujúci so skutočnými robotmi sa namiesto robotnačky pripojí k simulátoru, a komunikuje s ním úplne rovnako ako so skutočnou robotnačkou. Namiesto virtuálnej sériovej linky nad Bluetooth spojením, ktorá sa používa na komunikáciu s robotnačkou, sa so simulátorom komunikuje cez TCP spojenie. To je jediný rozdiel, takže v existujúcej knižnici pre komunikáciu s robotnačkou bola potrebná len jediná minimálna zmena, aby všetko fungovalo aj so simulátorom.

K simulátoru som naprogramoval aj webové rozhranie, ktoré vizualizuje aktuálnu situáciu v simulovanej aréne. Toto sa na webe zobrazuje namiesto obrazu z hornej kamery, ktorý tam je pri práci so skutočnými robotmi.

Simulátor podporuje pohybovanie robotmi, kreslenie perom, a snímanie čiar na zemi. Pri pohybe sa samozrejme detekujú kolízie medzi robotmi, takže v simulovanej aréne nevznikne žiadna "nemožná" situácia. Čo sa týka snímania čiar, tie sa dajú nastaviť cez webové rozhranie, poslaním obrázka, ktorý sa použije ako podklad pre virtuálnu plochu. Nemusia to byť samozrejme len čiary, ale akýkoľvek čierno-biely obrázok. Čiary nakreslené samotnými robotmi sa snímať nedajú, z jednoduchého dôvodu, pretože to nejde ani v skutočnom laboratóriu. Tam sú totiž nakreslené čiary príliš úzke na to, aby ich senzory robotov zaznamenali.

Simulátor má mnoho využití. Okrem samozrejmého uľahčenia testovania sa nájdu aj niektoré zaujímavejšie. Napríklad je možné ho použiť pre natrénovanie aplikácií, ktoré na ovládanie

robotov používajú učiace sa prostriedky umelej inteligencie (ako napr. neurónové siete alebo reinforcement learning). Na skutočné roboty sa potom nasadí už naučená aplikácia. Simulátor sa dá spustiť v zrýchlenom režime, čo znamená že roboty sa hýbu rýchlejšie ako v skutočnosti. To môže byť užitočné, napríklad ak treba spraviť nejaký dlho trvajúci experiment. Zrýchlenie je obmedzené iba výkonom procesora, na ktorom simulátor beží. Simulátor tiež veľmi oceníme, ak je skutočné laboratórium práve neprístupné. Aj to sa totiž občas stáva.

Literatúra

[1] Arkin R. - *Behavior-based Robotics*. The MIT Press, Cambridge, Mass., 1998

[2] Brooks R. - *A Robust Layered Control System for a Mobile Robot*. IEEE Journal of Robotics and Automation. RA-2, 1986b, pp. 14-23

[3] Brooks R. - *Intelligence without Reason*. Computers and Thought, IJCAI-91, 1991

[4] Brooks R. - *Cambrian Intelligence*. The MIT Press, Cambridge, Mass., 1999

[5] Ďurina D., Petrovič P., Balogh R. - *Robotnačka - The Drawing Robot*. Robtep 2006

[6] Eckel B. - *Thinking in Java, 4th Edition*. Prentice Hall, 2006

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

[8] Kelemen J. - *Strojovia a agenty.* Archa, Bratislava, 1994.

[9] Lúčny A. - *Tvorba inteligentných systémov na báze architektúry Agent-Space*. Dizertačná práca, FMFI UK, 2004

[10] Lúčny A. - *Od medzimodulových spojení k nepriamej komunikácii medzi agentami.* ZNALOSTI, Ostrava, 2007

[11] Petrovič P. - *Robotnačka pre Imagine Logo*. Didinfo, Banská Bystrica, 2005

[12] Petrovič P., Lúčny A., Balogh R., Ďurina D. - *Remotely-Accessible Robotics Laboratory*. Robtep, 2006

[13] Shoham Y. - *Agent-oriented programming*. Artificial Intelligence, 60(1):51-92, 1993

[14] http://en.wikipedia.org/wiki/Sobel

[15] http://en.wikipedia.org/wiki/Hough_transform

Abstract

Ronald Weiss: *Visual programming of control system for colony of robots*. Diploma thesis. Comenius University in Bratislava, Faculty of Mathematics, Physics and Informatics, Department of Applied Informatics. Advisor: RNDr. Andrej Lúčny PhD. Bratislava 2007.

This diploma thesis describes design and implementation of a programmable control system for colony of mobile robots. The system runs on a computer, which has direct access to the robots, and can be programmed through the internet using a web interface. It is a multi-agent system, whose programming consists of selecting and configuring agents, which should be launched. Programming with the web interface is visual, users do not have to learn any difficult programming language. They are able to define behavior of robots by just clicking with mouse, and filling parameter values. The multi-agent control system is based on unconventional software architecture Agent-Space, which uses indirect communication between agents. This gives the system some interesting attributes, differentiating it from other similar systems.

Keywords: mobile robots, control system, visual programming, Agent-Space