Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky
Katedra aplikovanej informatiky

Andrej Lúčny

Tvorba inteligentných systémov na báze architektúry Agent-Space

Dizertačná práca

Školiteľ: prof. RNDr. Jozef Kelemen, DrSc.

Bratislava **2005**

Ďakujem týmto všetkým ktorí ma povzbudzovali a podporovali v úsilí vytvoriť raz túto prácu, menovite prof. RNDr. Jozefovi Kelemenovi, DrSc., doc. PhDr. Jánovi Šefránkovi, CSc., Dr. Mikulášovi Popperovi a prof. Ing. Vladimírovi Kvasničkovi, DrSc. Ďakujem taktiež mojim kolegom z firmy MicroStep-MIS, menovite Ing. Jozefovi Omelkovi, vďaka ktorému má moje pracovné prostredie úroveň potrebnú na prenos technológií z výskumu do praxe. Ďalej ďakujem mojim blízkym, ktorí mi umožnili venovať tejto práci môj čas, ktorý patril im.

Prehlasujem na svoju česť, že predkladanú dizertačnú prácu som vypracoval samostatne a všetku použitú literatúru uvádzam v zozname

Andrej Lúčny

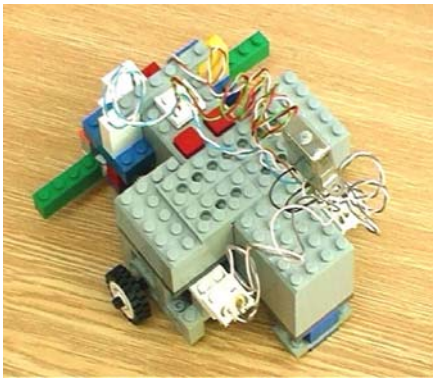
Obsah

Zoznam obrázkov

Použité skratky

ACM	Association for Computer Machinary
ACL	Agent Communication Language
AOP	Agent Oriented Programming
ASP	Action Selection Problem
BBS	Behavior-Based Systems
COOP	Concurrent Object-Oriented Programming
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
FIPA	Foundation for Intelligent Physical Agents
GOFAI	Good Old-Fashioned Artificial Intelligence
IIOP	Internet Inter ORB Protocol
IPC	Inter Process Communication
JDK	Java Development Kit
JMF	Java Media Framework
JRE	Java Runtime Engine
JRM	Java Reflection Model
JVM	Java Virtual Machine
KQML	Knowledge Query Manipulation Language
LAN	Local Area Network
LINDA	Linda Is Not aDA
MAS	Multi-Agent Systems
NAI	„New“ Artificial Intelligence
ODE	Open Dynamics Engine
OOP	Object Oriented Programming
ORB	Object Request Broker
OS	Operating System
POA	Portable Object Adaptor
PVM	Parallel Virtual Machine
ppp	Point to Point Protocol
QNX	Quick uNiX
QSSL	QNX Software Systems Limited
RAD	Rapid Application Development
RMI	Remote Method Invocation
RTOS	Real Time Operating System
SOAP	Simple Object Access Protocol
SRR	Send Receive Reply model
TCP/IP	Transport Communication Protocol / Internet Protocol
VM	Virtual Machine
VRML	Virtual Reality Modelling Language
WAN	Wide Area Network
WS	Web Services
XHTML	eXtensible HyperText Markup Language
XML	eXtensible Markup Language
XSD	XML Schema Definition
XSLT	eXtensible Stylesheet Language Transformation

Prehľad projektov vykonaných v súvislosti s touto v prácou

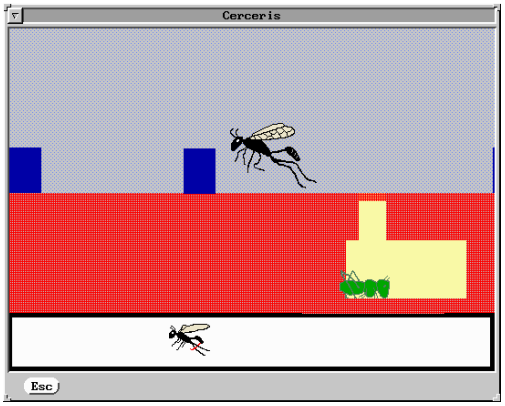


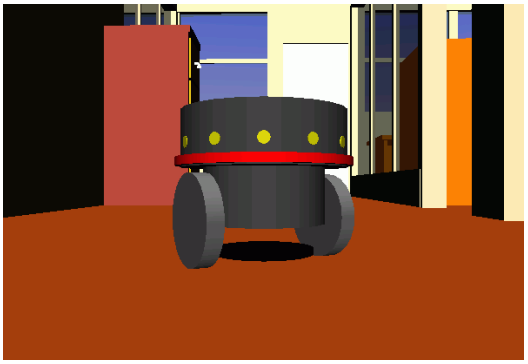
Tutbot

Ukážka primitívneho robota, ktorý sa pohybuje popri stene v uzavretej scéne. Ide o reimplementáciu robota Tutbot od Anity Flynnovej. Robot slúži na demonštráciu významu prostredia: hoci riadiaci systém je jednoduchý, deterministický a zreteľne pochopiteľný, pohyb robota je nepredvídateľný. HW: LEGO technik + relé, SW: žiadny.

Cerceris

Simulátor zakladania potomstva kutavky rodu cerceris. Ide o simulovanie známeho zlyhania v správaní kutavky a jeho hypotetické vysvetlenie na základe multiagentovej povahy riadiaceho systému, ktorý zdanlivo vykonáva sekvenciu krokov. HW: PC, SW: QNX4, MsAgent, C.



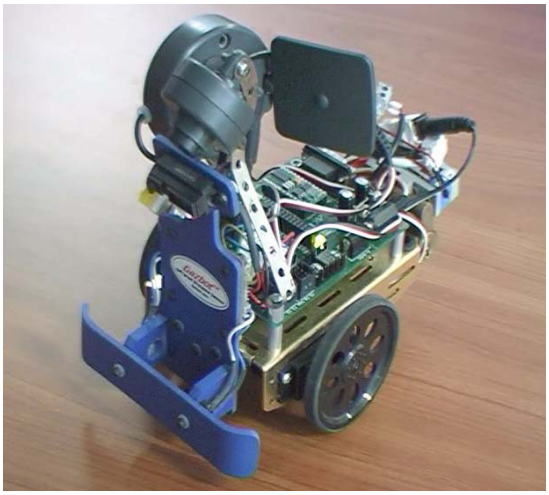


Allen

Reimplementácia robota Allen vo VRML. Príklad robota, originálne navrhnutého v sumbsumpčnej architektúre, realizovaného v architektúre Agent-Space

UDCS/QNET

Unified Data Collection System for Q-network. Komerčný systém postavený na architektúre Agent-Space. Jeho úlohou je zber dát z hydrometeorologických staníc rôznych druhov. Jeho správanie zahrňuje koordináciu homogénnych podsystémov na báze multiagentového systému. Pracuje za neurčitých podmienok. Niektoré jeho črty sa dajú považovať za prejavy brikoláže či emergencie. HW: industriálne PC. SW: QNX4, MsAgent, C. Okrem autora práce, ktorý projekt viedol, sa na ňom podieľali ďalší vývojári z MicroStep-MIS, najmä Martin Gažák a Ľubomír Drobný. K systému bol vyvinutý aj jeho simulátor vo VRML pre demonštračné účely.



PingPong

Robot naháňajúci pingpongovú loptičku. Radiaci systém je realizovaný na PC, ktoré má s robotom bezdrôtové spojenie. Je postavený na architektúre Agent-Space. HW: podvozok Boe-bot (Paralax) a kamera XCam2. SW: Java SE. Okrem autora sa na projekte podieľali: Richard Balogh, Pavol Boško, Ľubomír Lukáčik z FEI STU a Pavel Petrovič z FMFI UK.

Kamko

Lego-robot s bezdrôtovou kamerou ustupujúci pred detekovaným pohybom. Kombinuje autonómnu časť s riadiacim systémom na PC. HW: Lego Mindstorm a kamera XCam2. SW: Java SE + Lejos. Cieľom projektu bolo zvládnuť spracovanie obrazu. (Na snímke nevidno Lego tower, umiestnený nad scénou.)

I. Úvod

Cieľom tejto práce je prispieť k poznaniu, prečo súčasné umelé systémy založené na počítačoch trpia v porovnaní so živými systémami tak očividným nedostatkom schopností, ktoré obvykle začleňujeme pod pojem inteligencia. Čo im vlastne chýba, že sa ich výkony stávajú tak často a oprávnene predmetom vtipov?

Na túto otázku sa obyčajne odpovedá, že našim počítačom chýba obrovské množstvo poznatkov, ktoré sú súčasťou tzv. zdravého rozumu a schopnosť s nimi manipulovať. Prirodzeným dôsledkom tohto názoru je, že existuje niekoľko zaujímavých prác, ktoré podávajú pomerne rozsiahly rozsah mechanizmov, ktoré by mali byť zastúpené v inteligentom umelom systéme. Najznámejšou z nich je Society of Mind od Marvina Minského. Avšak i tomuto vedcovi svetového formátu, ktorý v umelej inteligencii nemá sebe rovného, sa podarilo úplne ma šokovať, keď na nedávnej diskusii v Prahe[1] vyhlásil - ako reakciu na otázku istého horlivého obdivovateľa - že „Society of Mind neimplementoval pre nedostatok peňazí a ľudí“. Teda, že v princípe ju považuje za priamo implementovateľnú súčasnou technológiou.

Avšak keď máme zoznam desiatok štruktúr a mechanizmov, ktoré sme na základe určitých pozorovaní schopní rozpoznať v mysli a o ktorých máme určitú predstavu, že by pomohli prekonať neschopnosti súčasných umelých systémov, to ešte neznamená, že sme ich schopní integrovať do jedného celku. Naopak, toto je nesmierne ťažký problém, na ktorý poukazuje práve Minsky, keď zastáva názor, že cieľom umelej inteligencie nemôže byť hľadať najlepší spôsob reprezentácie poznatkov, ale spôsob, ako skĺbiť rôzne reprezentácie poznatkov do jedného systému.

Naším cieľom teda bude nájsť cestu ako naprogramovať do jedného systému mnoho rôznorodých požiadaviek tak, aby spolu ladili. Na prvý pohľad triviálna vec, ale keď je „mnoho“ naozaj mnoho, je to súčasnými technológiami neprekonateľný problém[2]. Kľúčovými atribútmi riešenia tohto problému v rámci tejto práce bude decentralizovaný charakter modularity systému a možnosť, aby mohlo byť volanie jedného modulu druhým ovplyvnené modulom tretím. Hoci význam týchto prvkov bol rozpoznaný pred takmer dvadsiatimi rokmi, sú pre súčasné metódy programovania úplne netypické až neznáme. Našou úlohou bude vyniesť ich z ústrania mobilnej robotiky, v ktorej vznikli, do postavenia paradigmy programovania ako takého.

O tomto kroku v žiadnom prípade netvrdíme, že je oným kameňom mudrcov, ktorý zavŕši približovanie umelých systémov k živým. Pre potreby tejto práce vychádzame z predstavy, že kvality počítačov sú od živých systémov nepredstaviteľne vzdialené a že počítače sú len ich najlepšou súčasnou metaforou. Zaujímavým aspektom nášho snaženia je, že túto snahu, ktorú budeme vyvíjať, príroda vôbec vyvinúť nemusela. Schopnosť integrovať rôznorodú či rovnorodú „mnohosť“ do celku je totiž jej implicitnou vlastnosťou. Tento rozdiel naznačuje, že **prírodu nebudeme opakovať, ale len simulovať** v počítači a preto je to práve tak ťažké, lebo robíme vlastne niečo neprirodzené. Pôjde tu teda skôr o tzv. slabú UI. Netreba dodávať, že pod inteligenciou budeme rozumieť širokú škálu procesov, ktoré sa podieľajú na schopnosti živých systémov účelne sa správať v ich prostredí, pričom sa budeme zameriavať hlavne na „nevedomé“ procesy. Držíme sa teda dosť pri zemi.

Na druhej strane však spomínaný krok považujeme za nevyhnutný. Navyše za ten prvý, ktorý treba urobiť. Pokladáme ho však za tak zložitý, že nebudeme k nemu pridávať ani jasne nevyhnutú vlastnosť inteligentných systémov ako je adaptívnosť. V súvislosti s ňou skúmame v tejto práci vlastne štruktúry, ktoré sú hodné toho aby boli adaptované. (Špeciálne v prípade adaptívnosti realizovanej evolučnými algoritmami by šlo o mechanizmus odvodenia fenotypu z genotypu.) Domnievame sa totiž, že za obmedzenými možnosťami súčasných adaptívnych

prístupov sa skrýva predovšetkým nevhodnosť štruktúr, ktoré sú v ich rámci používané na generovanie správania systému z jeho zápisu (genotypu).

S prezentovanou technológiou sa teda nedostaneme príliš ďaleko, zato však nebudeme predbiehať. Nadviažeme na existujúce trendy v súčasnom programovaní, hoci nie na hlavné prúdy, ale určité špecifické smery. Stavať budeme hlavne na tých, ktoré aspoň čiastočne atakujú problém interaktívnosti počítačových systémov.

Interaktívnosť je totiž práve tá bázová vlastnosť, ktorej absencia znemožňuje počítačom inteligentné správanie z princípu. Rozumieme pod ňou nazeranie na počítač ako na systém, ktorý nepretržite beží, nepretržite dostáva podnety z okolia (vrátane užívateľa) a nepretržite produkuje určité výstupy. Hoci hardwarové rozhranie počítača tento charakter má, pre súčasné operačné systémy je charakteristický opak, t.j. rezultatívnosť. Ich interaktívna zložka slúži vlastne len ako rozhranie na púšťanie programov, ktoré produkujú nejaký výsledok a skončia. Za takých podmienok je ale jasné, že počítač nebude vedieť vykonať požiadavku „robme teraz to, čo včera“, lebo v ňom nie je nič, čo by udržovalo potrebnú kontinuitu.

Tento defekt je do značnej miery daný súčasnými hardwarovými možnosťami a to hlavne delením dát na tie, ktoré sú uložené v pamäti (dočasné) a na tie, ktoré sú uložené na disku (trvalé). V skutku mnoho zo súčasného programovania je o tom, ako prehadzovať dáta medzi pamäťou a diskom, čo je dosť nešťastné mrhanie síl. Budeme mať preto na pamäti také riešenia, ktoré aspoň ideovo používajú jediné, veľké a trvalé dátové médium.

S interaktívnosťou je silne spojená ďalšia vlastnosť predstavujúca posun v tradičnom zmýšľaní – a to behaviorálnosť. Na systémy budeme nazerať tým spôsobom, že od nich očakávame určité správanie, že to čo od nich chceme, nie je výstup, ale nekonečná postupnosť výstupov či odoziev na meniaci sa vstup, ktorej je vlastný určitý charakter.

No a tretia základná vlastnosť nami uvažovaných systémov bude ich komplexnosť. Týmto termínom budeme vyjadrovať, že požiadavky obsahujú určitú „mnohosť“, ktorá bráni aby boli realizované triviálnym spôsobom. Komplexnosť budeme chápať ako základný limit, ktorý znemožňuje vytvoriť inteligentný systém určitej povahy tradičným spôsobom. V princípe táto „mnohosť“ môže mať dva dôvody: pribúdajúce či meniace sa požiadavky na jednej strane a neznáme či nejasné požiadavky na strane druhej.

Z hľadiska aplikácii budeme mať na pamäti tie systémy, ktoré sa vyznačujú potrebou interaktívnosti, behaviorálnosti a komplexnosti: mobilné roboty, monitorovacie systémy, riadiace systémy, simulátory živých systémov a podobne. Tým však nechceme povedať, že by sa aplikovateľnosť nášho prístupu týkala iba týchto oblastí. Naopak, prínos spočíva v tom, že náš prístup je úplne všeobecný.

Výsledkom našej práce bude všeobecná programátorská technológia, ktorou – okrem iného - budeme schopní dostať do počítača určité vlastnosti živých systémov, ktoré nie sú pri použití tradičných prístupov bežné.

História a členenie práce

Po dlhých úvahách, či pri písaní prác uprednostniť metodologický alebo teoretický výklad, rozhodol som sa vzhľadom na väčšiu úspornosť textu pre druhú možnosť. Členenie práce teda nebude zodpovedať postupu akým sa k prezentovanému riešeniu prišlo. Preto najprv pár slov o histórii tejto práce.

Keď som v roku 1994 začínal štúdium PhD, plánovali sme s mojím školiteľom pokračovanie mojej diplomovej práce zameranej na emergenciu v kolónii homogénnych agentov, ktoré medzi sebou priamo nekomunikovali, nanajvýš mohli používať „chemotaxiu“ (naše agenty boli skutočne aj vyobrazené v našom simulátore ako loziace mravce, pre ktoré je

tento spôsob komunikácie typický). Mali sme však ambíciu zaťať do niečo užitočného, doslova sme chceli implementovať určité postupy spojené s myslením - napríklad prehľadávanie problémového priestoru - ako lozenie mravcov (dnes je to obľúbený slogan odvetvia „ant systems“, ale vtedy to bolo ponímané skôr ako bláznivý nápad). Rok 1994 sa niesol v znamení veľkého „boomu“ multiagentových systémov, čo reprezentovalo hlavne júlové číslo Communications of ACM. V ňom sa v sérii článkov predstavili takmer všetky dnešné prúdy a aplikačné oblasti prezentujúc v dobrej viere, že čosi spoločné ich spája a zdalo sa, že v krátkom čase dôjde k zjednoteniu, vyčisteniu pojmov a multiagentové systémy zaujmú ústredné miesto vo vývoji informatiky a umelej inteligencie. Nestalo sa, multiagentové systémy ostali nesúrodou zmesou najrozličnejších aplikácii pomerne voľne chápanej „agentovej“ metafory. Dokonca niektoré kľúčové osoby diskutujúce v danom čísle o agentoch a ich skvelej budúcnosti, pojem „agent“ prestali nadobro používať (napr. Marvin Minsky). V tomto čísle bolo ešte dosť článkov ovplyvnených Rodney Brooksom (Pattie Maes, dokonca Bart Selman - neskorší člen tzv. americkej školy na čele s Katiou Sycara), dnes však prevláda iný prúd, ktorý v danom čísle reprezentoval Michael Genesereth a dnes hlavne Michael Wooldridge a Nicholas Jennings. Stalo sa tak vďaka iniciatíve FIPA97. Ja som akosi dodnes ostal verný prúdu, ktorý je ovplyvnený Brooksom a v súčasnosti ho aspoň čiastočne reprezentujú Jacques Ferber a Yves Demazieu. Preto v práci budem používať multiagentovú terminológiu, hoci si uvedomujem, že tým len prispievam k chaosu v danej oblasti. Tento chaos sa však v žiadnom prípade do práce nepremieta, nakoľko sa nesnažím ani trochu aby môj spôsob použitia bol akceptovateľný čo najširším spektrom prúdov, ktoré sa v oblasti vyskytujú. Jedine takto dokážem pre moju prácu zachovať logiku. Minského riešenie použiť inú terminológiu by mi pripadlo bolestné, lebo metafora „agentov“ mi perfektne vyhovuje.

Ako prvý krok som si vo svojej práci vytýčil odpútať sa od veľmi limitujúcej tzv. trojvrstvovej architektúry, kde sa z modulov skladali agenty a z agentov kolónia. Táto bola dobrá tak na simulovanie mravcov behajúcich po stole, ale vystihnúť ňou niečo zložitejšie bolo prakticky nemožné. Pravdu povediac, tento problém by som asi nikdy nevyriešil – niekoľko krát som si bol istý, že to mám a následne som zistil, že nie – keby ho za mňa nebol vyriešil môj školiteľ pomerne jednoduchou radou zaviesť agenty, ktoré by mali dvojaké receptory a efektory – jedny pôsobiace na vyššej úrovni a druhé pôsobiace na nižšej úrovni opisu sveta. Detaily tohto riešenia prediskutujeme neskôr, tu spomeňme len to, že toto riešenie umožňuje hierarchiu opisu s ľubovoľným počtom úrovní, nakoľko premieňa hierarchiu na zapuzdrenie. Zdanlivou nevýhodou je, že priamu komunikáciu opodstatňuje len v rámci rovnakej úrovne hierarchie. Postrčený touto vlastnosťou som sa začal orientovať na výhradné používanie nepriamej komunikácie až som získal určitú konkrétnu predstavu o alternatívnej architektúre.

V roku 1996 som sa v rámci mojej komerčnej obživy v MicroStep-HDO zúčastňoval na riešení problému, ako eliminovať niektoré nepríjemné vlastnosti tzv. pyramidálnej client-server architektúry. Šlo o to, že sme vyvíjali systémy pracujúce v reálnom čase (a 365 dní v roku po 24 hodín) pod operačným systémom QNX4, ktorý bol nesmierne spoľahlivý a dobre podporujúci reálny čas (latencia 2μs), takže pri správnom aplikačnom softwari, by aj výsledné riešenie malo tieto vlastnosti. Architektúra aplikačného softwaru, ktorú odporúčalo QSSL (výrobca QNX4) však spôsobovala pri zložitejších systémoch vážne problémy. Kolegovia Róbert Štefanec a Tomáš Hrmo vtedy pre riadiaci systém gumárenského mixéra navrhli inú architektúru, ktorá mi nesmierne pripomenula moju mnoho-hierarchickú multiagentovú architektúru. Zároveň ma priviedla na myšlienku, že agenty by bolo oveľa vhodnejšie definovať ako procesy so špecifickým tvarom kódu, ktorý zachytáva volanie komunikačných primitív v rámci určitého modelu posielania správ, než ako proces, ktorý má 10-20 vlastností (ako je to doteraz bežné). Návrh mojich kolegov som teda odobril, zároveň som však začal pracovať na vlastnom riešení, ktoré by bolo postavené na multiagentovej terminológii. Bolo podstatne čistejšie a podarilo sa mi presvedčiť vedenie, aby sme touto architektúrou

preprogramovali náš najrozsiahlejší systém (Integrated Meteorological System). Týmto krokom sa výrazne zlepšili vlastnosti systému ako zotaviteľnosť z chýb, práca v reálnom čase, škálovateľnosť, konfigurovateľnosť a modifikovateľnosť, čo sa nemalou mierou prejavilo na jeho komerčnom úspechu (dnes okolo 200 inštalácii v šiestich krajinách). S touto architektúrou sme implementovali aj ďalšie systémy (a mnohé modifikácie pôvodných) a vychytali sme ešte zopár detailov. Podstatné myšlienky tohto prístupu som zhrnul v článku na konferenciu doktorandov na Ekonomickej univerzite v roku 1998.

Vybavený týmto implementačným prostriedkom som obrátil pozornosť na vzťah tejto architektúry a umelej inteligencie (či umelého života[3]). Týmto sa zaoberám dodnes. Už v návrhu dizertačnej práce v roku 1997 som ukázal, že s drobnými odchýlkami ňou možno vyjadriť systém implementovaný Brooksovou subsumpčnou architektúrou. Tým pochopiteľne vyvstala otázka, či ňou možno vyjadriť aj niečo čo sa subsumpčnou architektúrou vyjadruje len ťažko, alebo sa to vôbec nedá. Hoci, aj keby sa nič také nenašlo, za nezanedbateľný prínos by som pokladal, že moje vyjadrenie nie je úzko previazané s aplikačnou oblasťou mobilnej robotiky, ale predstavuje všeobecný programátorský prostriedok. V skutku, čo sa týka nastoľovania kooperácie medzi agentami, nenašiel som nič takého (ani nič také podľa mňa neexistuje). Avšak, zatiaľ čo modelovanie konkurencie medzi agentami sa v subsumpčnej architektúre prakticky nedá vyjadriť, v našej to ide ľahko. Pravda, keď tvoríme umelý systém, zdá sa šialenstvom, aby sme do neho vkladali prvky čo potierajú kooperáciu medzi jeho časťami. Preto som svoju pozornosť upriamil na modelovanie živých systémov, ktoré vykazujú vo svojom správaní určité „inteligenčné“ zlyhania a snažil sa ich modelovať konkurenciou agentov v rámci mojej architektúry. V rigoróznej práci v roku 2001 som to skúšal z hmyzom a neskôr som skúsil siahnuť i na ľudskú myseľ vychádzajúc z prác Marvina Minského. Z týchto pokusov mi vyplynulo, že by ani nebolo takým šialenstvom použiť konkurenciu v umelom systéme, pokiaľ by riešenie ňou dosiahnuté – hoci nefungujúce vždy a všade – bolo oveľa jednoduchšie nakódovateľné. Hoci príroda uprednostňuje logiku, nemusí to byť preto, že pracuje iba s logicky správnymi štruktúrami, ale preto, že najlepší výkon podáva riešenie, ktoré logiku – viac-menej iba náhodou – má. Aj tak je však logika iba jeden z vonkajších parametrov zvažovaných pri selekcii a preto môže byť prevážená inými parametrami, napríklad jednoduchosťou riešenia. Tento princíp sme následnej použili aj v reálnych aplikáciách.

To, že je moja architektúra univerzálnym programovacím prostriedkom som si overil – mimo iné – pri spomínaných simuláciách, keď som narazil na potrebu dokonalejších simulátorov, napr. pohybu robota či simulovaného hmyzu a implementoval som ich touto technikou.

Veľa energie som taktiež vynaložil na sledovanie podobných prístupov, ktoré sa objavili v minulosti a postupne sa taktiež objavovali v priebehu práce. Aj pod vplyvom týchto prác som sa rozhodol svoju architektúru pomenovať v angličtine nakoniec „agent-space“, hoci v slovenskom preklade by som radšej ostal pri pôvodnom názve „agent-prostredie“.

Toľko z histórie a teraz k samotnému členeniu práce. Po „povinnej“ kapitole venovanej prehľadu prác o ktoré sa opierame, uvedieme spomínanú architektúru agent-space a implementačne ju ukotvíme do existujúcich rámcov architektúr klient-server na báze komunikácie medzi procesmi, objektového a sieťového programovania. V ďalšej kapitole predstavíme prirodzené vlastnosti tejto architektúry s dôrazom na ich využitie pre tvorbu aplikácií reálneho času. Pokračujeme skúmaním vzťahu architektúry s inkrementálnym vývojom a ukazujeme, že ňou možno vyjadriť akékoľvek riešenie dosiahnuté subsumpčnou architektúrou. Vychádzajúc zo známej kritiky subsumpčnej architektúry poukazujeme na zaujímavý význam tzv. čistej reaktivity. V ďalšej kapitole zhŕňame poznatky o možnostiach modelovania pomocou našej architektúry a predstavujeme určité odvážne hypotézy o povahe živých systémov a inteligentných systémov vôbec. V závere porovnávame naše riešenie s podobnými prístupmi a uvádzame, v čom vidíme prednosti nášho riešenia.

V priebehu celej práce sa snažíme o priebežné demonštrovanie na konkrétnych príkladoch. Ich funkčné podoby sú dostupné na webovej stránke www.microstep-mis.com/~andy/pub.htm, kde sú taktiež všetky publikácie, ktoré sa na jednotlivé projekty špecificky zameriavali.

Na záver úvodu, malá jazyková poznámka. Veľa pojmov v práci používaných nemá ustálený slovenský ekvivalent. Väčšinou sa snažím také slová preložiť, ale pri tých, kde by mohlo dôjsť následkom prekladu k nepochopeniu, uvádzam radšej anglický pojem v úvodzovkách. Pri opakovanom používaní ho už potom uvádzam poslovenčené. Anglické názvy konkrétnych produktov a technológií ako i zaužívané skratky (LAN, WAN, IPC, OOP, ...) neprekladám. Najväčším orieškom je skloňovanie najčastejšie používaného pojmu „agent“. Prikláňam sa tu k stratégii zavedenej mojím školiteľom, ktorá vychádza zo skloňovania slova „robot“ – teda čiastočne životne, čiastočne neživotne, hoci to nie je v súlade so súčasnými pravidlami spisovného jazyka, ktorý pozná len životnú formu tohto slova.

[1] Spomínaná verejná diskusia sa konala 25.6.2004 na ČVUT

[2] Toto všeobecne uznávané pozorovanie sa (zatiaľ) neopiera o žiadne teoreticky podložené vysvetlenie.

[3] Ťažko sa rozhodnúť, ktorý z týchto dvoch pojmov používať v súvislosti s touto prácou. Na jednej strane používame jednoznačne syntetickú metódu, čo je typické skôr pre umelý život, na druhej strane neuvažujeme nič súvisiace s reprodukciou. Túto taxonomickú hádanku preto prenechávame povolanejším.

II. Predchádzajúce práce

Túto kapitolu môže čitateľ oboznámený s problematikou v princípe aj preskočiť, lebo vo zvyšku sa snažíme podať skoro samonosný výklad. Nebolo by však rozumné objavovať objavené a preto staviame na pomerne širokom spektre prác iných autorov. Títo si zaslúžia byť menovaní, navyše – špeciálne v tejto práci – musíme čo najpresnejšie vymedziť, čo je prevzaté a čo je pôvodné. Vychádzame z nasledovných oblastí vedy a techniky:

Multiagentové systémy

Multiagentové systémy (MAS) sú dnes pomerne rozsiahla a nesúrodá oblasť, ktorá sa zaoberá najrôznejšími aplikáciami „agentovej" metafory. Pri najširšom výklade tejto metafory je „agent" zástupca. Už tu však dochádza k štiepeniu. Na jednej strane si na zástupcovi môžeme ceniť, že za nás vykonáva to, čo my potom vykonávať nemusíme, teda autonómnosť. Na druhej strane môžeme oceňovať, že to vykonáva tam, kam kde my nemusíme ísť, teda mobilitu. Tieto „mobilné agenty" sa často považujú za špeciálnu časť multiagentových systémov, avšak so zvyškom majú len veľmi málo spoločné. Väčšinou ich možno považovať proste za mobilné kódy, ktorých význam spočíva najmä v tom, že niekedy je objemovo výhodnejšie poslať kód, ktorý realizuje istú službu za dátami, než tieto dáta za kódom. To, čo je pre mobilné a autonómne agenty spoločné, spočíva v tom, že autonómnosť z pochopiteľných dôvodov podporuje mobilitu. Každopádne, v rámci našej práce pod pojmom agent budeme vždy myslieť autonómny agent a mobilitu nebudeme uvažovať.

Ani takéto ponímanie však nestačí na pojmové vyčistenie. Autonómnosť totiž možno poňať jednak ako doplnkový prostriedok, ktorý obohacuje klasické systémy a umožňuje z nich vytvárať sieťové komplexy v rámci internetu, jednak ako základný prostriedok, ktorý predstavuje ústredný princíp tvorby software. Keďže to prvé poňatie je spoločensky naliehavejšie a komerčne zaujímavejšie, tvorí v súčasnosti hlavný prúd. Najvýznamnejšou iniciatívou tohto smeru bolo vydanie normy FIPA97, dnes sa prežívuje hlavne ako časť veľmi populárnej technológie Web Services.

My sa v práci zameriavame na druhé ponímanie, teda chápeme autonómne agenty ako základný tvorivý princíp softwaru. Pre takéto nazeranie na vec bola snaha rezervovať termín „agentovo orientované programovanie", ale tento „buzzword" sa ukázal byť príliš chytľavý a dnes jeho význam zmiešaný, možno sa však niekedy ešte očistí.

Našťastie oba tieto smery chápu (komunikačnú) povahu interakcie agenta s okolím rovnako a preto veľa problémov a spôsobov ich riešenia sa zhoduje. Rozdielna je len škála na ktorej sa s nimi pracuje. V prvom prípade je to spolupráca systémov na internete (WAN programovanie), v druhom prípade je to spolupráca procesov v rámci jedného uzla či lokálnej siete („concurrent" či LAN programovanie). Pokiaľ sa aj objavia rozdiely, sú dané práve fyzickými rozdielmi medzi týmito komunikačnými médiami, ktoré sú známe ako Deutschove klamy, napríklad nespoľahlivosťou WAN.

Ďalšie štiepenie je však spôsobené názormi na to, čo má byť vo vnútri agenta, teda čím sa dosahuje jeho „inteligencia". Tu opäť existujú dve vetvy. Prvá, tzv. deliberatívna vetva, nadväzuje na existujúce technológie, ktorá chápu agent ako obálku tradičných technológii umelej inteligencie založených na reprezentácii poznatkov pomocou logických formúl a ich manipulácii na základe logického odvodzovania. Naopak, tzv. reaktívna vetva vychádza z predstavy, že vnútro agenta by nemalo predstavovať nejakú špeciálnu súčiastku, ale že má „inteligenciu" získavať len pomocou vhodnej kooperácie s inými agentami. Dominujúcou

v oblasti vedy i techniky je v súčasnosti jednoznačne deliberatívna vetva, reaktívna viac-menej živorí. Avšak pokiaľ máme ambíciu skúmať ako inteligencia vzniká z elementárnejších schopností, pokiaľ skúmame ako zo systémov, ktoré sú schopné niečo konať, plynulo odvodiť systémy schopné konať „rozumne“, nič iné nám neostane, ako sa prikloniť k reaktívnej vetve. Na otázku „ako zostrojiť inteligentný systém“ teda nebudeme hľadať odpoveď typu „použite pri jeho stavbe túto inteligentnú súčiastku“, ale odpoveď typu „pospájajte obyčajné súčiastky týmto inteligentným spôsobom“. Reaktívne agenty sú často zaznávané z toho dôvodu, že nedochádza k uvedomeniu si toho, že hoci tieto stavebné jednotky sú veľmi primitívne a len veľmi málo možno nimi osve urobiť, pri správnom prepojení takýchto jednotiek možno realizovať viac-menej čokoľvek [Minsky 1986].

Pokiaľ sa MAS zameriavajú len na komunikačnú zložku agenta, hovorí sa o tzv. softwarových agentoch, zatiaľ čo keď sa uvažuje jeho vnútorná štruktúra na báze manipulácie s poznatkami, hovorí sa o tzv. inteligentných agentoch. Tieto pomerne nevhodné názvy sú viazané predovšetkým na rozdielne komunity, ktoré z týchto predpokladov vychádzajú – softwarových inžinierov a (distribuovaných) umelých inteligentov. Z pohľadu tohto členenia sú vzhľadom na jednoduchosť svojho vnútra agenty v tejto práci skôr softwarové. Napriek tomu však o nich často budeme hovoriť, že majú cieľ, čo nie je stopercentne kompatibilné so zaužívaným názvoslovím. Obyčajne sa pod týmto slovom rozumie explicitný cieľ, teda nejaké dáta vo vnútri agenta, ktoré vyjadrujú o čo sa má snažiť a používa sa len v súvislosti s inteligentnými agentami. Takýto cieľ mať naše agenty nebudú. Budú však mať implicitný cieľ, teda bude sa im dať priradiť špecifická činnosť ktorú realizujú. Slovom cieľ teda vyjadríme napríklad rozdiel medzi neurónovou sieťou a multiagentovým systémom: neurón nemá ani implicitný cieľ, zatiaľ čo aj najjednoduchší agent ho má. Inak povedané agent nie je konekcionistická jednotka.

Z chaosu, ktorý v MAS panuje, ponúkame nasledovné východisko: chápať multiagentové systémy ako metaforu, ktorá má veľa rôznych aplikácii. Za základ tejto metafory navrhujeme brať špecifický spôsob akým možno v počítači reprezentovať entitu reálneho sveta, t.j. spôsob akým ju tam možno preniesť. Od vzniku programovania sa vyskytli len tri takéto spôsoby (a možno ich už nebude viac), pričom ich rozlišovacím znakom je forma aktivity, ktorá sa im prisudzuje (viď Obrázok č. 1).

Obrázok č. 1 Rôzne typy aktivity

Typickým predstaviteľom pasívnej entity je stena, ktorá nás nemôže zraniť, len my sa na nej môžeme zraniť. Na rozdiel od toho hrable sú reaktívne: dokážu nás zraniť, len na ne musíme najprv stúpiť. Proaktívna entita ako je pes nás však dokáže zraniť bez toho, že by sme to nejako iniciovali.

Historicky najstarší spôsob prenáša do počítača pasívnu entitu, ktorej bol prisúdený názov záznam (record) alebo štruktúra (struct), od čoho nazývame tento spôsob štruktúrovaným programovaním. So záznamami môžeme manipulovať, ale nemôžeme od nich nič očakávať.

Po ňom sa na výslnie dostal spôsob, ktorý na reprezentáciu používa reaktívnu entitu, ktorú nazývame objekt. Táto so sebou nesie kód, ktorý môže byť zdrojom aktivity, pokiaľ je na

to externe vygenerovaný podnet. Ak zavoláme metódu objektu, niečo urobí, inak je pasívny. Používanie takejto reprezentácie nazývame objektovo-orientovaným programovaním.

Tretí spôsob, ktorý sa objavuje v poslednom desaťročí, ale ešte neprevládol (a možno ani neprevládne) je založený na proaktívnej entite. Túto nazývame agentom. Výnimočnosťou agenta je jeho stála aktivita, bez ohľadu na to, či ho niekto volá, alebo nie. Práve tomuto spôsobu by malo byť priradené ono spomínané „buzzword" agentovo-orientované programovanie.

Agent je vždy vybavený vlastným aktívnym vláknom, v ktorom musí zisťovať stav svojho prostredia a na základe toho v ňom realizovať určité zmeny. Tieto dve činnosti musia byť premostené určitou voľbou, a celé to nesmie nikdy prestať. Túto voľbu možno realizovať mnohými spôsobmi a vôbec to nemusí byť práve spôsob založený na manipulácii s poznatkami typický pre umelú inteligenciu. Tak ako v do zbla dotiahnutých, objektových prostrediach možno 1+1 interpretovať ako objekt 1, ktorého metóda plus zavolaná s argumentom, ktorým je objekt 1, vracia objekt 2, možno v rámci agentovo-orientovaného programovania považovať 1+1 za nahradenie dvoch agentov 1 jedným agentom 2, ktoré vykonáva vo svojom prostredí agent plus. Avšak podobne ako v OOP prevláda C++ nad Smalltalkom, aj na multiagentovom poli možno očakávať víťazstvo rozumnejších a kompromisnejších riešení.

Nakoľko v súčasnosti sme v prechodovej fáze, kedy agenty ešte nezískali všeobecnú obľúbenosť, je pochopiteľné, že na konferenciách o agentoch panuje taký chaos, ako by dnes panoval na konferenciách o OOP, keby tam prišli rôzni vývojári s rôznymi objektovými jazykmi a najrôznejšími aplikáciami dokazovať, že ľahšie sa im tieto aplikácie implementujú s objektami než so záznamami. Pokojne by sa tam mohol stretnúť odborník na monitorovanie pohybu tučniakov v Patagónii s odborníkom na poskytovanie úverov svetovou bankou. A to je častý zjav na najrôznejších akciách o agentoch. Je však na mieste predpokladať, že tento stav nebude trvať donekonečna a AOP sa stane rovnako čistým pojmom ako je dnes OOP.

Jazyk LINDA

Tento jazyk paralelného programovania s pomerne pikantným názvom (dostal ho údajne po jednej ľahkej žene a na dodatočnú žiadosť prisúdiť mu skratku jeho autora nenapadlo nič lepšie ako Linda Is Not aDA) bol navrhnutý ako prostriedok komunikácie medzi paralelnými procesmi. Hoci je to pomerne dávna záležitosť (konečnú podobu dostala od svojho autora Gelerntera v roku 1985), tento jazyk prežil aj pokles záujmu o paralelné programovanie a svoje miesto si našiel v tzv. koordinačnom programovaní, čo je špeciálna odnož WAN programovania. Pre nás je zaujímavý tým, že preferuje výhradne nepriamu komunikáciu, takže je v princípe podobný tomu, čo budeme používať. Nepovažujem za náhodu, že implementáciu jazyka LINDA pre WAN programovanie zvanú Jada (*Ja*va-Lin*da*), urobili Ciancarini a Rossi zhruba v tom istom čase, keď sme urobili implementáciu našej architektúry. Hoci naša architektúra slúži na rozdiel od Jada nie distribuovanému ale „concurrent" programovaniu, obe sú založené na rovnakej myšlienke založenej na špecifickej koordinácii paralelných procesov. Pritom nezáleží na tom, či ide o paralelizmus skutočný, teda na paralelnom počítači alebo v počítačovej sieti, alebo zdanlivý, teda v operačnom systéme s „multitaskingom".

Základným prvkom jazyka LINDA je dátová štruktúra zvaná „tupple space", ktorá dokáže obsahovať n-tice termov; v princípe LISPovské zoznamy. Táto štruktúra je prístupná procesom, ktoré s ňou môžu manipulovať pomocou primitívnych funkcií:

- out(t) zapisuje novú n-ticu
- in(t) prečíta a odstráni určitú n-ticu; pokiaľ taká nie je k dispozícii, proces sa v čítaní zablokuje do doby, kedy sa objaví

- rd(t) robí to čo in(t), len n-ticu neodstraňuje ale ponecháva
- inp(t) vracia TRUE a odstráni určitú n-ticu pokiaľ taká je; inak vráti FALSE
- rdp(t) robí to čo inp(t), len n-ticu neodstraňuje ale ponecháva

Pritom je pri čítaní na špecifikáciu manipulovanej n-tice použitý unifikačný princíp, čítame napr. n-ticu (Janko*, ma, *, rokov) a tak sa dozvieme koľko má Janko Hraško rokov).

V charaktere nepriamej komunikácie jazyka LINDA teda výrazne dominuje synchronizácia čitateľa a zapisovateľa n-tice a referencia n-tíc na základe unifikácie. Odlišné prevedenie týchto dvoch vlastností je typické pre rôzne deriváty, ktoré z jazyka LINDA vznikli pre prenose na iné platformy.

Najznámejším derivátom je Java Space od spoločnosti Sun, ktorý je súčasťou technológie Java Jini, ktorá je určená na budovanie distribuovaných systémov s veľmi premenlivou štruktúrou. V tejto štruktúre hrá Java Space rolu pevných záchytných bodov. Žiaľ Java Space bol navrhnutý len ako „interface“ a jeho implementácie nie sú známe, takže je skôr slávny než užitočný. Pekne však vystihuje, že referencia na základe unifikácie je príliš ťažko realizovateľná pre dnešné bežné platformy a preto miesto toho sa volí zoznam viacerých typov referencie, medzi ktorými dominuje referencia menom a maskou na toto meno. Ďalšou zmenou prechádza synchronizácia. Už nie je taká prísna a je doplnená „notifikáciou“ zodpovedajúcou modelu spracovania udalostí knižnice swing. Taktiež názvoslovie funkcií sa obracia k bežne zaužívaným názvom read/write. Novinkou je „leasing“ dát. Spočíva v zavedení časovej platnosti dát, po vypršaní ktorej sú dáta zo „space“ odstránené. (Názov je odvodený od predstavy, že si agenty prenajímajú miesto na svoje dáta v „tupple space“.)

```
package net.jini.space;
import java.rmi.*;
public interface JavaSpace {
  Lease write (Entry entry, Transaction txn, long lease)
          throws TransactionException, RemoteException;
  Entry read (Entry tmpl, Transaction txn, long timeout)
          throws UnusableEntryException, TransactionException, InterruptedException, RemoteException;
  Entry readIfExists (Entry tmpl, Transaction txn, long timeout)
          throws UnusableEntryException, TransactionException, InterruptedException, RemoteException;
  Entry take (Entry tmpl, Transaction txn, long timeout)
          throws UnusableEntryException, TransactionException, InterruptedException, RemoteException;
  Entry takeIfExists (Entry tmpl, Transaction txn, long timeout)
          throws UnusableEntryException, TransactionException, InterruptedException, RemoteException;
  EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listener, long lease,
          MarshalledObject handback)
          throws TransactionException, RemoteException;
}
```

Podobne ako Java Space sa naša architektúra bude líšiť od jazyka LINDA v charaktere synchronizácie a v referencií dát. Obe sa líšia nielen od jazyka LINDA, ale aj od všetkých jeho derivátov, ktoré sú nám známe: Paradise, KNOs, Bauhaus, Jada. Niektoré z nich sú známe ako tzv. „backboard“ architektúry. S Java Space zdieľame myšlienku „leasingu“ dát, len mu dávame prízemnejší názov časová platnosť a o niečo zložitejšiu formu. Hoci sme pri tvorbe našej architektúry o existencii jazyka LINDA nevedeli, takže je nášmu prístupu skôr starším bratrancom než predchodcom, patrí mu z hľadiska nepriamej komunikácie historický primát. Napriek istým rozdielom, je toho dosť, čo sa dá z jazyka LINDA aplikovať pre naše potreby, napríklad vybudovanie nad klasickým OOP. Naopak, podstatným rozdielom je, že LINDA v princípe neuvažuje, že by n-tica v prostredí slúžila na inú dátovú výmenu než medzi dvomi procesmi, teda typu 1:1. My naopak budeme používať tento koncept práve kvôli tomu, že potrebujeme výmenu typu many:many.

SRR model posielania správ

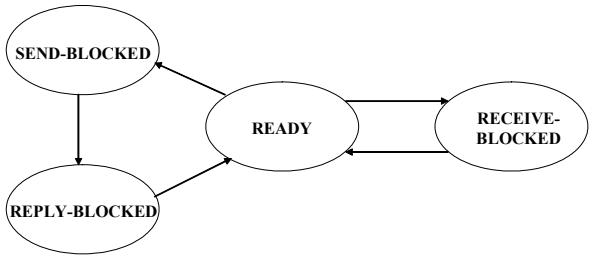
Najkomfortnejším prostriedkom komunikácie medzi procesmi je posielanie správ. Je žiaľ dostupné len na výnimočných platformách, ako je napr. QNX4, svojho času najlepší RTOS (nielen kvalitou, ale aj objemom na trhu). Obohacuje preemptívny „multitasking“ o jednotný a jediný spôsob výmeny dát medzi procesmi a ich synchronizácie. Najznámejším modelom posielania správ je tzv. SRR model, ktorý okrem toho, že je natívny v QNX4, je ako prídavný modul k dispozícii pre Linux (www.cogent.com). SRR model bol prvým prostriedkom nad ktorým sme vybudovali našu architektúru a umožňuje nám začleniť ju do problematiky tvorby systémov reálneho času.

V SRR modeli má každý proces jednoznačný pid („process ID“) a môže mať štyri stavy:
- READY – proces je vykonávaný v procesore alebo je k tomu pripravený
- SEND-BLOCKED – proces je zablokovaný, lebo čaká na odoslanie správy
- REPLY-BLOCKED – proces je zablokovaný, lebo čaká na príjem odpovede
- RECEIVE-BLOCKED – proces je zablokovaný, lebo čaká na príjem správy

a môže volať tri blokujúce komunikačné primitívy:
- void Send (pid_t pid, void *dataout, void *datain, int sizeout, int sizein)
- pid_t Receive (pid_t pid, void *datain, int sizein)
- void Reply (pid_t pid, void *dataout, int sizeout)

(Preto sa tento model volá SRR.)

Možné stavové prechody znázorňuje Obrázok č. 2:

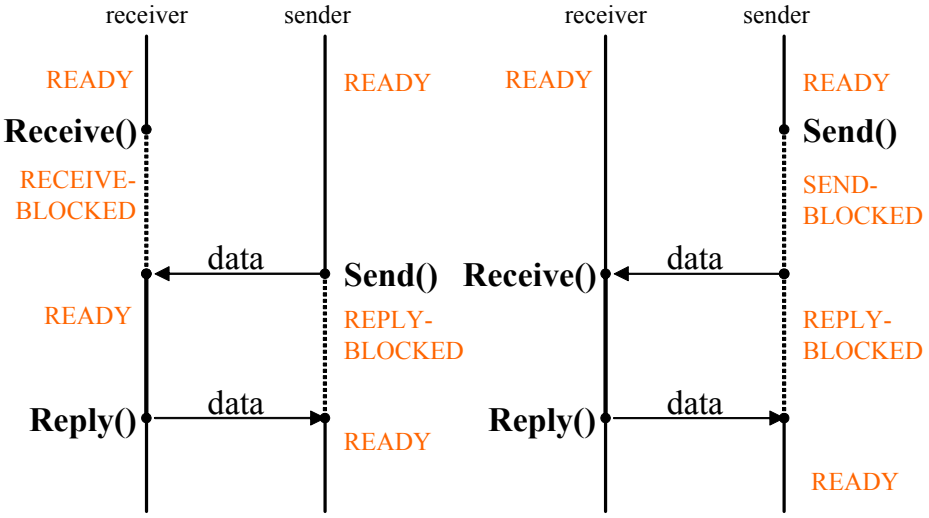


Obrázok č. 2 Stavové prechody v SRR modeli

V komunikácii rozlišujeme „sendera“ a „receivera“. „Sender“ volá Send(), „receiver“ Receive() a Reply(). Send() slúži na poslanie otázky a prijatie odpovede, Receive() na prijatie otázky a Reply() na poslanie odpovede.

Podľa toho, či prv zavolá „receiver“ Receive() alebo „sender“ Send(), sú možné dva scenáre. Ak zavolá najprv „receiver“ v stave READY Receive(), je tento zablokovaný v stave SEND-BLOCKED, v ktorom čaká na zosynchronizovanie sa so „senderom“, teda na stav, kedy je možné posielané dáta prekopírovať z pamäte „sendera“ do pamäte „receivera“. Keď potom nejaký „sender“ zavolá Send() na tohto zablokovaného „receivera“, dáta zo „sendera“ sa prekopírujú do “receivera“ a stav „sendera“ sa zmení (okamžite) na REPLY-BLOCKED. Naopak, „receiver“ prejde po kopírovaní so stavu READY a môže na prijatú otázku zostaviť adekvátnu odpoveď. Túto odpoveď pošle pomocou Reply(). Keďže „sender“ je stále zablokovaný, „receiver“ sa nezablokuje a dáta sa ihneď prekopírujú. Tým pádom prejde „sender“ do stavu READY (viď Obrázok č. 3).

Iný priebeh vznikne, keď najprv „sender“ v stave READY zavolá Send(). Vtedy prejde do stavu SEND-BLOCKED a čaká kým „receiver“ zavolá Receive(). Keď sa tak stane, dáta zo „sendera“ sa prekopírujú do „receivera“ a stav „sendera“ sa zmení na REPLY-BLOCKED. V tomto stave čaká „sender“ až pokým „receiver“ nezavolá Reply() a prekopíruje sa odpoveď. Stav „receivera“ sa v tomto prípade nemení, je stále READY (viď Obrázok č. 3 vpravo).

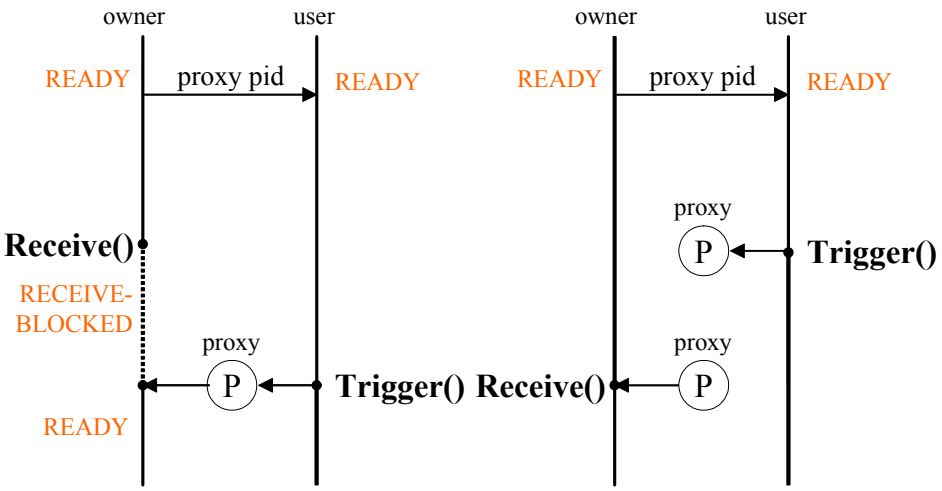


Obrázok č. 3 Blokujúce posielanie správ v SRR modeli

Na to aby mohol „sender“ poslať správu, musí poznať pid „receivera“. Pokiaľ tento nie je napríklad jeho rodičovský proces, je to v princípe problém. Preto SRR používa špeciálny mechanizmus pomocou ktorého môže „receiver“ zverejniť svoj pid pod určitým menom (primitíva name_attach()). „Sender“ potom zavolá službu ktorá mu podľa mena vráti zverejnený pid (primitíva name_locate()).

Podobne „receiver“ musí vedieť pid „sendera“, keď mu chce poslať odpoveď. Tento pid sa však ľahko dozvie, lebo ho vracia primitíva Receive().

Prenášané dáta majú povahu “buffrov”, je to teda postupnosť bytov určitej dĺžky. Vystáva preto otázka odkiaľ sa “receiver” dozvie dĺžku “buffra” do ktorého má prijať dáta. Ak je totiž dĺžka prijímacieho „buffra“ menšia než vysielacieho, dáta sa neprenesú celé. Naopak, ak je väčšia, zvyšná časť prijímacieho „buffra“ ostane v nedefinovanom stave. Odpoveď na túto otázku je prostá: to, aby veľkosti sedeli, zabezpečuje programátor. Spravidla definuje určitú štruktúru prenášaných dát a zároveň ich veľkosť. Je nepísaným pravidlom, že táto štruktúra začína dvomi slovami: prvé zvané „header“ identifikuje štruktúru a druhé zvané „action“ identifikuje ako čítať dáta ktoré nasledujú. Za týmito slovami totiž ide spravidla „union“ s rôznymi variantmi štruktúr ktoré sa pri komunikácii používajú. Príjemca teda podľa „headra“ rozpozná, aký typ štruktúry má použiť na pochopenie dát a podľa „action“ rozpozná, ktorá konkrétna zložka „unionu“ mu bola doručená. Veľkosť prijímacieho „buffra“ je teda stanovená na maximum zo sizeof() všetkých štruktúr, ktoré prijímame. Pre prípad, že by nám niekto zaslal dáta inej štruktúry, spoznáme to pomocou slova „header“. To sa totiž naozaj môže stať a to kvôli tomu, že určitý pid, ktorý si nejaký proces zapamätal s úmyslom, že naň pošle dáta, je po nejakej dobe od skončenia svojho procesu recyklovaný a pridelený inému procesu.



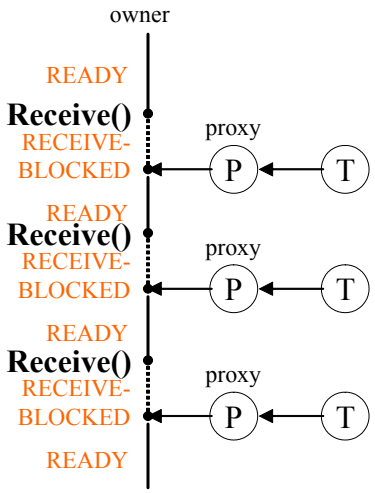
Obrázok č. 4 Neblokujúce posielanie správ cez proxy v SRR modeli

Zatiaľ sme v rámci SRR modelu hovorili o blokujúcom posielaní správ. Jeho výhodou je dokonalá synchronizácia procesov, ktorá nastáva v momente kopírovania dát. Nevýhoda však spočíva v tom, že proces, ktorý chce správu poslať, musí čakať kým ju príjemca prevezme. Pritom z povahy veci môže vyplývať, že vôbec žiadnu odpoveď od príjemcu nepotrebujeme. Na prekonanie tejto nevýhody slúži neblokujúca forma posielania správ, ktorá je v SRR zúžená na možnosť poslať dáta nulovej dĺžky. Neblokujúco sa dá teda poslať len štuchanec, len správa, že sa niečo stalo, ale už nie čo sa presne stalo. Toto obmedzenie vyplýva so spôsobu jeho implementácie pomocou tzv. proxy. V rámci SRR (proxy je inak dosť často používaný názov na rôzne veci, spravidla však majú niečo spoločné s vzdialeným volaním či dátovými prenosmi) je proxy virtuálny proces (má svoj pid), ktorý je v skutočnosti púhym počítadlom. Proxy vždy patrí určitému procesu. Takýmto vlastníkom sa proces stáva pomocou volania primitívy proxy_attach(), od ktorej sa dozvie pid, pridelený vytvorenému proxy. Ak iný proces tento pid pozná, môže toto proxy „triggernúť“ pomocou primitívy Trigger(), čím zvýši dané počítadlo. Keď potom vlastník proxy zavolá Receive(), kontroluje sa či počítadlo s ním asociované je nenulové. Ak áno, Receive() vráti pid tohto proxy a počítadlo sa zníži (Obrázok č. 4, vpravo). Opačný prípad nastáva, keď vlastník zavolá Receive() skôr, než mu niekto proxy „triggerne“. Vtedy ostáva zablokovaný v stave RECEIVE-BLOCKED, až kým sa tak nestane (Obrázok č. 4, vľavo). Proxy teda pred zablokovaním nechráni svojho vlastníka, ale toho, kto ho „triggeruje“.

Samozrejme otázkou ostáva ako sa používateľ proxy dozvie jeho pid. V SRR na to nie je iná možnosť, než použiť blokujúcu komunikáciu, teda používateľ zavolá Send(), v ktorom požiada vlastníka, aby mu pridelil určité proxy. Ten po prijatí tejto požiadavky v Receive(), zavolá proxy_attach() a získaný pid proxyho mu pošle späť cez Reply(). Pritom si zapamätá, že keď mu Receive() niekedy vráti tento pid, znamená to, že ten daný proces od neho niečo chce (pid tohoto procesu sa dozvie z Receive(), ktorým bola prijatá žiadosť o pridelenie proxy a asociáciu pidu proxy a pidu jeho používateľa si musí vlastník pamätať). Následne potom môže vykonať určitú akciu. Jednou z možností je opýtanie sa používateľa čo presne chce a to prostredníctvom zavolania Send() na jeho pid.

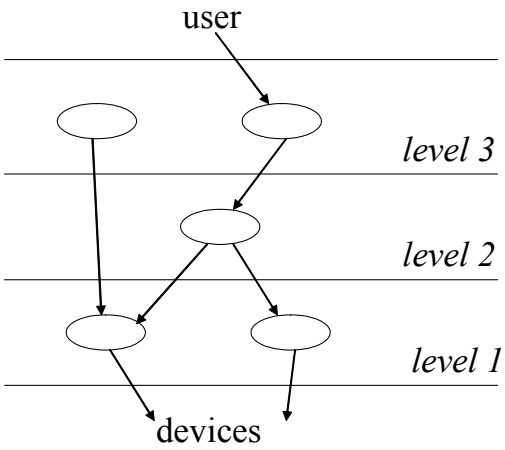
Keďže SRR je model slúžiaci hlavne pre potreby systémov reálneho času, zahŕňa v sebe ďalší typ virtuálneho procesu a to časovač („timer“). Časovač možno vytvoriť cez primitívu timer_create() a cez timer_set() ho možno nastaviť aby o nejaký čas a prípadne

s nejakou periódou poskytoval časové impulzy. Tie sú realizované tak, že časovač „triggeruje“ určité proxy (Obrázok č. 5).



Obrázok č. 5 Použitie časovaču v SRR

Model SRR je síce unikátny z hľadiska jednotnosti dátovej výmeny medzi procesmi – počnúc od jadra systému až po grafické aplikácie – nič menej v sebe skrýva aj určité záludnosti. Nemožno s ním totiž pracovať bez dodržiavania určitého poriadku, v opačnom prípade by sme v systéme rýchlo vytvorili tzv. „deadlock“. Napríklad by sme nechali dva procesy naraz jeden na druhého urobiť Send(), alebo by urobili nejaký väčší cyklus zo Send(). Podobne sa deadlock dá spraviť z Receive(), či kombinácie Send() a Receive(). Vo všeobecnosti môžeme definovať reláciu medzi procesmi vyjadrujúcu, že sa určitý proces môže zablokovať, kým iný niečo nespraví. Ak je v tejto relácii cyklus, je zle. Základným prostriedkom ako tomuto zabrániť je poslúchať určité pravidlá, podriadiť sa istej architektúre. Odporúčaným riešením pre SRR je tzv. pyramidálna architektúra klient-server.



Obrázok č. 6 Pyramidálna architektúra klient-server

Táto architektúra vyžaduje aby medzi každými dvomi vzájomne komunikujúcimi procesmi bol vzťah klient-server (Obrázok č. 6 to zobrazuje šípkou od klienta k serveru). Klientom je pritom „sender“, serverom „receiver“. Proces môže byť vo vzťahu k jednému procesu

serverom a súčasne k inému klientom. Typické je to vtedy, keď požiadavku určitého procesu prijíma ako server, ale vybavuje ju prostredníctvom iných procesov ako ich klient. Pyramidálnosť architektúry spočíva v tom, že požaduje, aby návrhár definoval určité úrovne a každý proces, ktorý v nejakom vzťahu vystupuje ako server pevne priradil na určitú úroveň. Pritom to musí urobiť tak, aby každý klient mal svojich serverov iba na nižších úrovniach. Tým je jednak triviálne zaručené, že nevznikne „deadlock“, jednak, že na najnižšej úrovni budú zariadenia a na najvyššej pomyselný užívateľ.

Na ukážku uvedieme ako vyzerajú kódy charakteristických procesov. Keď chceme nakódovať určitý server, spravidla si zadefinujeme jeho komunikačnú štruktúru server_msg (tá bude popisovať dáta vymieňané s klientami, t.j. definuje ako prevedú jednotlivé údaje do bytov a späť – tzv. „marshalling“ a „demarshalling“), tzv. „port“ (údaj ktorý si server pamätá o svojom klientovi, napríklad asociáciu medzi jeho pidom a pidom prideleného proxy, ale vo všeobecnosti stav dialógu s klientom) a môžeme kódovať (Obrázok č. 7):

```
typedef struct server_msg {
   short header;
   short action;
   union {  ...
   };
};

typedef struct server_port {
   pid_t pid;
   ...
};

main () {
   struct server_msg msg;
   struct server_port *port;
   // inicializacia
   name_attach();
   ports_init();
   for (;;) {
      pid = Receive(0,&msg,sizeof(msg));
      if (msg.header != SERVER_HEADER)
         continue;
      ports_reinit();
      if ((port = port_get(pid)) == -1) {
         port = port_new();
         port_setdefaults(port);
      }
      switch (msg.action) {
         case SERVER_ACTION1:
            // spracuj *port a msg
            break;
         ...
         case SERVER_ACTIONx:
            ...
            break;
      }
      Reply(pid,&msg,sizeof(msg));
   }
}
```

Obrázok č. 7 Typický kód serveru v SRR

Server si teda „attachne“ meno, čím prezradí svoj pid a potom v nekonečnom cykle prijíma požiadavky od klientov. Pre každú požiadavku najprv overí či mu nie je zaslaná omylom, teda či sa jej „header“ zhoduje s očakávanou identifikáciou štruktúry, ktorú podporuje. Potom identifikuje, ktorého klienta sa požiadavka týka a ďalej koná s vedomím stavu dialógu uloženého v príslušnom porte. Pokračuje tým, že podľa hodnoty „action“ požiadavku pochopí a vykoná príslušnú službu, čím zostaví odpoveď. Tá má rovnaký „header“ ale s vysokou pravdepodobnosťou rôznu „action“, nakoľko odpoveď bude asi vyžadovať iný tvar dát než požiadavka. Na záver odpoveď odpošle klientovi, čím ho odblokuje.

Aby server fungoval ako sa patrí musí väčšinu času stráviť v stave RECEIVE-BLOCK na volaní Receive(). Iba vtedy je totiž pripravený slúžiť poriadne svojim klientom. Nevyhnutným predpokladom teda je, že odpoveď na žiadosť o službu nasleduje dostatočne rýchlo. (V žargóne sa tomu hovorí „blik-blik“, čo je komický ale veľmi výstižný názov: to prvé blik predstavuje žiadosť a to druhé odpoveď, pričom druhé musí nasledovať tesne po prvom).

Pokiaľ z hľadiska požadovanej služby nie je možné medzi žiadosťou a dostatočne rýchlou odpoveďou službu vykonať, je stále možné kooperáciu urobiť, aby bola „blik-blik“, hoci je to dosť pracné. Miesto scenára „daj mi miliónte prvočíslo“ – „tu ho máš“, musíme použiť scenár „daj mi miliónte prvočíslo, moje proxy je XY“ – „OK, idem počítať“ – „trigger XY“ – „čo si vyrátal ?“ – „toľko a toľko“. Pritom však musíme zariadiť, aby server jednak konal službu, jednak neprestal obsluhovať klientov. Sú len dve možnosti ako na to: buď sa služba vykoná v samostatnom vlákne, alebo v samostatnom procese. V oboch prípadoch je pritom pre potreby systémov reálneho času preferované, aby tieto vlákna či procesy boli naštartované v konečnom rozumnom počte a to pri inicializácii systému. V opačnom prípade by nepriaznivo a nekontrolovane ovplyvňovali kvalitu reálneho času.

Na druhej strane pre klienta je ťažšie napísať typický kód, preto uvedieme dve možnosti. V oboch prípadoch klient musí zistiť podľa mena pid servera, potom zostaviť požiadavku, odoslať ju na server, prijať odpoveď, rozobrať ju a použiť. Obrázok č. 8 vľavo ukazuje najjednoduchší prípad, keď klient jednorázovo použije určitú službu (takýto klient sa nazýva „utility“). Vpravo je znázornený zložitejší prípad, keď klient používa určitú službu opakovane, v tomto prípade ho k tomu budí sekundový časovač. Neuvádzame bežný prípad, keď by klient volal službu z vnútra obsluhy určitej služby v rámci vykonávania požiadaviek na serveri, nakoľko takýto kód si čitateľ ľahko domyslí skombinovaním kódu „utility“ a servera (do vnútra obsluhy case SERVER_ACTIONx by sa vložil kód „utility“).

Samozrejme v praxi je kód klienta štruktúrovanejší v tom zmysle, že neobsahuje priamo tú časť, ktorá sa zaoberá skladaním a rozoberaním správy, ale iba volá vhodnú funkciu z klientskej knižnice, ktorej odovzdáva parametre. Prenesenie skladania a rozoberania správy do knižnice má ten zmysel, že sa potom toto kóduje raz, miesto toho aby sme to museli robiť pri každom klientovi. Takéto funkcie, keďže „zaobaľujú“ volanie Send(), sa nazývajú zaobaľovacie (enwrapping) funkcie.

```
main () {
   struct server_msg msg;
   // inicializacia
   pid = name_locate("...");
   msg.header = SERVER_HEADER;
   msg.action = SERVER_ACTIONy;
   // naplni msg.data;
   Send(pid,&msg,&msg,
      sizeof(msg),sizeof(msg));
   // spracuje msg.data
}
```

```
main () {
   // inicializacia
   pids = name_locate("...");
   pidp = proxy_attach();
   pidt = timer_create(-pidp)
   timer_set(pidt,RELATIVE,0,0,1,0);
   for (;;) {
      pid = Receive(0,NULL,0);
      if (pid == pidp) {
         // vytvor msg
         Send(pids,&msg,&msg,
            sizeof(msg),sizeof(msg));
         // spracuj msg
      }
   }
}
```

Obrázok č. 8 Typické kódy klientov v SRR

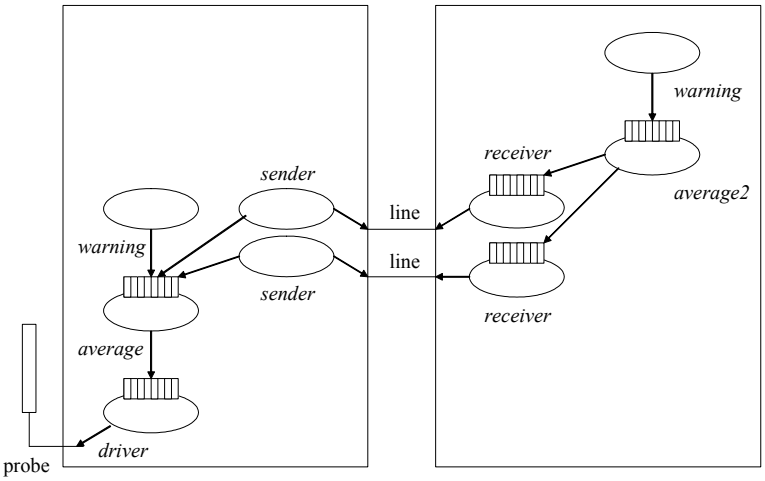
Ako príklad systému reálneho času si teraz uvedieme jednoduchý varovný systém. Hovoriť pri takomto systéme o inteligencii by bolo prehnané, avšak príklad nám umožní neskôr demonštrovať rozdiel medzi riešeniami, ktoré poskytuje štandardná pyramidálna architektúra

klient-server a naša architektúra inšpirovaná metódami umelej inteligencie. Uvidíme, že určité myšlienky vychádzajúce z UI, sú prínosom aj pre obyčajné programovanie. Takže k príkladu:

Príklad č. 1

Majme sondu ktorá meria určitú veličinu. Táto slúži na varovanie osôb. Meranie má značný rozptyl a preto sa kritická hodnota stanovuje pre priemer za určité obdobie. Varovanie prebieha v mieste merania a na inom vzdialenom mieste. Je dôležité a preto sú tieto dve miesta prepojené dvomi nezávislými spojeniami rovnakého druhu, ktoré sa navzájom zálohujú.

Riešenie, ktoré vychádza z pyramidálnej architektúry klient-server ukazuje Obrázok č. 9. Zo sondou bude komunikovať *driver*, ktorý bude na základe impulzov od časovača realizovať jednotlivé merania. Zároveň – ako server – poskytuje poslednú meranú hodnotu. Toto rozhranie využíva jediný klient *average*, ktorý vypočítava priemer za potrebné obdobie a poskytuje ho – ako server – klientovi *warning*, ktorý realizuje varovanie. Tým istým rozhraním sa k priemeru dostanú klienti typu *sender*, ktorí zabezpečia v spolupráci s klientmi typu *receiver* distribúciu priemeru z miesta merania na vzdialené miesto a to dvomi nezávislými cestami. Na vzdialenom mieste poskytuje priemer server *average2*, ktorý ho získava ako klient cez serverovské rozhranie *receiverov* (dostáva normálne dve rovnaké hodnoty ale stačí mu aj jedna). Preto varovanie môže sprostredkovať rovnaký klient *warning*. Toto riešenie sa môže zdať zbytočne zložité, ale pokiaľ by sme napríklad počítali v budúcnosti s rôznymi modifikáciami systému, má zmysel kombinovať zásady zvolenej architektúry z vysokou modularizáciou systému.



Obrázok č. 9 Príklad systému v pyramidálnej architektúre klient-server

Pyramidálna klient-server architektúra sa však vyznačuje aj značnými nevýhodami:

- Na strane serverov neposkytuje žiadnu normalizáciu rozhrania a zvádza k vytváraniu špecifických klientských knižníc pre každý server. Klienti, ktorí používajú viacero serverov, potom z týchto knižníc „tučnejú“.
- Kód procesu, ktorý je zároveň serverom i klientom je bez použitia vlákien („threadov“) dosť odpudzujúci, pokiaľ požadujeme, aby vybavovanie klientských požiadaviek nebolo spomalené blokovaním sa pri volaní služieb od iných serverov. Neustále sa v ňom totiž strieda obsluha klientov s kooperáciou s inými serverami. Toto robí problém hlavne keď táto kooperácia neprebieha „blik-blik“.

- Je značný problém reštartovať určitý server. Jeho pid by sa tým totiž zmenil a všetci jeho klienti by stratili schopnosť s ním komunikovať. Ledaže by tento prípad v kóde každý jeden z klientov osobitne ošetroval, alebo by sme reštartli aj ich. Každopádne je jednoduchšie reštartovať celý systém, čo sa ale nepriaznivo prejaví na kontinuite jeho behu.

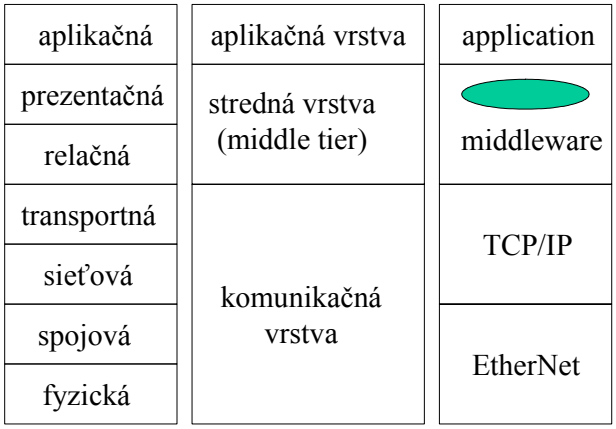
Tieto nevýhody sa stupňujú s komplexnosťou systému a my v práci ponúkame alternatívnu architektúru, ktorá ich úplne odstraňuje (hoci zadarmo nie je).

Iný spôsob ako bojovať s „deadlockom“ predstavuje zavedenie neblokujúceho posielania správ (aj nenulovej dĺžky) medzi procesmi. Obľúbeným riešením je tu tzv. „Message queue“. Ide o špeciálny server, ktorý umožňuje vytvoriť komunikačný kanál medzi dvomi procesmi, do ktorého jeden neblokujúco zapíše a druhý z neho neblokujúco prečíta. Aby sa pritom nestrácali dáta, musí tam byť FIFO („queue“), čím sa ale posielanie správ („message passing“) v podstate degraduje na úroveň „named pipes“, ktoré sú bežne dostupné na väčšine platforiem. Kardinálny problém tohoto riešenia je, že poskytuje iba komunikáciu typu 1:1 alebo many:1. Navyše problémom je, že „queue“ je chápaný len ako púhy komunikačnú kanál, cez ktorý môže prúdiť mnoho rôznych dát rôzneho formátu.

Naše riešenie bude podobné v tom zmysle, že bude v systéme zavádzať špeciálny server, ale ku posledne menovanej vlastnosti bude duálne: bude pracovať ako keby s komunikačným kanálom, ktorý slúži všetkým procesom (many:many), za to však len jednému typu a formátu prenášaných údajov.

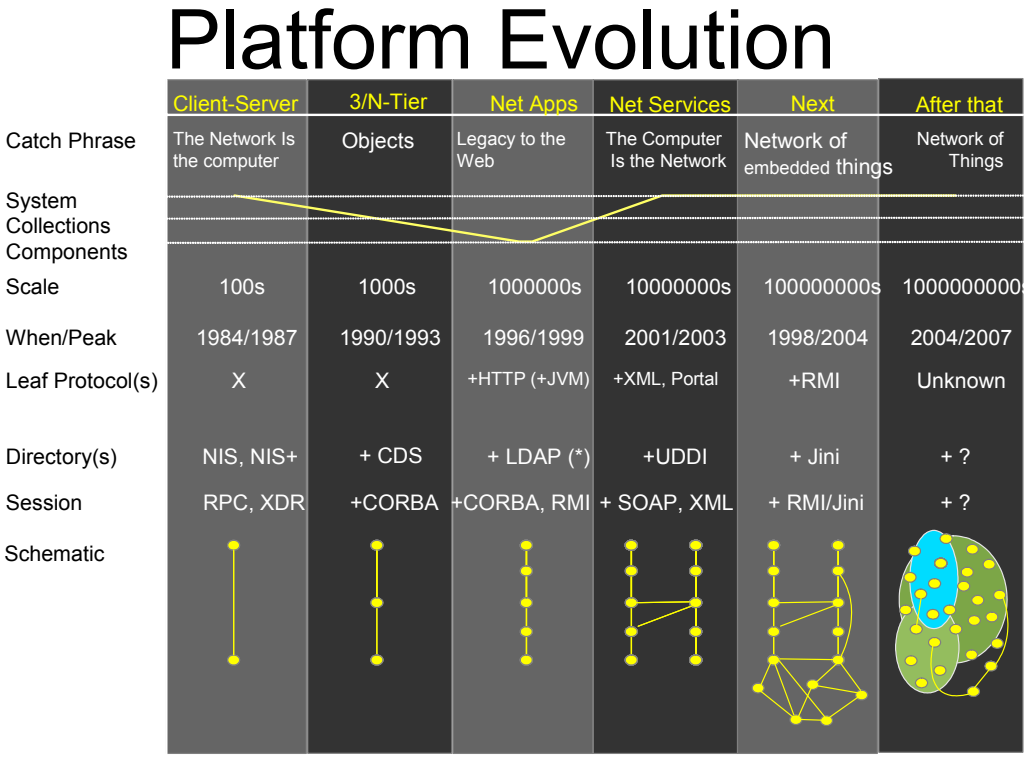
Middleware

Okrem SRR budeme pre porovnanie budovať našu architektúru aj nad sieťovými technológiami. Tu použijeme model ISO OSI, ktorý je našťastie – na rozdiel od SRR modelu – všeobecne známy. Preto stačí spomenúť, že architektúra ktorú budeme prezentovať bude middlewarom – implementáciou relačnej („session“) a prezentačnej („presentation“) vrstvy („layer“). Pritom by sme mohli stavať priamo na vrstve transportnej (t.j. použiť napr. TCP/IP), nič menej efektívnejšie bude postaviť implementáciu nad niektorým existujúcim middlewarom (RMI, CORBA). Je taktiež možnosť použiť niektorý z middlewarov, ktorý implementuje nejaký mechanizmus IPC cez TCP/IP (SRR napríklad takto implementuje FLEET, ďalší dobrý kandidát je PVM). Obrázok č. 10 vymedzuje oválom pole na ktorom budeme implementačne pôsobiť.



Obrázok č. 10 Postavenie MAS v rámci modelu ISO OSI

Pre nás istým povzbudzujúcim prvkom je, že vývoj middlewarov postupuje od jednoduchých štrukturálnych vzťahov ku zložitým štruktúram, kde sa každá architektúra musí vysporiadať s problémom deadlocku, s potrebou dátových tokov many:many, s problémami rýchlej odozvy a podobne. Tento vývoj pekne demonštruje Obrázok č. 11 .



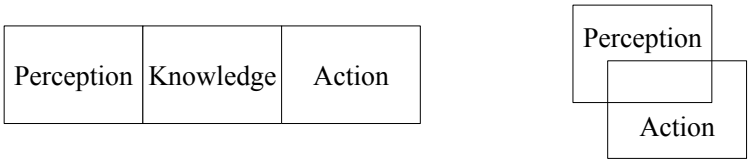
Obrázok č. 11 Odhad vývoja sieťových platforiem z [Waldo 2001]

Pekne tu vidno, že vývoj sieťových platforiem kopíruje vývoj jednouzlových architektúr s cieľom unifikácie oboch prístupov. Pritom triky jedného prístupu sa stávajú motiváciou pre druhý a opačne. Kým v istých fázach vývoja prevláda snaha použiť sieť na vybudovanie virtuálneho počítača s vysokým výkonom („The network is the computer“), v inej fáze vývoja prevláda uplatňovanie princípov distribuovaného programovania pre lokálne potreby na jednom uzle („The computer is the network“). Do tohto chlievika patrí aj architektúra prezentovaná v tejto práci. Na rozdiel od iných autorov však pritom budeme venovať veľmi malú pozornosť rozdielom medzi sieťovým („network“) a jednouzlovým („concurrent“) programovaním, lebo náš primárny záujem je budovať architektúru pre jeden uzol. Budeme teda ignorovať už spomínané Deutschove klamy („Deutsch's falacies“):

1. sieť je spoľahlivá
2. čas odozvy je nulový
3. šírka pásma je nekonečná
4. sieť je bezpečná
5. topológia siete sa nemení
6. v sieti je jediný administrátor
7. cena prenosu dát je nulová (nezáleží na prenášanom objeme), ktoré pri žiadnom sieťovom riešení nemožno ignorovať.

Subsumpčná architektúra a jej deriváty

V polovici osemdesiatych rokov minulého storočia pretavil R. Brooks kritiku tradičnej umelej inteligencie („Good Old-Fashioned AI“) do alternatívneho riešenia („New AI“). Rozdiel medzi týmto novým a starým prístupom spočíval hlavne v type dekompozície systému. Kým modul pri GOFAI obsahuje kódy, ktoré sa zhodujú vo funkcii, ktorú vykonávajú (sú podobné z algoritmického hľadiska), modul NAI obsahuje kódy, ktoré slúžia na produkovanie určitej zložky správania systému (majú rovnaké poslanie). Pokiaľ na systém aplikujeme dekompozíciu funkciou (GOFAI), dostaneme ako jeden z modulov tzv. kognitívny modul, ktorý je zodpovedný za logické odvodzovanie nad vnútornou reprezentáciou poznatkov. Percepcia a akcia budú teda modulárne oddelené od kognície. Kameňom úrazu tohto riešenia je práve prepojenie medzi percepciou a kogníciou na jednej strane a kogníciou a akciou na strane druhej. Z hľadiska vstupu do kognitívneho modulu je nevyhnutné dáta vyjadrené rôzne previesť na jednotný tvar (napr. Hornove klauzuly), pričom je problém nastaviť tento proces tak, aby bolo vzaté všetko čo je pre adekvátny výsledok potrebné, ale aby toho nebolo vzaté príliš veľa. Kognitívny modul sa totiž len pri pohľade z vonku javí ako rozumne uvažujúci, vnútri pracuje na čisto syntaktickom základe. V dôsledku toho sa stráca sémantická informácia a odvodzovanie sa často spomaľuje. Na druhej strane výstup z kognitívneho modulu sa dá len málokedy priamo previesť do akcii systému a vyžaduje na tento prevod špecifické postupy, ktorých nároky na „inteligenciu“ v ničom nezaostávajú za samotným kognitívnym modulom. Preto NAI navrhuje aby takýto modul v systéme vôbec neexistoval a aby bola kognícia rozptýlená vo všetkých moduloch, pričom percepcia a akcia sa budú prelínať (Obrázok č. 12).



Obrázok č. 12 Dekompozícia systému podľa GOFAI a NAI

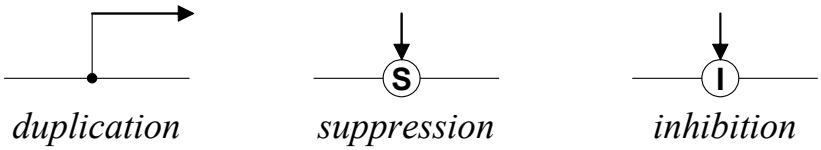
Na základe tejto predstavy Brooks formuloval nasledovné pojmy či postuláty[4]:

- Situovanosť
  - Pri tvorbe neuvažujeme, že sa vytváraný systém nachádza v akýchkoľvek podmienkach vyjadrených všeobecne, ale uvažujeme veľký súbor špecifických podmienok
  - Môžeme najprv urobiť verziu, ktorá funguje len pre časť týchto podmienok a neskôr ju rozšíriť na ostatné
  - Nebudeme podmienky ošetrovať jediným modulom, ale špecifické podmienky budú ošetrované špecifickými modulmi – „raz zaberie taký trik, inokedy onaký“
  - Nesnažíme sa budovať abstraktnú reprezentáciu sveta. „Najlepšou reprezentáciou sveta je svet sám“.
  - Systém vyvíjame pre konkrétne prostredie, v ktorom ho testujeme (ladíme)
- Stelesnenosť
  - Neoddeľujeme návrh od implementácie
  - Navrhujeme s prototypom v ruke
  - Tým je umožnené po skončení každej fázy vývoja systém testovať a ladiť.
  - Pracujeme so skutočnými vstupmi, spoliehame sa na ich reálny charakter.
- Emergencia
  - Správanie systému je výsledkom interakcie jednotiek z ktorých sa skladá
  - Súvislosť medzi správaním jednotiek a systému môže byť na prvý pohľad nejasná

  - Pri vhodnej štruktúre systému môžeme explicitnou implementáciou určitých schopností implicitne implementovať schopnosť doteraz neuvažovanú. Vývojár môže byť neschopný tento jav predvídať, hoci jeho dodatočné vysvetlenie by aj nebolo zložité.
- Interakcia
  - Systém môže inteligenciu čerpať z dynamiky prostredia. Často možno zložitý riadiaci systém nahradiť jednoduchým, ktorý vhodne stavia na dynamike prostredia (napr. jeden druh morského slimáka v akváriu uhynie, lebo keď vylezie z vody, nevie sa do nej vrátiť; v mori však dokáže prežiť, lebo na to používa príliv a odliv)
  - Paradoxne (na rozdiel od systémov ktoré obsahujú kognitívny modul) v statickom prostredí sa takýto systém správa horšie než v dynamickom.
  - Inteligencia sa prejavuje reakciami systému na stav jeho prostredia, teda v globálnom správaní, nemožno ju nájsť vnútri systému
  - Reaktivita systému zabezpečuje jeho odolnosť voči rušivým zmenám a prechodným stavom v prostredí
- Hierarchia (Inkrementálnosť)
  - Systém vyvíjame inkremenálne, po vrstvách, pričom každá implementuje určitú aktivitu
  - Postupujeme pritom zdola nahor. Začíname s hierarchicky nižšími vrstvami (biologická metafora: historicky staršími), teda s primitívnejšími aktivitami
  - Postupne pridávame nové a nové vrstvy, pričom po každom inkremente, testujeme, ladíme a opravujeme, kým naozaj nezbehnú všetky testy určené pre danú fázu vývoja úspešne
  - Vďaka tomu sa nám nemôže stať že implementácia zlyhá na integrácii častí v celok. Hoci na druhej strane nám potenciálne hrozí, že systém dostaneme do stavu, že novú aktivitu už nemožno konzistentne pridať
  - Hierarchia systému spočíva v tom, že vyššie vrstvy môžu využívať nižšie. Môžu ich pasívne monitorovať alebo aj aktívne na ne pôsobiť tým, že potláčajú alebo dokonca modifikujú ich činnosť.

Na základe týchto postulátov navrhol Brooks pre tvorbu mobilných robotov tzv. subsumpčnú architektúru. Podľa nej sa systém skladá z modulov organizovaných do vrstiev, ktoré zodpovedajú jednotlivým aktivitám a zároveň fázam vývoja. Každý modul je založený na konečno-stavovom automate rozšírenom o vstupno-výstupné a interné registre, o možnosť volať náročnejšie výpočtové procedúry a hlavne o časovač. Tento zabezpečuje, že modul opakovane transformuje signály prijaté do vstupných registrov na signály zapísané do výstupných registrov. V rámci jednej vrstvy môžu byť moduly vzájomne prepojené vedením, po ktorom sú signály z výstupných registrov odposlané do vstupných registroch iných modulov. Okrem toho sa moduly z vyššej vrstvy môžu „vtierať“ („intrude“) do činnosti modulov v nižších vrstvách a to prostredníctvom troch mechanizmov (Obrázok č. 13):

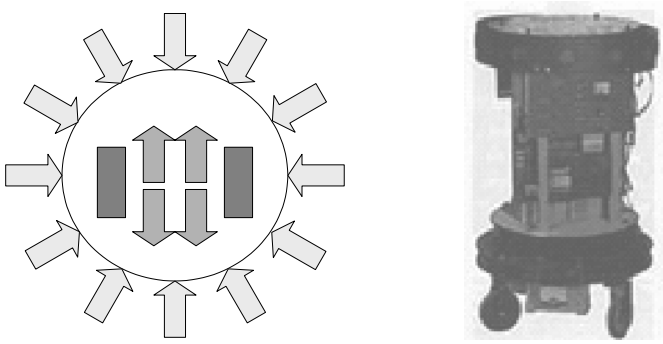
- odpočúvaním – pri ňom dochádza k rozdvojeniu vedenia v nižšej vrstve a odvedenia jeho novej vetvy do vyššej vrstvy; modul vo vyššej vrstve takto monitoruje, aké signály sú posielané medzi dvomi modulmi vo nižšej vrstve
- „supresiou“ – pri nej do vedenia v nižšej vrstve vkladáme špeciálne zariadenie – „supresor“ – a napájame ho na výstup z modulu vo vyššej vrstve. „Supresor“ zabezpečí, že signál z vyššej vrstvy nahradí signál, ktorý sa šíri po vedení v nižšej vrstve.
- „inhibíciou“ – pri nej podobne vkladáme do vedenia v nižšej vrstve „inhibítor“, ktorý ak dostáva signál z vyššej vrstvy, zastaví šírenie signálu v nižšej vrstve.



Obrázok č. 13 Mechanizmy subsumpcie

„Supresor“ a „inhibítor“ majú pritom časovú konštantu, ktorá zabezpečuje, že ak začnú „supresiu“ či „inhibíciu“ bude to trvať istý čas. (V určitých verziách Brooks ďalej požaduje aby sa „inhibítor“ používal na výstupe z modulu, zatiaľ čo „supresor“ na vstupe do modulu. Teda na určitom vedení by boli všetky „inhibítory“ pred všetkými „supresormi“. Nie je nám známy zmysel toho obmedzenia a preto ho neuvažujeme.)

Vstup a výstup niektorých modulov je prepojený so senzormi a aktuátormi systému a to bez ohľadu na to v ktorej vrstve sa modul nachádza (Obrázok č. 15 ich označuje obojsmernou šípkou.)

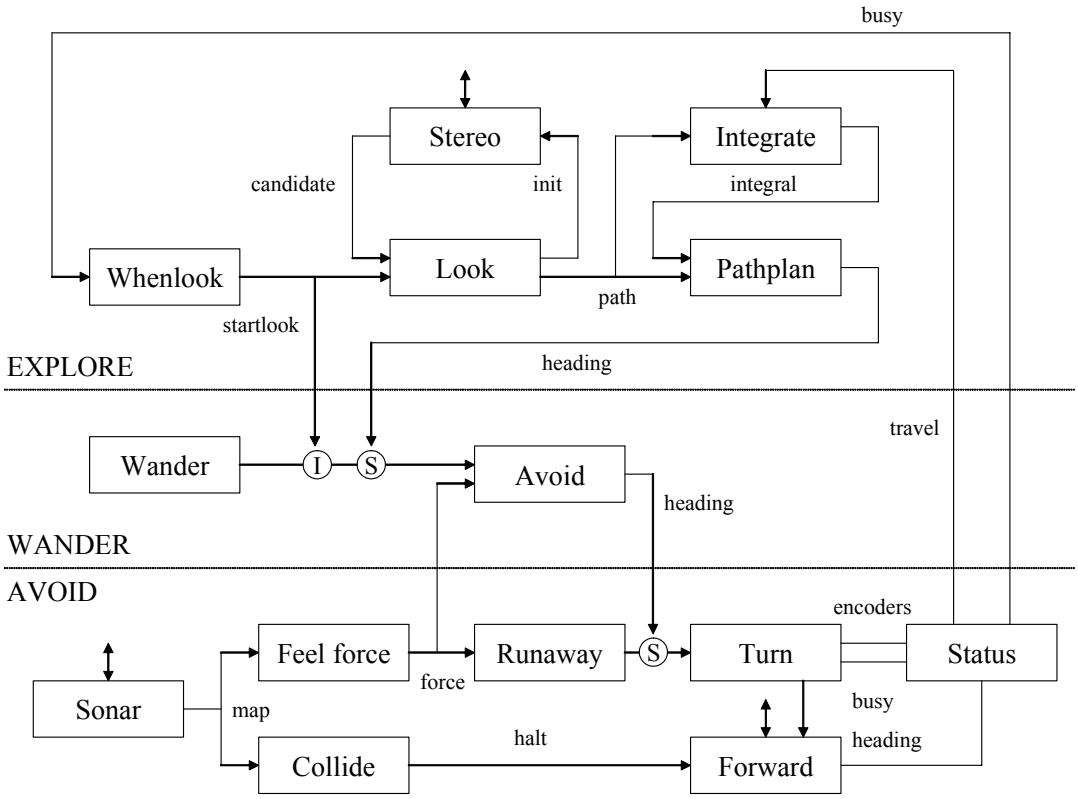


Obrázok č. 14 Senzory a aktuátory robota ALLEN

Ako príklad si uvedieme najjednoduchší (a najstarší) robot postavený subsumpčnou architektúrou a to robot ALLEN [Brooks 1991]. Tento prípad použijeme v ďalšom, preto ho uvedieme relatívne podrobne (pričom upozorňujeme, že isté nepublikované detaily o tomto robote musíme odhadnúť). ALLEN je robot, ktorý má prechádzať miestnosťami na jednom podlaží aby navštívil podľa možnosti čo najviac miest (čo je základ pre budovanie kognitívnej mapy podlažia). Robot je vybavený dvanástimi sonarmi (ultrazvukovými senzormi), ktoré merajú vzdialenosť najbližšej prekážky v okolí robota im pridelenom výseku. Sonary sú rozmiestnené rovnomerne, takže na každý sonar pripadá výsek veľkosti 30°. Podvozok robota umožňuje pohyb dopredu a dozadu ako aj otáčanie sa na mieste (Obrázok č. 14).

V zmysle subsumpčnej architektúry sa ALLEN skladá z troch vrstiev (Obrázok č. 15): AVOID, WANDER a EXPLORE. Ako prvá bola implementovaná vrstva AVOID realizujúca najjednoduchšiu aktivitu a to vyhýbanie sa prekážkam. Začína modulom *Forward*, ktorý ovláda dopredný a spätný pohyb a pokiaľ nedostane iný pokyn realizuje dopredný pohyb robota. Teda keby mal robot iba tento modul šiel by rovno a narazil by do prvej prekážky. Aby sa tak nestalo je vo vrstve AVOID modul *Sonar*, ktorý prijíma vzdialenosti najbližších prekážok zo sonarov a na základe toho produkuje mapu týchto vzdialeností, indexovanú relatívnym smerom. Časť tejto mapy, ktorá sa viaže k dopredným smerom (napríklad 300° - 60°) vstupuje do modulu *Collide*, ktorý vyhodnotí či nastala, alebo hrozí zrážka. (Robot sa pohybuje vždy v smere 0°, lebo všetky smery sú relatívne – robot nemá žiadny kompas). Pokiaľ *Collide* vyhodnotí, že nastala alebo nastáva zrážka zaradí na určitú dobu spätný chod prostredníctvom modulu *Forward*. S týmito modulmi už robot do prekážky nenarazí, ale cúvne

pred ňou. Nič mu to však nie je platné, lebo po cúvnutí opäť vyrazí smerom k prekážke a tak stále dokola. Zjavne treba pridať otáčanie. To realizuje modul *Turn*, ktorý potláča činnosť modulu *Forward* (robot sa otáča na mieste). *Turn* však realizuje otáčanie iba keď na to dostane pokyn. Tento generujú moduly *Feelforce* a *Runaway*. *Feelforce* vyhodnocuje z mapy vzdialeností smer v ktorom najviac hrozí zrážka. Na to *Runaway* vyhodnotí či sa treba otáčať a vypočítava o koľko a ktorým smerom. Túto informáciu potom posiela do modulu *Turn*. Treba si uvedomiť, že tu môže vznikať určitá spätná väzba, keď otáčanie trvá dlhší čas: tým že *Turn* začne realizovať príkaz od *Runaway* na otočenie o 150°, mení sa poloha robota a tým pádom aj mapa a smer v ktorom najviac hrozí zrážka a následne príkaz modulu *Runaway*, ktorý už žiada otočenie len o 100°. Na druhej strane, moduly pracujú v taktoch, takže ak sa žiadosť o otočenie o 5° vykoná prv než prebehne vygenerovanie nového príkazu, tento už bude znieť „netočiť sa“. Ďalším modulom vrstvy AVOID je *Status*, ktorého prítomnosť je v podstate porušením princípu inkrementálneho vývoja zdola nahor (v práci vysvetlíme príčinu a ukážeme ako sa bez neho zaobísť). Jeho úlohou je poskytovať dostatočne výstižné informácie o pohybe robota pre vyššie vrstvy. Pokiaľ by robot obsahoval iba vrstvu AVOID, tak by vyštartoval rovno, pri prekážkach by sa otáčal do voľného priestoru. Zaujímavosťou je to, že hoci AVOID bola navrhnutá pre statické prekážky, funguje rovnako dobre aj pre pohybujúce sa objekty, ktoré sú pomalšie než robot. Už pri prvej časti vrstvy AVOID vie robot cúvať pred pohybujúcim sa objektom, pri celej vrstve dokáže pred ním utekať.



Obrázok č. 15 Subsumpčná architektúra robota ALLEN

Pohyb, ktorý generuje AVOID, je však natoľko deterministický, že pri pustení robota do scény pomerne rýchlo príde k zacykleniu jeho trajektórie. Odstrániť tento nedostatok je úlohou vrstvy

WANDER. Túlanie realizujú dva moduly. Prvý z nich – *Wander* – generuje raz za istý čas (prípadne s istou pravdepodobnosťou) náhodný smer a posiela ho do druhého modulu *Avoid*. Ten má zariadiť, aby sa robot otočil týmto smerom. Keďže vrstva AVOID ovláda aktuátory pohybu, musí tak urobiť s jej pomocou. Tá však žiadne rozhranie na takúto spoluprácu neposkytuje, keďže bola implementovaná spôsobom zdola nahor – teda bez uvažovania, čo bude v implementácii nasledovať. Preto *Avoid* musí použiť jeden z mechanizmov subsumpcie, ktorý takéto rozhranie vytvorí. Konkrétne bude použitá supresia: do vedenia medzi modulmi *Runaway* a *Turn* vložíme supresor napojený na výstup z *Avoid*. Tým pádom *Turn* bude považovať hodnotu z *Avoid* za smer, ktorý mu poslal *Runaway*, teda za smer, ktorým sa treba otočiť, aby nenastala zrážka. Čo treba doriešiť je prípad, keď zrážka naozaj hrozí. Preto *Avoid* použije aj mechanizmus odpočúvania: vedenie medzi *Feelforce* a *Runaway* zdvojíme a odvedieme na vstup modulu *Avoid*, ktorý tak získa potrebnú informáciu o nebezpečenstve zrážky. V takom prípade nebude podnet z *Wander* realizovať.

AVOID a WANDER zabezpečia, že pohyb robota bude bezpečný a nedeterministický. To však zďaleka nestačí na to, aby robot prehľadával priestor. Naopak, bude sa motať na mieste. Opak zabezpečuje vrstva EXPLORE. Jej úlohou je presúvať robot z jednej oblasti do inej udržujúc stály smer pohybu (ale obchádzajúc pritom prekážky). Podmienku aktivácie vrstvy do činnosti vyhodnocuje modul *Whenlook* na základe údajov o pohybe robota od modulu *Status*. Keď sa mu zdá, že sa robot motá na mieste, uvedie do činnosti modul *Look*. Ten spolupracuje s modulom *Stereo*, ktorý preberá údaje zo senzorov a hľadá medzi nimi smer, v ktorom je dostatok voľného priestoru (napr. chodba). Tohto kandidáta posiela modulu *Look*, ktorý ho prípadne vyhlási za smer presunu do inej oblasti od tohto momentu modul *Pathplan*, otáča robota tak, aby stále držal tento smer. Keďže smery sú relatívne, od momentu voľby musí robot počítať odchýlku aktuálneho absolútneho smeru od jeho hodnoty v čase voľby, čo robí modul *Integrate*. Ten na to samozrejme potrebuje dostatočné informácie o pohybe robota – tie mu sprostredkúva modul *Status* z vrstvy AVOID. *Pathplan* pritom realizuje otáčanie robota cez vrstvu WANDER a to opäť supresiou – ako keby šlo o náhodný smer zabezpečujúci túlanie, hoci tento nie je náhodný, ale generovaný podľa presnej logiky. Pritom prekážky budeme obchádzať, lebo *Avoid* nepošle tento smer do *Turn*, ak hrozí zrážka. Je logické zabezpečiť aby sa do činnosti EXPLORE nemiešal modul *Wander*, preto je počas presunu potlačená jeho činnosť mechanizmom inhibície. Nič menej, aj keby sme toto nespravili, presun by sa podaril, len by počas neho dochádzalo k vybočeniam a návratom aj bez výskytu prekážok.

Všetky tri vrstvy realizujú dostatočne bohatý pohyb, aby sa dal použiť na sofistikovanejšie úlohy, ako je napríklad budovanie kognitívnej mapy prostredia.

Na záver spomeňme, že sa subsumpčná architektúra stala podnetom pre tvorbu ďalších podobných architektúr. Konštruktívnej kritike ju podrobili hlavne Rosenblatt a Payton, ktorí jej vyčítali tri vlastnosti:

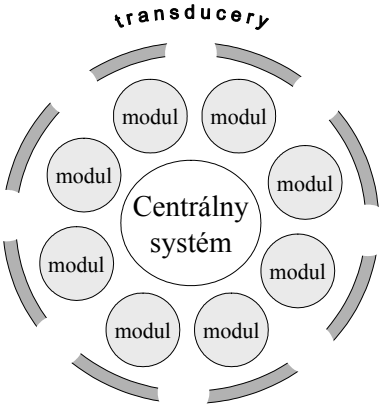
I. Neprístupnosť k vnútornému stavu – pokiaľ sa ukáže, že niektorý z modulov v nižšej vrstve obsahuje vo svojom vnútornom registri informáciu potrebnú pre vyššiu vrstvu, je nemožné ju získať.
II. Nemožnosť porušenia úrovne kompetencie – priorita modulov je vždy presne stanovená a nemenná, nie je možné aby moduly súperili o vplyv na systém.
III. Absencia mechanizmu dátovej fúzie – z možných hodnôt sa vždy vyberie jedna, ktorá prehluší ostatné, ale nie je k dispozícii žiadny mechanizmus, ktorý by ich zlúčol do novej hodnoty, ktorá reprezentuje postoj viacerých modulov.

Na základe tejto kritiky sa potom navrhovali architektúry, ktoré sa snažili buď rozbiť moduly na menšie jednotky, alebo zlúčiť moduly do väčších celkov (napríklad celú vrstvu reprezentovať jedinou jednotkou). V prvom prípade bol výsledok podobný neurónovej sieti (takto to spravili napríklad Rosenblatt a Payton), v druhom prípade z toho vzišli tzv. behaviorálne systémy (napríklad Arkin). Našu architektúru možno tiež považovať za derivát subsumpčnej architektúry a to derivát veľmi blízky. Uvedené problémy I. a II. však máme vyriešené. Problém III. – dátovú fúziu – nepokladáme za šťastné riešiť, lebo každé také riešenie dramaticky obmedzí charakter dát, ktoré si moduly môžu vymieňať – spravidla na reálne čísla. To je podstatne limitujúcejšie než absencia mechanizmu dátovej fúzie, ktorá sa vždy dá riešiť pridaním modulov, ktoré ju realizujú.

Modelovanie mysle

Posledným významným zdrojom pre našu prácu sú nekonecionistické modely ľudskej mysle. Budeme sa opierať o dva z nich: Fodorov model mysle a Minského spoločenstvo mysle („Society of Mind“).

Fodor svoj model založil na pozorovaní nezávislosti poškodenia špecifických funkcií mozgu pri rôznych úrazoch. Pokiaľ sa stáva, že pri jednom poranení sa zachovala funkcia A a bola poškodená funkcia B a pri druhom poranení to bolo naopak, znamená to, že funkcie A a B sú na sebe nezávislé. Podľa miesta poškodenia sa dá ďalej usúdiť z niekoľkých prípadov poškodenia určitej funkcie, či je realizovaná konkrétnou anatomickou časťou mozgu alebo je rozptýlená. Na základe takýchto pozorovaní Fodor dospel k tomu, že myseľ sa skladá jednak z modulov, ktoré realizujú špecifické činnosti (často špecifickejšie než by sme normálne pripustili – napríklad rozpoznávanie tvárí) a z centrálneho systému, ktorý pracuje holisticky (Obrázok č. 16). Hoci dobre vedel, že aktívne tkanivo mozgu sa skladá z neurónov, pripisoval modulom symbolovo-výpočtové schopnosti, ich konekcionistickými vlastnosťami sa nezaoberal. Okrem doménovej špecifickosti modulom pripísal niekoľko pre nás zaujímavých vlastností ako charakteristickú rýchlosť spracovania, plytký výstup, informačnú uzavretosť a fixnú architektúru (vlastnosti ako napríklad mandatórnosť[5] sú pre naše potreby nepodstatné). V našej práci budeme používať podobný nekonekcionistický prístup pre modelovanie mysle.



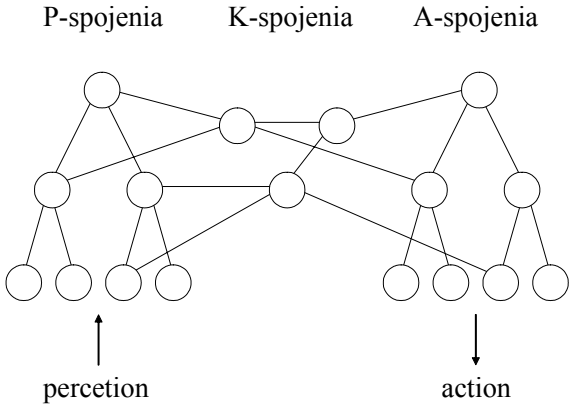
Obrázok č. 16 Fodorov model mysle

Marvin Minsky použil podobne ako Fodor pri opise mysle jednotky vyššej úrovne opisu než neuróny. V Society of Mind ich nazýva agentami, v neskorších prácach zdrojmi („resources“). Spojenia medzi agentami delil na tri druhy (P-spojenia, K-spojenia, A-spojenia – ako

perception, knowledge a action), v žiadnom prípade však nepredpokladal, že by v mysli bol nejaký špeciálny modul kognície, ako by sa z tohto rozdelenia mohlo povrchne zdať (Obrázok č. 17). Podobne ako u Fodora je charakter základných jednotiek skôr symbolovo-výpočtový než konekcionistický.

Modelovaním špecifických častí mysle a ich vývoja sa Minsky snažil vysvetliť výsledky Piagetových experimentov s deťmi a tým odhaľovať aké štruktúry K-spojení sa v mysli nachádzajú. Spomína ich okolo osemnásť, z nich pre nás sú dostatočne popísané a zaujímavé nasledovné (v práci ich budeme nazývať Minského štruktúry):

- Polynémy – na základe predmetu aktivujú atribúty, ktoré sa k nemu viažu (počujem o jablku – aktivujem, že tvar okrúhly, vôňa príjemná, chuť výborná)
- Melanémy – na základe atribútu aktivujú predmety ktoré ho majú (počujem červená – aktivujem krv, paradajku, zastavenie na križovatke)
- Pronómy – ukazujú na objekty, sú to premenné (pronóm „čo-hľadám“, môže chvíľu obsahovať spojenie na „papuče“, inokedy na „okuliare“)
- Scripty – vykonávajú postupnosť krokov (script „preložiť X na Y“ aktivuje postupne „nájdi X“, „uchop X“, „zdvihni X“, „nájdi Y“, „polož „X“, „pusti X“)
- Rámce – obsahujú informáciu usporiadanú podľa príbuznosti, umožňujú meniť aktuálne spracúvaný objekt a zachovať pritom kontext (mapa prostredia, v ktorej si okrem toho, čo vidím, pamätám aj čo je vľavo a vpravo; keď potom urobím krok vľavo, budem vidieť to, čo predtým bolo vľavo a to, čo som videl, sa stane tým, čo mám teraz vpravo)
- Memorizery – učia sa kontext na základe určitého kľúča; keď sa potom objaví kľúč, obnovia kontext – sú to tzv. myšlienkové skratky (susedia - alkoholici majú zase čo piť, tak to mi určite niekto opäť vykradol vínnu pivnicu)
- Recognizery – rozpoznávajú na základe atribútov predmet, sú duálne k polynémom (vidím, že tvar je okrúhly, vôňa príjemná, farba červená, povrch tvrdý, to bude jablko)



Obrázok č. 17 Minského model mysle

Na oboch modeloch nám vyhovuje, že
- sú relatívne prízemné (nesnažia sa vysvetliť všetko)
- sú odvodené z pozorovaní a experimentov
- používajú vyššiu úroveň opisu než neuróny
- naša architektúra je relatívne vhodná na ich implementáciu

Bibliografická poznámka

Prehľad multiagentových systémov podáva niekoľko monografií, najnovšie je uznávaná [Wooldridge 2002], ktorá sa však zameriava hlavne na inteligentné agenty. K zhutnenej podobe sa možno dostať v článku [Jennings 2000]. Bližšie k nášmu pochopeniu je monografia [Ferber 1999]. Z „domácej" literatúry sa problematikou MAS zaoberajú skriptá [Kubík 2000] a zborník [Luck – Mařík – Štepánková – Trappl 2001]. Pekný článok, ktorý vystihuje záludnosti tvorby MAS vyplývajúce z povahy ich modularity (a tie sú plne platné i pre túto prácu) je [Wooldridge and Jennings 1998]. Vyčerpávajúci a dodnes platný prehľad rôznych vetiev MAS s popisom rozdielov medzi nimi podáva [Nwana 1996].

Jazyk LINDA je popísaný v historickom článku [Gelernter 1985]. Jeho moderné aplikácie presadzuje najmä Paolo Ciancarini, viď. napr. [Ciancarini – Rossi 1997]. Java Space obsahujú práce pojednávajúce o technológii Java Jini [Waldo 2001].

SRR model je možné nájsť v každej slušnej knihe o systémoch reálneho času, či o „concurrent" systémoch všeobecne, napr. [Snow 1992]. My sme samozrejme čerpali vedomosti o tomto modeli hlavne z technickej dokumentácie k operačnému systému QNX4.

Middlewarové technológie sú všeobecne známe. Z prístupnej literatúry možno nájsť základ na ktorom stoja v [Naik 1999], zvyšok v špecializovanej literatúre pre jednotlivé druhy middlewaru. Popis middlewarov CORBA a RMI možno nájsť i v populárnej [Eckel 2000], najmä však možno čerpať z technických manuálov, k Jave, k C++, k linuxu a pod.

Subsumpčná architektúra bola postupne profilovaná v sérii článkov [Brooks 1986a] [Brook 1986b] [Brooks 1989] a finálnu podobu nadobudla v [Brooks 1991]. Sumár týchto prác s odstupom času prezentuje [Brooks 1999] a [Brooks 2002]. Kritiku subsumpčnej architektúry a konkurenčnú architektúru založenú na zjemnení štruktúry modulov podáva [Rosenblatt - Payton 1989]. Naopak, architektúru založenú na moduloch o veľkosti celej vrstvy – tzv. behaviorálnych modulov prezentuje napríklad [Arkin 1998]. Okrem toho samozrejme veľký počet ďalších prác, nakoľko subsumpčná architektúra má značný ohlas. V domácej komunite tento ohlas predstavuje hlavne [Kelemen 1994].

Informácie o Fodorovom modeli mysle sme čerpali hlavne z prác Dušany Rybárovej, ktoré sú súčasťou napr. [Rybár 2002], ktoré zase vychádzajý z [Fodor 1983]. Minského model je opísaný vo všeobecne známej knihe [Minsky 1986]. Tento model je ďalej rozvíjaný samotným autorom [Minsky 2004], ako aj ďalšími výskumníkmi [Singh 2003].

Samozrejme množstvo poznatkov čerpáme v tejto práci aj z monografií pokrývajúcich celé pole umelej inteligencie. Z nich dominantné miesto zaujíma [Nilsson 1997], ktorá je jedinečná podaním umelej inteligencie skrz optiku „agentov". Z obdobnej „domácej" literatúry sme použili [Kelemen 1992], [Návrat 2002] a [Mařík – Štepánková – Lažanský 1993 - 2003].

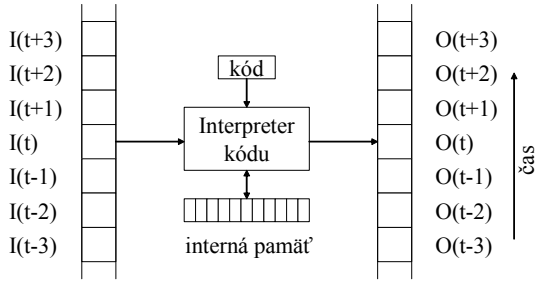
[4] Brooks uvádza stelesnenosť, emergenciu, interakciu a hierarchiu ako štyri postuláty, situovanosť definuje pred nimi ako jeden z dôležitých pojmov.
[5] Nemožnosť ovládať moduly vôľou

III. Východzie motivácie

Prv než sa pustíme do technického návrhu špecifickej architektúry na tvorbu inteligentných systémov, uvedieme niekoľko argumentov, prečo by niečo také mohlo byť osožné. Zameriame sa teda na rozdiely, ktoré prináša zameranie sa na tie vlastnosti, ktoré budeme našou architektúrou podporovať.

Interaktívnosť

Základnou vlastnosťou každého systému, o ktorom budeme v tejto práci uvažovať, je jeho interaktívnosť, teda to, že beží nekonečne dlho a priebežne prijíma svoj vstup a produkuje výstup. Že takýto pohľad na vec prináša niečo nové, môžeme ľahko nahliadnuť už keď zoberieme triviálny prípad keď je systém monolytický (jeho kód nie je modulárny). Kód takéhoto systému beží v nekonečnej slučke a v každom prechode cyklom na začiatku načíta ďalší vstup a na konci vyprodukuje ďalší výstup (Obrázok č. 18).



Obrázok č. 18 Interaktívny systém

Ako príklad triviálneho požadovaného správania môžeme uviesť výpočet plávajúceho maxima, či sumy, teda požadujeme aby výstup v kroku t zodpovedal maximu či sume zo vstupov z krokov t, t-1, t-2, ..., t-k, kde k je veľkosť tzv. okna. Budeme požadovať absolútnu presnosť riešenia. Ako výpočtovú zložitosť riešenia budeme uvažovať počet operácii vykonaných v jednom cykle.

Ľahko nahliadneme, že sumu celých čísel môžeme realizovať v čase O(1). Stačí totiž vždy pripočítavať číslo, ktoré pribudne do okna a odpočítavať číslo, ktoré z okna vypadáva. Samotné okno možno udržiavať ako cyklický buffer, čo nežiada viac ako konštantný počet operácii, hoci v pamäti zaberá O(k). Celé riešenie môžeme zaznamenať nasledovne (zložitosť pritom počítame len z vnútra nekonečného cyklu cez časové okamihy)[6]:

```
for (j=0; j<k; j++) w[j]=0;          // inicializujeme okno
j = 0;                               // j bude v okne ukazovat aktualny okamih
s = 0;                               // inicializujeme sumu
for (t=0; true; t++) {               // prechadzame do nekonecna cez casove okamihy t
   s = s - w[j];                     // odratame prvok, ktory z okna vypadava
   w[j] = i(t);                      // vlozime do okna novy prvok
   s = s + w[j];                     // pripocitame vlozeny prvok k sume
   j = j + 1; if (j >= k) j=0;       // pousunieme sa v okne
  o(t) = s;                          // vratime sumu
}
```

Naproti tomu maximum celých čísel možno spočítať len v čase O(log k). Okno budeme opäť reprezentovať ako kruhový buffer, okrem toho však nad ním vybudujeme binárny strom, v ktorom sa maximum šplhá nahor. V rodičovskom vrchole teda budeme uchovávať maximum z dcérskych vrcholov. Pre jednoduchosť budeme predpokladať, že okno je dĺžky $k = 2^m$ a čísla sú pozitívne (nulu použijeme na reprezentáciu prázdneho miesta). Číslo, ktoré z okna vypadáva, odstránime zo stromu tým, že poopravujeme požadovaný vzťah medzi rodičom a dcérami po celej ceste od odstráneného prvku nahor. Podobne číslo, ktoré do okna pribúda, budeme odspodu vymieňať za menšie čísla, až sa nedostane na správnu pozíciu. V strome je tak vždy číslo v ľubovoľnom vrchole maximom všetkých čísel v jeho podstromoch (to je sledovaný invariant). Číslo v koreni je potom požadovaným maximom. Algoritmus, používajúci podobné štruktúry dát ako triediaci algoritmus tree selection, môžeme zaznamenať nasledovne:

```
// inicializujeme okno w[k..2*k-1] s binarnym stromom w[1..k-1]
for (j=1; j<2*k; j++) w[j] = 0;
j = k;
// prechadzame do nekonecna cez casove okamihy t
for (t=0; true; t++) {
   // odstranime prvok vypadavajuci z okna
   w[j] = 0;
   for (p = j; p > 1; ) {                    // p je uzol stromu
      r = p / 2;                             // r je jeho rodicovsky uzol
      q = r * 2 + 1 - (p % 2);               // q je jeho sestersky uzol
      if (w[p] > w[q]) w[r] = w[p];
      else w[r] = w[q];
      p = r;
   }
   // vlozime novy prvok
   w[j] = i(t);
   for (p = j; p > 1; ) {                    // p je uzol stromu
      r = p / 2;                             // r je jeho rodicovsky uzol
      q = r * 2 + 1 - (p % 2);               // q je jeho sestersky uzol
      if (w[r] > w[p]) break;                // uz sa vysplhal dostatocne vysoko
      w[r] = w[p];
      p = r;
   }
   // vratime maximum
   o(t) = w[1];
   // posunieme sa v okne
   j = j + 1; if (j >= 2*k) j = k;
}
```

Z hľadiska zložitosti je zrejmé, že táto závisí od výšky stromu. V prípade, že dĺžka okna je $k = 2^m$, je tento strom dokonale vyvážený, takže výška stromu je logaritmus dĺžky okna, z čoho dostávame ohlásených O(log k).

Vidíme, že v porovnaní s klasickým výpočtom sumy a maxima (obe O(k)), majú tieto problémy pri interaktívnom pohľade rozdielnu povahu. Odlišuje ich znovupoužiteľnosť výpočtu na riešenie podobného - posunutého problému. Nie je teda jedno či opakovane púšťame rezultatívny algoritmus, alebo bežíme nekonečný interaktívny algoritmus.

Hoci predpokladáme konečnú veľkosť internej pamäte systému[7], okno do minulosti, ktoré môže interaktívny systém zachytávať, nemusí byť tak striktne konečné ako v prechádzajúcom príklade. Systém môže integrovať všetky vstupy, ktoré od začiatku dostal na

vstup, len tým starším musí prisúdiť natoľko menšiu váhu aby bol výsledok integrácie konečný. Typickou ukážkou je výpočet mienky. Systém dostáva na vstup či určitá vec práve zafungovala alebo nezafungovala a má vypočítať či dlhodobo funguje. Pozitívny vstup *v(t)* budeme kódovať ako 1, negatívny ako -1. Vnútorne budeme počítať mienku *m*, ktorú počítame ako $m(t) = 0.9m(t-1) + v(t)$. Mienka teda kolíše od -10 po 10. Na výstup budeme dávať „funguje“, ak je mienka nad *8.5*, „nefunguje“, ak mienka pod *-8.5* a „neviem“ inak. Systém, ktorý má v sebe zahrňovať potrebnú kontinuitu, by mal byť prešpikovaný takýmito trikmi.

Podobne kód interpretovaný v interaktívnom systéme nebude nedeterministický, za to však má veľký význam aby mohol byť pravdepodobnostný. Pravdepodobnosť totiž jednak pomáha vymaniť systém zo stavu zacyklenia, jednak dokáže nahrádzať pamäť. Pre prvý prípad uvažujme, že náš systém sa musí rozhodnúť či zabočiť vľavo alebo vpravo. Potenciálne sa dá o tom uvažovať, ale toto uvažovanie môže vyžadovať potenciálne nekonečný výpočet (napríklad kvôli dokazovaniu platnosti určitej logickej formuly na báze rezolvencie). Pustíme ho teda, ale ak nedoráta pred určitým časovým limitom, zastavíme ho a hodíme si radšej mincou. V druhom prípade zvažujme, že systém potrebuje pracovať s niečím ako netrpezlivosť. Môže si takúto „emočnú“ premennú počítať, ale rovnako dobre mu bude fungovať, keď bude netrpezlivý s istou pravdepodobnosťou. V neposlednom rade dokáže pravdepodobnosť nahrádzať nedeterminizmus a to v tom prípade, keď je vysoká pravdepodobnosť, že náhodne generovaná možnosť je správnym riešením. Dôsledkom toho bude počet pokusov po neúspešnej voľbe malý a akceptovateľný.

Význam súčasnej pozície interaktívnych systémov v informatike sa dá nádherne vyjadriť podľa epistemologickej koncepcie [Kvasz 1998] ako dôsledok tzv. ruptúry typu vizualizácia. Programy na počítači boli zavedené ako nedokonalý fyzický obraz hromadných, determinatívnych a rezultatívych algoritmov. Dochádza však k pochopeniu, že medzi programami možno nájsť aj také, ktoré klasickému pojmu algoritmu nezodpovedajú. To môže viesť k zavedeniu nových pojmov, ako je napríklad interaktívny systém. Podobne napríklad Descartes zaviedol grafy, aby vizualizoval polynómy a potom zistil, že vie nakresliť graf exponenciálnej funkcie, hoci ju nevie prostriedkami vtedajšej matematiky symbolicky vyjadriť. Z tohto uhla pohľadu interaktívne systémy ležia mimo tradičnej teoretickej informatiky sústredenej na vypočítateľnosť a formálne modely výpočtových zariadení. Všetky ich doterajšie formálne modely (napríklad eko-gramatické systémy [Csuhaj-Varjú – Kelemen – Kelemenová – Paun 1997]) sa na nich dokázali pozrieť len tak, že v konečnom dôsledku generujú nejaký jazyk, teda že niečo počítajú. Snažili sa ich teda vopchať do rámca súčasných symbolických prostriedkov informatiky. Potrebujú ale oveľa viac: potrebujú prelomiť škrupinu vypočítateľnosti a nahradiť pojem jazyka niečím o dimenziu bohatším, nejakým „hovorením“. Obrazne povedané súčasná teoretická informatika má tendenciu skúmať, čo nejaký interaktívny systém môže povedať (jazyk), miesto toho, čo kedy povie („hovorenie“). Na zavedenie tohto bohatšieho pojmu do formálnych prostriedkov informatiky ešte len čakáme.

Práca v reálnom čase

Základnou vlastnosťou interaktívneho systému, ktorý je prepojený s fyzickým prostredím je, že pracuje v reálnom čase. Pod týmto pojmom rozumieme, že reakčný čas systému (rozdiel časov vstupu a výstupu, ktorý už v sebe zohľadňuje vstupnú informáciu) bude zhora možné ohraničiť nejakým rozumne malým limitom, ktorý systém dlhodobo dodržuje. Implementácii takéhoto systému výrazne napomôže, pokiaľ sa pri nej opierame o OS reálneho času (RTOS). Takýto systém zaručuje, že pokiaľ dva procesy budú pripravené na vzájomný prenos dát, tento prenos prebehne do určitého časového limitu (tento limit sa nazýva latencia). V takomto systéme

máme zabezpečené, že pokiaľ odštartujeme nejakú namáhavú operáciu (napríklad kopírovanie veľkého množstva dát), nepozastaví to beh ostatných procesov, ale samo sa spomalí natoľko, aby zvyšok mohol fungovať. Samozrejme len do tej miery, kým nevyčerpáme kapacitu procesora, teda kým možno plánovanie procesov utriasť tak, že sa všetko načas stíha. Kľúčovým tu je schopnosť dostatočne rýchlo prepínať procesy v procesore. V rýchlosti tohto prepínania dosahujú RTOS voči bežným OS (Linux, Windows) pomer zhruba 100:1. Našťastie Moorov zákon nám neustále pridáva na výkone počítačov natoľko, že dnes si aj obyčajné OS môžu trúfnuť na úlohy, ktoré by pred desiatimi rokmi nešli bez RTOS vôbec zvládnuť.

Komplexnosť

Ako stúpajú nároky na vytvárané umelé systémy, vynárajú sa nové metódy, pomocou ktorých ich možno pokryť. Pritom tieto nároky môžu dosiahnuť hranicu, za ktorou je požadované správanie tak rozsiahle, že požiadavky nemožno nadefinovať na základe štruktúry systému, iba na základe jeho požadovaného správania, či na základe testov, v ktorých má obstáť. Napríklad môžeme od textového procesora požadovať takú automatickú opravu textu, aby opravil preklepy a aby pritom mylne nemenil to, čo napíšeme správne. Takéto zadanie možno ľahko vysloviť a dokonca tým priamo definujeme model použitia (use-case) a testovaciu procedúru. Nesmierne obtiažne však bude premietnuť toto zadanie do návrhu štruktúry systému bez toho, že by sme začali samotný proces implementácie. Dôsledok tejto vlastnosti problému – ktorú budeme nazývať komplexnosťou – vidno aj vo fungovaní editora v ktorom píšem tento text. S pribúdajúci verziami sa jeho kvalita zlepšuje, ale to len podčiarkuje jeho nedokonalosť. Komplexný systém môžeme teda zadefinovať ako taký systém, ktorý je nevyhnutné vyvíjať postupne z jednoduchších verzií.

Komplexné systémy sú charakteristické práve pre riešenia úloh umelej inteligencie a umelého života, oblastí ktoré sú zadefinované podobným spôsobom: ich úlohou je vytvárať také umelé systémy, ktoré majú vlastnosti podobné systémom živým – v prvom prípade je to inteligencia, v druhom život, nič menej obe sú priamo neredukovateľné, komplexné.

Lákavou možnosťou tvorby komplexných systémov je nepoužiť analytickú, ale syntetickú metódu – nechať systém, nech za poskladá, naladí alebo vyvinie sám. Máme na to viac možností (ale ich reálna aplikovateľnosť nie je bez limitu): napríklad adaptívny mechanizmus neurónovej siete alebo evolvovanie určitej štruktúry genetickým algoritmom. Alternatívou je použitie určitého algoritmu manipulujúceho dáta, ktoré potom ladí vývojár – napr. prázdneho expertného systému do ktorého vkladáme pravidlá. V našom prístupe pôjdeme ešte k menej špeciálnej forme tradičných systémov, nebudeme disponovať adaptívnym ani nijakým iným univerzálnym algoritmom, iba budeme definovať univerzálnu povahu štruktúry systému. Jedine tým sa budeme odlišovať o tradičného programovania, kde je už čokoľvek dovolené, ale kde sa zase niet čoho zachytiť.

Dôležité je uvedomiť si, že aj riešenie komplexného problému bude len postupnosťou inštrukcií v strojovom kóde. Túto by sme teoreticky boli vždy schopní vytvoriť bez akejkoľvek šikovnej techniky, prakticky je to však načisto nemožné. Je to podobné známemu príkladu, keď sa od opice ťukajúcej do písacieho stroja očakáva, že napíše niektorú Shakespearovu drámu. Príčinou, prečo je to tak, sú obmedzenia našich ľudských schopností ako čas, peniaze, chybovosť vzájomnej komunikácie v tíme, ale predovšetkým ohraničenosť počtu vecí u ktorých sme naraz schopní strážiť vzájomné vzťahy a samotná poruchovosť tohto stráženia. Ľudské bytosti potrebujú nájsť správnu úroveň opisu, aby vzťahy medzi opísanými entitami dokázali preliezť úzkymi hrdlami v ich mozgoch. Práve jednu takúto úroveň opisu – založenú na reaktívnych agentoch a nepriamej komunikácii medzi nimi – v práci predkladáme.

Snáď jedného dňa memetická teória položí teoretické základy vysvetlenia našich ohraničených schopností a umožní objektívne vyhodnotiť softwarové architektúry z hľadiska ich použiteľnosti pre komplexné systémy. Do tých čias bude hodnotenie založené len na tom, čo sa komu s jeho architektúrou podarí poskladať.

Inteligencia

Z definícii inteligencie, ktoré používa psychológia, sa nám najviac hodí definovať inteligenciu ako schopnosť systému adekvátne reagovať na zmeny odohrávajúce sa v jeho prostredí. Tu si hneď treba uvedomiť, že ani od „inteligentného“ systému nemožno požadovať adekvátnu reakciu na akúkoľvek zmenu jeho prostredia. Musíme množinu spomínaných zmien prostredia obmedziť len na určitú triedu a snažiť sa, aby táto bola netriviálna a čo najširšia. Bude závisieť od rozsahu a „tvaru“ tejto triedy a od charakteru správania, ktorého produkciu od nášho systému požadujeme, či bude daný systém v reále uznaný za inteligentný alebo nie. Môžeme sa na celú vec pozerať tak, že pre každú požadovanú funkčnosť systému existuje určitá latka, ktorú musí šírka a „tvar“ spomínanej triedy preskočiť, aby bol systém, ktorý sa ňou vyznačuje považovaný (ľuďmi) za inteligentný. Existuje teda nejaká spojitá veličina, ktorá je vlastnosťou systému, ktorá ak prekročí určitý limit, systém je inteligentný, inak nie je. Spomeňme napríklad program ELIZA [Gál – Kelemen 1992], ktorý dokázal hrať úlohu psychiatra pri rozhovore s pacientom[8] tak dobre, že ho pacienti jednoznačne označovali za skutočného psychiatra. Pritom tento program napíše každý študent umelej inteligencie za jeden večer. To je vysvetliteľné tým, že spomínaná latka pre správanie psychiatra je u jeho pacientov tak nízko, že ju aj taký jednoduchý program ako ELIZA dokáže preskočiť. Takéto nazeranie na vec je zaujímavé preto, lebo to, čo týmto „neinteligentným“ systémom chýba, je schopnosť pokryť adekvátnymi reakciami netriviálne rozsiahlu triedu zmien v ich prostredí. Spomínanou spojitou veličinou nie je teda nič iné než spomínaná komplexnosť.

Medzi inteligenciou a komplexnosťou je teda určitý vzťah a nám sa do istej miery ponúka možnosť definovať jedno podľa druhého: inteligencia je dvojhodnotovou kategorizáciou komplexnosti. Teda pre každý účel existuje určitý súbor funkcií a správaní, ktoré do seba musí systém pojať, aby bol považovaný za inteligentný. Táto kategorizácia je subjektívnou záležitosťou ľudí, inak povedané inteligencia je len abstraktný pojem, ktorý si vymysleli ľudia ako skratku za „dostatočne komplexný pre úlohu XY“.

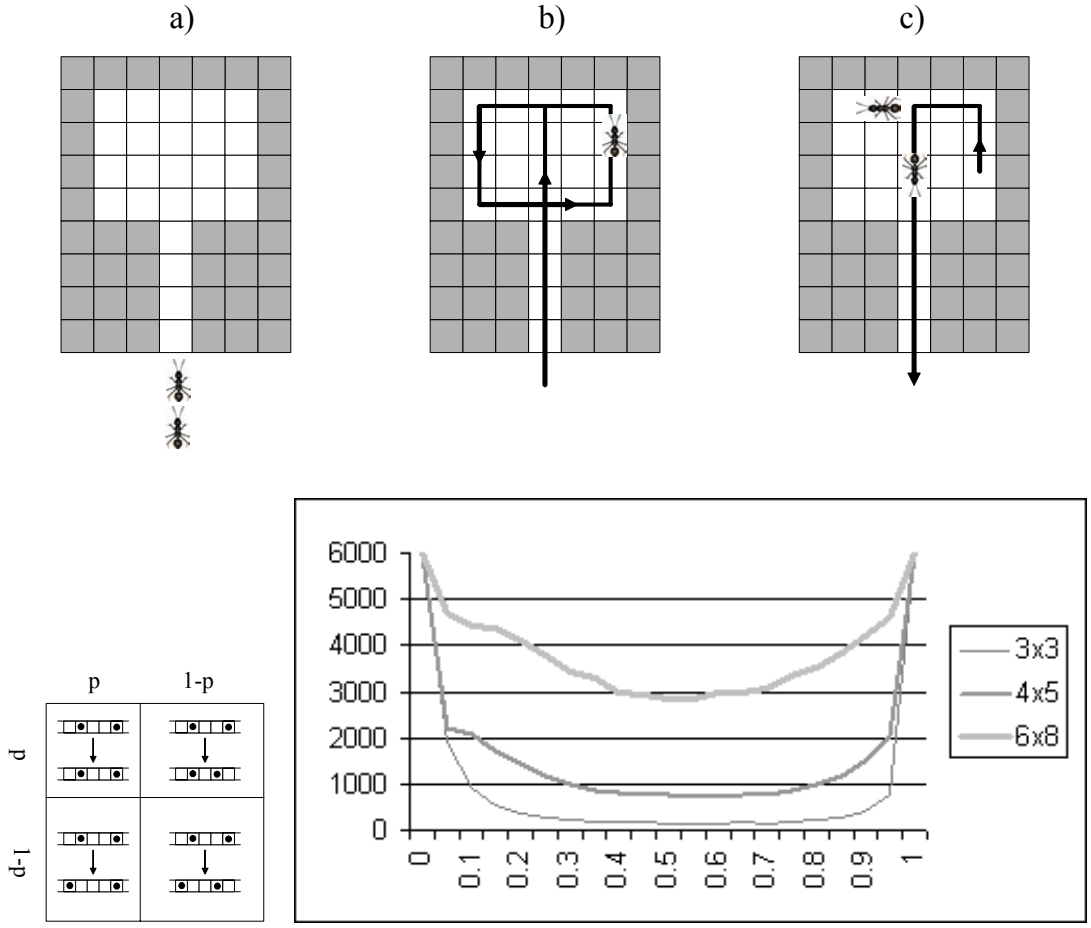
Ak chceme teda skonštruovať inteligentný systém, musíme hľadať také implementačné postupy, ktoré umožňujú implementovanému systému dosiahnuť čo najvyššiu komplexnosť.

Emergencia

Pod emergenciou rozumieme jav, keď systém zložený z určitých stavebných jednotiek získa vďaka ich vzájomnej interakcii vlastnosť, ktorú tieto jednotky nemajú [Holland 1998]. V triviálnom prípade nemá táto samovznikajúca vlastnosť pre stavebné jednotky ani zmysel. Bolo by však riadne pritiahnuté za vlasy tvrdiť, že napríklad schopnosť Euklidovho algoritmu počítať nsd emerguje zo spojenia jeho jednotlivých príkazov. Niekde tu sa však začína výskum emergencie. Napríklad skupinami gramatík generujúcich konečný jazyk, ktoré pôsobia v rovnakom prostredí sa dá generovať trieda bezkontextových jazykov – nakoľko každá taká gramatika zrealizuje pravidlá pre príslušný neterminál [Kelemen – Kelemenová 1992]. Triviálne, ale už keď pridáme časovanie, je to zrazu veľmi zaujímavé: dostaneme aj niektoré kontextové jazyky. A to ešte nemáme interaktívny systém, ale čisto statické prostredie.

V priebehu posledného desaťročia prebehol pomerne bohatý výskum, prinášajúci napríklad simuláciu vrátane dynamiky prostredia v tzv. ekogramatickom systéme [Csuhaj-Varjú – Kelemen – Kelemenová – Paun 1997]. Postupne sa objavujú rôzne mechanizmy, ktoré sú za emergenciu zodpovedné [Kelemen 2001a]. Okrem časovania (aktívnosti stavebných jednotiek) je to pravdepodobnosť (ich pôsobenia), neurčitosť (ich činnosti), a zrejme ďalšie o ktorých zatiaľ nevieme. Ďalej sa ukazuje, že emergencia sa dá dosiahnuť nad heterogénnymi, ale aj homogénnymi jednotkami (čo nijako neprekvapuje, keď to prirovnáme k celulárnemu automatu [Csontó – Palko 2002]).

Ako príklad emergencie si uvedieme jeden z vlastnej dielne. Obľúbenou metaforou používanou v MAS je mravenisko (vidno to aj z obalov monografií, na ktorých spravidla nejakého mravca nájdete). Uvažujme pascu na mravce obdĺžnikového tvaru, do ktorej vedie jediná prístupová a zároveň úniková cesta (Obrázok č. 19a).



Obrázok č. 19 Príklad emergencie

Vedeli by sme napísať taký riadiaci program mravčeka, že keď dnu pustíme jedného nikdy von nevyjde, ale keď tam pustíme dvoch, jeden z nich vyjde[9]? Zatiaľ najlepšie riešenie tejto úlohy čo sa mi dostalo do rúk pochádza od mojej študentky Márie Nanásiovej: mravček pri náraze na prekážku zabočí doľava a inak ide rovno, s určitou pravdepodobnosťou *p* však nevykoná krok, ale ostane stáť. Keďže sa točí vždy len vľavo, tak pokiaľ bude sám, nemá šancu dostať sa von (Obrázok č. 19b). Avšak keď budú dvaja, môže sa stať, že jeden druhému urobí popri stene

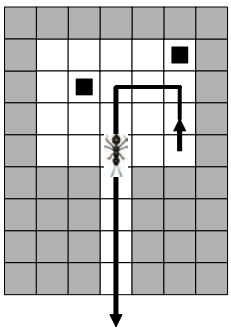
protiľahlej ku vchodu pomyselnú prekážku a otočí ho tak rovno do únikovej chodby (Obrázok č. 19c). Priemerný počet krokov mravčeka, za ktoré sa dostane jeden z nich von, zobrazuje Obrázok č. 19 dolu vpravo. Toto číslo závisí od rozmerov pasce (uvádzame tri veľkosti), nezávisí od časového odstupu medzi vbehnutiami mravcov (ak nie je rádovo väčšie než obvod pasce, samozrejme) a keďže závisí od *p(1-p)*, čo je pravdepodobnosť, s ktorou jeden mravec dobieha druhého (jeden stojí, druhý ide – viď Obrázok č. 19 dolu vľavo) najlepšie výsledky dáva pre *p=0.5*.

Heterogénny príklad emergencie dostaneme, keď do tejto pasce pustíme jedného ľavotočivého a jedného pravotočivého mravčeka s občasným státím. Ani jeden z nich nevie vyjsť von, pokiaľ je sám. Ak ich tam však dáme spolu, jeden z nich po čase vyjde.

Brikoláž

Brikoláž[10] je jav podobný emergencii, respektíve by sme ju mohli považovať za jej špeciálny prípad. Ide o stav, kedy sa pri vložení systému do zmenených podmienok ukáže, že nejaký mechanizmus, ktorý v ňom slúžil na určitý účel, poskytuje novú vlastnosť. Vďaka nej môže byť systém schopný riešiť situáciu pre ktorú nebol určený. Hoci bol pre ňu zjavne predpripravený, objavenie sa takejto vlastnosti môže byť načisto prekvapujúce.

Pre uvedenie triviálneho príkladu môžeme nadviazať na predchádzajúci príklad s mravčekom a pascou. V mravčekovi funguje mechanizmus, ktorý ho bráni pred narážaním sa do prekážky – ten ho pri prekážke otočí doľava. Ak okrem toho obsahuje už len pravidlo, že inak ide rovno, z pasce nikdy (sám) nevyjde. Ak ale zmeníme podmienky a do pasce nasypeme kamienky, môže sa stať, že sa mravček pri odrážaní od kamienkov dostane von. V zmenenom prostredí pracuje teda protinárazový mechanizmus aj ako únikový mechanizmus (hoci nespoľahlivo, nie vždy, ...). Krajší príklad uvedieme neskôr práve v súvislosti s našou architektúrou: systém totiž musí mať určitú štruktúru na to, aby sa brikoláž mohla vôbec objaviť. Podstatou je to, aby aktivačná podmienka určitých jednotiek presahovala aktuálnu potrebu. Napríklad keby bol protinárazový mechanizmus mravčeka aktivovaný podmienkou „na políčku pred nami je okrajová stena“, žiadna brikoláž by nebola. Tá sa objaví len ak je podmienka „pred nami nie je voľné políčko“.



Obrázok č. 20 Príklad brikoláže

Je fakt, že ani emergencii, ani na brikoláži nie je nič zázračného. Dívame sa na ňu ako na jeden z mechanizmov, ktorý dáva systému schopnosť vyprodukovať zo seba to, čo sme doňho explicitne nevkladali (samozrejme my sme to tam vložili, ale implicitne). (Zaujímavé je, že toto implicitné môže byť aj nevedomé a prekvapivé.) Sú aj účinnejšie mechanizmy ako napr. neurónové siete, genetické algoritmy, v istých prípadoch takto dokáže zafungovať inferenčný

mechanizmus expertného systému, ale toto sú z hľadiska programátorského jazyka vždy nejaké špeciálne komponenty, ktoré sa nedajú previazať so samotnou organizáciou kódu systému. Obrazne povedané, vyžadujú používanie špeciálnych súčiastok. Emergenciu a brikoláž však môžeme dostať už pri šikovnom organizovaní obyčajných súčiastok. V našom prípade pôjde o organizáciu nad procesmi či vláknami, ktoré implementujú pomerne jednoduché programy vytvárané obyčajnými vývojovými prostriedkami. Poďme si teda túto navrhovanú architektúru predstaviť.

[6] Toto riešenie sa nedá rozšíť na reálne čísla, z dôvodu numerickej nestability. Pre reálne čísla by bola optimálna zložitosť O(k), v každom kroku by sme museli spočítať všetky prvky okna

[7] Opak by tejto práci (aplikovaná informatika) veľmi nepomohol, ale aj takéto výskumy sa robia a ukazujú, že určité druhy interaktívnych systémov podliehajúce správnej mašinérii, môžu z hľadiska vypočítateľnosti prekonať Turingov stroj [Leeuwen – Wiedermann 2001], [Wätjen 2003]

[8] tzv. liečba rozhovorom

[9] nemáme samozrejme záujem o riešenie typu „počkám kým stretnem iného mravca a potom výjdem von“

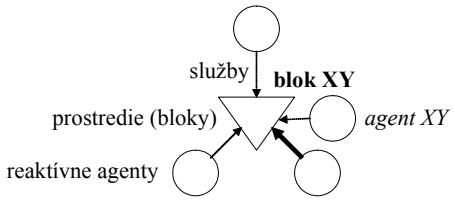
[10] Tento názov si požičiavame z biológie, je známy ako „bricolage“, „tinkering“, „kutilství“, „bastlování“. Prikláňam sa tu k slovenskej verzii tohto slova ktorú zaviedol Ladislav Kováč na seminári Kognitívne vedy

IV. Architektúra Agent-Space

V tejto kapitole predstavíme architektúru, ktorú sme vytvorili za účelom byť vhodným implementačným nástrojom umelých inteligentných systémov a súčasne vhodným nástrojom na modelovanie systémov živých. Jej základným atribútom je absencia akéhokoľvek modulu, ktorý by zodpovedal kognícii alebo bol za ňu zodpovedný. Inteligentné správanie produkuje len na základe šikovnej organizácie pomerne primitívnych modulov. Je preto blízka obyčajným architektúram, ktoré umožňujú vzájomne komunikujúce moduly, bez ohľadu na to, či ide o „concurrent“ alebo „distributed“ programovanie. A nielen, že je blízka, ale je proste jednou z nich. V jej pozadí však stojí istá ambícia použiteľnosti pre systémy umelej inteligencie a to podobného (prízemného) charakteru aké tvoril Rodney Brooks a jeho nasledovníci (viď stranu 27).

Štruktúra a dynamika

Z hľadiska štruktúry je naša architektúra založená na multiagentovom systéme. Tento pozostáva zo spoločenstva reaktívnych agentov, medzi ktorými prebieha nepriama komunikácia, ktorú zabezpečuje špecifická štruktúra zvaná space (do slovenčiny by ho však rozhodne preložil ako prostredie, nie priestor). Agenty medzi sebou nemôžu komunikovať priamo a nie sú schopné ani vzájomnej referencie na základe nejakého identifikačného čísla alebo mena. Môžu iba zapisovať a čítať pomenované dáta uložené v prostredí (Obrázok č. 21).



Obrázok č. 21 Architektúra agent-space

Systém pozostáva z reaktívnych agentov (znázorňujeme krúžkom) a prostredia (znázorňujeme trojuholníkom), ktoré im poskytuje služby (znázornené šípkou, ktorej smer vyjadruje aktivitu: teda to, že je to agent, kto svojou žiadosťou inicializuje vykonanie nejakej služby v prostredí). Typ šípky znázorňuje typ služby: prerušovaná čítanie, tenká zápis, hrubá oboje. Názvy blokov sú tučne, názvy agentov kurzívou. Pokiaľ je v schéme viac trojuholníkov, môžu reprezentovať viac blokov jedného prostredia alebo aj viac prostredí, pokiaľ je systém distribuovaný.

Prostredie („Space“) je entita schopná obsahovať potenciálne ľubovoľný počet pomenovaných dátových „buffrov“, ktoré budeme nazývať bloky. Do týchto blokov môžu všetky agenty zapisovať konkrétne dáta a môžu z nich tieto dáta čítať (Obrázok č. 22 vľavo). Na operácie čítania a zápisu blokov pritom treba nazerať ako na služby, ktoré prostredie agentom poskytuje. Vďaka nim dokáže každý agent zanechať v prostredí správu pre iného agenta. Zámerne mu však tento koncept neumožňuje túto správu nasmerovať k jej príjemcovi. Je ponechané výlučne na tomto príjemcovi, či túto správu prevezme. Navyše každý blok môže čítať i zapisovať viacej agentov. Je teda ponechané na návrhárovi systému, aby zabezpečil patričnú kooperáciu všetkých zapisovateľov a čitateľov tohto bloku. Po zapísaní dát do bloku už zapisujúci agent nemôže nijako ovplyvniť, čo sa bude so zapísanou správou diať, ľubovoľný agent znalý jej pomenovania ju môže prečítať, alebo dokonca prepísať iným obsahom. Pre

takúto správu niektorí autori používajú názov stigma (značka) a takýto spôsob výmeny dát potom nazývajú stigmergickou komunikáciou [Valckenaers 2001].

Každý blok môže byť prázdny. Nie je požadované, aby bol vytvorený pomocou nejakej špecifickej operácie. Fyzicky vznikne prvým zápisom, avšak čítať je ho možné už predtým: nevráti sa chyba, ale prázdne dáta. Bloky (spravidla) predstavujú jednoduchú dátovú štruktúru schopnú pojať jeden kus dát - nie sú to fronty („message queue“) či komunikačné kanály („pipe“, „channel“), takže každý zápis zničí dáta zapísané predchádzajúcim zápisom. Naopak, operácia čítania blok nijako nemodifikuje, teda čítané dáta nie sú z bloku odobraté, len okopírované. Dôležitou vlastnosťou dát zapísaných v bloku je ich platnosť („validity“). Do bloku môžu byť zapísané trvalo alebo na obmedzený čas, po vypršaní ktorého sú dáta bez zásahu zvonku z bloku odstránené a blok sa stáva prázdnym. Pritom vypršanie platnosti dát nijako nehľadí na to, či vôbec boli niekým prečítané.

Forma dát uložených v prostredí nie je v rámci architektúry nijako predpísaná, prostredie na ne hľadí ako na sekvenciu bytov. Dáta uložené v prostredí sa svojim spôsobom podobajú na povestný uzlík na vreckovke: ich čitateľ musí vedieť, čo znamenajú, aby im rozumel. Zabezpečiť, aby čitateľ dát týmto dátam aj rozumel, je teda ponechané na návrhárovi systému. Samozrejme, v rámci konkrétnej aplikácie toto možno zabezpečiť zvolením presnejšie vymedzenej formy dát. Obyčajne v bloku býva zapísaná minimálne dátová štruktúra typu záznam (record, struct), môže tam však byť aj tzv. ontologická štruktúra vyjadrená v nejakom reprezentačnom jazyku (napr. v XML). Zapisovaná, čítaná a spravovaná prostredím je však vždy ale ako sekvencia bytov určitej dĺžky. V princípe je možné aj to, aby ju čítali dva agenty a to každý s inou dĺžkou, čo má význam, keď je v bloku štruktúra typu záznam a vyvstane potreba jej rozšírenia: pokiaľ modifikujeme tak, že len pridávame nové položky na koniec štruktúry, budú s ňou vedieť pracovať aj agenty vytvorené pred touto modifikáciou.

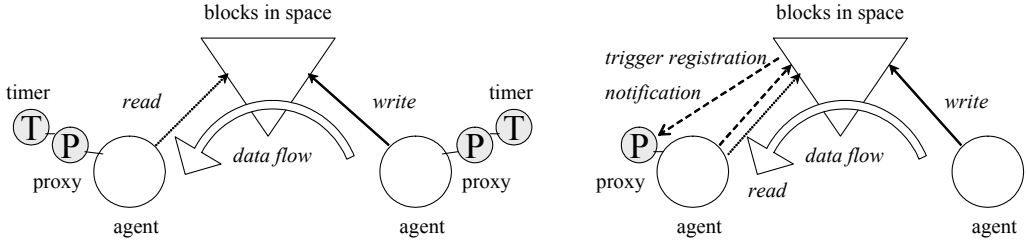
Dôležitý detail procesu prístupu agenta k dátam je, že každá služba prostredia sa vykonáva bež možnosti jej prerušenia inou službou. Nehrozí teda, že sa nad určitým blokom vykoná polovica zápisu, potom jeho čítanie a nakoniec dokončenie zápisu. Tým pádom odpadajú problémy sprevádzajúce prácu so zdieľanou pamäťou. Túto vlastnosť netreba nejako špeciálne implementovať, lebo je daná tým, že operácie zápisu a čítania sú služby, ktoré realizuje prostredie ako server pre klientov. Obsluha klientov prebieha v rámci jediného vlákna (thread) a preto sú všetky služby realizované sekvenčne. Tento prístup je samozrejme možný len vďaka tomu, že služby sú zvolené tak, aby boli dostatočne atomické. Väčšiu námahu dá zabezpečiť, aby bolo možné bez prerušenia vykonať aj celú sekvenciu služieb, čo je pri určitých aplikáciách vhodné.

Operácie zápisu (write) a čítania (read) sú základné služby, ktoré prostredie realizuje, spravidla však nie sú jediné. Ďalšou možnou službou je zmazanie obsahu bloku (delete).Užitočnou službou je taktiež blokovanie zápisov na základe definovanej priority (write s prioritou, delete s prioritou), jej význam uvidíme neskôr. Aj ďalšie operácie sa v reálnych aplikáciách zídu. Tieto slúžia buď na prekonanie fyzických ohraničení techniky alebo na masové operácie s blokmi.

Najdôležitejšou z nich je registrácia triggrov, ktoré poskytujú agentovi upozornenie, že sa určitý blok práve zmenil. Pritom je možné registrovať trigger i na blok, ktorý ešte nebol vytvorený a samozrejme na jeden blok sa môže registrovať viac triggrov. V momente zmeny bloku sú všetky agenty, ktoré majú zaregistrovaný trigger na tento blok, zobudení k činnosti (Obrázok č. 22 vpravo).

Ďalšími službami prostredia, ktoré môžu byť pri reálnych aplikáciách užitočné, sú masové operácie s blokmi ako čítanie všetkých blokov, ktorých meno vyhovuje určitej maske alebo čítanie všetkých neprečítaných blokov, o zmene ktorých bol agent upovedomený notifikáciou. No a v praxi ešte zoznam uzatvárajú služby, umožňujúce prekonať obmedzenia

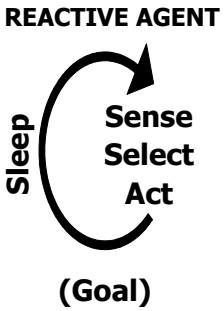
vyplývajúce z výkonu súčasnej techniky, ktoré sa objavujú pri aplikáciách, keď dáta vstupujú do systému z vysokou frekvenciou – nebudeme sa však s nimi v tejto práci zaoberať.



Obrázok č. 22 Dátový tok medzi dvomi agentami

Vľavo je štandardný prípad, keď jeden agent opakovane dáta zapisuje a druhý opakovane číta. Pokiaľ však druhý agent potrebuje spracovať určité dáta ihneď ako sú zapísané, neostáva mu iná možnosť ako požiadať prostredie, aby ho o zápise týchto dát upovedomilo. Slúži na to mechanizmus zvaný trigger. Agent, ktorý si trigger zaregistroval, potom číta zapísané údaje na základe notifikácie. Časovanie a notifikácia sa realizujú v závislosti od platformy, tu sú znázornené pomocou proxy.

Služby prostredia sú štandardné v tom zmysle, že sú nezávislé od aplikačnej oblasti. Všetok aplikačný kód (t.j. taký kód, ktorý je zviazaný s úlohou ktorú riešime) je sústredený mimo prostredia a to v relatívne malých a jednoduchých programoch zvaných **reaktívne agenty**. Z hľadiska štruktúry kódu je reaktívny agent program, ktorý – po určitej inicializácii – vykonáva donekonečna cyklus typu *sense – select – act*. Tento cyklus je vykonávaný s určitou frekvenciou, pričom medzi dvomi prechodmi je určitá pauza (Obrázok č. 23). Vo fáze *sense* agent načíta dáta z prostredia, vo fáze *select* zvolí (vypočíta), ako bude na ne reagovať a vo fáze *act* ich zapíše do prostredia. Pauzu, či fázu *sleep*, zabezpečuje čakanie na timer alebo trigger. Trigger pritom pokladáme len za doplnkový mechanizmus, ktorého hlavnou úlohou je, aby čítajúci agent prijal dáta včas – pokiaľ je to potrebné.



Obrázok č. 23 Reaktívny agent

Takýto proces si dovoľujeme nazvať agentom, nakoľko opakovane **vníma svoje prostredie a na základe toho volí akcie ktoré v ňom vykonáva a sleduje sa tým určitý**, hoci implicitný **cieľ** [11]. Prívlastok reaktívny si tiež zaslúži, už len preto, že nejaký treba použiť na odlíšenie od „na trhu“ prevládajúcich deliberatívnych agentov, ktoré v sebe majú „kognitívnu“ súčiastku (napr. plánovanie). Navyše sa toto použitie zhoduje s ostatnými autormi. Na druhej strane voľba akcií prebieha na základe obyčajného výpočtu, teda výsledok bude naozaj púhou reakciou na stav prostredia a prípadne na vnútorný stav agenta. Pod týmto vnútorným stavom

rozumieme informáciu ktorá v agentovi prežije fázu *sleep*, teda odovzdáva sa medzi dvomi prechodmi cyklom. Pokiaľ sa žiadna takáto informácia neodovzdáva, hovoríme, že agent nemá vnútorný stav a nazývame ho čisto alebo rýdzo („purely“) reaktívnym agentom.

Pri výklade sa teraz dostávame k detailom, ktoré by sme na prvý pohľad mohli ponechať voliteľné, všetky však majú svoje opodstatnenie a boli zaradené na základe dlhodobého uváženia a testovania. Menovite ide o nasledovné vlastnosti:

- *absencia potreby najprv vytvoriť blok a až potom ho používať na zápis a čítanie a s tým spojenú schopnosť agentov čítať nezapísaný blok*
  Túto schopnosť nemá ani jeden derivát jazyka LINDA a je to veľká škoda. Ak totiž musíme blok explicitne vytvoriť, musí byť na to určený práve jeden agent. Pokiaľ blok slúži len na realizáciu dátového toku medzi dvomi agentami, nevadí to. Ale ak sú naň navešané mnohé agenty, musíme nejako zabezpečiť, aby počkali, kým ten jeden vyvolený agent blok zriadi. To sú veľmi nechutné nábehy systému a ešte nechutnejšie reštarty, lebo ich treba buď centrálne riadiť, alebo do každého agenta vkladať mechanizmus, ktorý skúša, v akom stave je prostredie a detekuje, kedy nastáva ten správny čas pre začiatok vlastnej aktivity. Je oveľa lepšie vytvoriť blok pri prvom zápise. Musíme však potom vyriešiť, čo robiť v prípade, keď dôjde k jeho čítaniu pred vytvorením. Tu môžeme zaviesť hodnotu „neznáma“, teda vrátiť agentovi informáciu, že blok ešte nebol vytvorený. Viac sa nám ale osvedčilo používať defaultnú hodnotu, ktorú si každý agent definuje sám: „chcem čítať blok X, a ak sa nedá, vráť mi hodnotu Y“. Hodnotu Y si potom agent môže interpretovať ako potrebuje, či už ako vlastnú hodnotu „neznáma“, default alebo ako bežnú hodnotu, pri ktorej nevyvíja žiadnu aktivitu a podobne.

- *nedefinovanie dĺžky, typu či štruktúry dát, ktoré sa do bloku ukladajú*
  Ďalšou vlastnosťou, ktorou musíme zaplatiť za automatické vytvorenie bloku pri prvom zápise, je nemožnosť definovať pre blok nejaké pevné atribúty. Ak do bloku budú striedavo zapisovať dva agenty dáta rôznej dĺžky, tak sa proste bude meniť dĺžka bloku. Každú informáciu o bloku musia vedieť všetky agenty, prostredie vie iba jeho meno. Programátorovi však nerobí problém takúto koordináciu zabezpečiť.

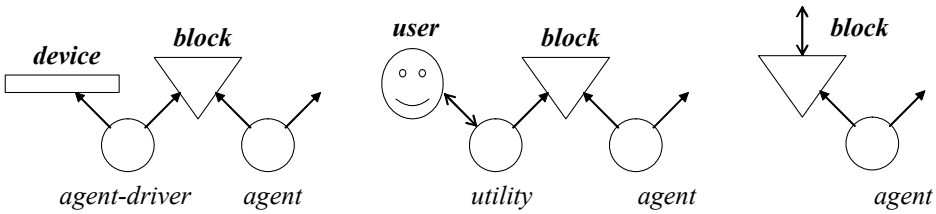
- *nepodporovanie zásobníkového charakteru blokov*
  Z analogických príčin považujeme blok za jednu jednotku, ktorej obsah je prepísaný a stratený pri najbližšom zápise. To sa veľmi odlišuje od tzv. „message queue“, ktoré sa dosť používajú. Urobiť z bloku zoznam je v princípe možné ale veľmi nešikovné, lebo opäť by jeden vyvolený musel povedať, ako je veľký a prostredie by si muselo viesť údaje o všetkých čitateľoch, aby vedelo, čo už z neho prečítali. Preto túto možnosť radšej obetujeme jednoduchosti riešenia.

- *potreba vykonania niekoľkých služieb pre jedného agenta bez prerušenia iným agentom.*
  Túto potrebu ľahko nahliadneme, keby sme napríklad chceli realizovať blok typu zámok („lock“). Na to potrebujeme vykonať tesne po sebe čítanie a zápis, pritom zápisom zamykáme. Či sme to boli my kto zamkol, však zistíme pomocou prečítanej hodnoty – ak už bola zamknutá, neboli sme to my, ak nebola, tak sme ju zamkli pre seba. Podobne sa môže stať, že do prostredia zapisujeme viac blokov, medzi ktorými je určitý vzťah, napr. *x* a *f(x)*. V takom prípade si neželáme, aby niekto mohol prečítať časť nových a starých hodnôt, napr. *x* a *f(y)*. (Sú naopak prípady, kedy to nevadí.)

- *odmietanie priamej komunikácie medzi agentami*
  Každý MAS pozná predovšetkým priamu komunikáciu medzi agentami, keď jeden druhému posiela správu, ktorú nemôže nik ďalší monitorovať. Zákaz takejto komunikácie je z nášho pohľadu kľúčový. Vedie k lepšej decentralizácii systému a umožňuje špeciálny prístup k jeho vývoju (o tomto prístupe diskutujeme v kapitole VI).

Prepojenie na zariadenia a užívateľa

Čo sa týka prepojenia systému vyjadreného v agent-space na fyzický svet reprezentovaný zariadeniami a užívateľom, máme viacero možností. Správna je však len jedna a to tá, ktorá umožňuje ľubovoľnému agentovi monitorovať dáta, ktoré sú vymieňané medzi systémom a svetom. Na to je potrebné rozdeliť riadiaci a komunikačný kód do dvoch agentov, komunikujúcich cez určitý blok. Totiž práve informácia, ktorú riadiaci kód dáva do komunikačného je hodná monitorovania. Keby sme ju nevedeli monitorovať, dostali by sme sa do podobnej situácie ako je to s modulom Status v robotovi ALLEN (viď stranu 30) – museli by sme vopred pripraviť čo budú neskôr nejaké agenty potrebovať. A to je proti princípu vývoja zdola nahor. Práve na rozhraní systému a sveta návrh ALLENa zlyháva, hoci vnútri funguje všetko výborne. Preto pre agent-space stanovujeme presne ako si toto rozhranie predstavujeme, čím sa problém odstráni. Pokiaľ ovládame určité zariadenie, komunikáciu s ním (protokol) bude obstarávať jeden agent – volajme ho driver – a riadiacu logiku iný agent (môže ich byť prípadne viac). Títo budú vzájomne komunikovať cez blok v prostredí, takže sa tu bude dať monitorovať čo sa so zariadením deje (Obrázok č. 24 vľavo). Podobne to bude s užívateľom (Obrázok č. 24 v strede).



Obrázok č. 24 Transducery

V oboch prípadoch budeme znázorňovať prepojenie blokom s obojsmernou šípkou (po vzore subsumpčnej architektúry) (Obrázok č. 24 vpravo). Inšpirovaní Fodorom budeme nazývať tieto konštrukty transducermi.

Základné vlastnosti

V princípe má agent-space tieto základné vlastnosti:

(1) agenty komunikujú výlučne nepriamo, vedia len o blokoch a nemajú žiadnu referenciu na iné agenty

(2) bloky v prostredí netreba vytvárať

(3) bloky v prostredí sa prepisujú bez ohľadu na to, či ich niekto stihol prečítať alebo nie

(4) aplikačný kód je sústredený výlučne v agentoch

Veľa ďalších zaujímavých vlastností z nich vyplýva:

(5) v systéme nemôže vzniknúť deadlock. Jediným „receiverom" v systéme je prostredie a agenty komunikujú s ním. Nie je z čoho deadlock upliesť, pritom dátová výmena môže byť realizovaná medzi ľubovoľnými agentami. To je veľká výhoda napr. oproti pyramidálnej architektúre klient server (pozri stranu 21). Tienistou stránkou tohto usporiadania je, že prostredie je tzv. úzkym hrdlom („bottleneck"). Avšak nakoľko je aplikačne nezávislé, je to úzke hrdlo podobné operačnému systému, databáze a pod., teda relatívne akceptovateľné.

(6) ľubovoľný agent možno kedykoľvek reštartnúť bez rizika, že by iný proces stratil na neho referenciu. Iné agenty žiadnu referenciu nestratia, nakoľko podľa *(1)* žiadnu nemajú. Prostrediu sa to môže stať, pokiaľ má agent zaregistrovaný trigger. Keďže si však tento trigger registruje sám agent, pri reštarte si zaregistruje nový a ten starý nebude potrebovať. Nakoľko nie je problém zistiť, že určitý trigger stratil adresáta (s adresátom zanikne aj proxy, ktoré vlastnil a pri triggernutí takého proxy sa zistí, že neexistuje), prostredie proste také triggre bude rušiť a nič zlého sa nestane. Možnosť reštartu agentov totiž vzhľadom na *(4)* znamená možnosť reštartu ľubovoľnej časti aplikačného kódu. A to je niečo nedostižné pre pyramidálnu architektúru klient-server, lebo pri nej je aplikačný kód aj v serveroch a keď reštartneme server, každý jeho klient (ktorý si pamätá jeho pid) má problém.

(7) ľubovoľný čisto reaktívny agent možno kedykoľvek reštartnúť. Nie každý agent však možno vo všeobecnosti reštartnúť bez následkov na logike správania systému. Agenty totiž môžu viesť (prostredníctvom blokov) komplikovanejší dialóg, stav ktorého môže byť obsiahnutý v ich vnútornom stave. Vnútorný stav sa pri reštarte stráca. Pokiaľ sa však obmedzíme na čisto reaktívnych, môžeme vymýšľať akékoľvek dialógy, tento predpoklad nás donúti uchovávať relevantnú informáciu v prostredí a teda reštartnutý agent pokračuje presne tam, kde predošlý prestal. Vhodnosť používania čisto reaktívnych agentov je dosť prekvapujúca, nakoľko sa ukazuje, že menej je viac. Na druhej strane rovnako dobre stačí ak sa obmedzíme v zložitosti dialógu medzi agentami na dialógy bez stavu. Použitie čisto reaktívnych agentov má však aj iné výhody, ako uvidíme neskôr.

(8) zotaviteľnosť z chýb. Úžitok z ľahkosti, s akou sa dá ľubovoľný agent reštartnúť, spočíva práve v možnosti, že sa zrúti kvôli zriedkavo sa vyskytujúcej internej chybe. Nie je totiž problém kontrolovať procesy a v prípade, že nejaký nájdeme spadnutý, reštartnúť ho. Problémom je zabezpečiť, aby po takom reštarte fungoval.

(9) konfigurovateľnosť. Reštarty i čistá reaktivita majú osobitný význam pre konfigurovateľnosť systému. Keď sa za jazdy systému zmení nejaký parameter, tak v prípade, že bola pôvodná hodnota natiahnutá do vnútorného stavu nejakého agenta, vítame možnosť reštartnúť ho. Naopak, pokiaľ je z dôvodu čistej reaktivity nejakým agentom zapísaná do prostredia, stačí ju zmeniť v príslušnom bloku a agent si ju pri najbližšom zobudení prevezme, nakoľko tak robí pri každom zobudení bez ohľadu na to, či sa hodnota mení. Tento spôsob sa zdá dosť nezvyklý, ale jednak systém nezaťažuje – veď agent aj tak čosi z prostredia číta, takže toto nadbytočné čítanie sa vybaví v rámci jedinej sumárnej požiadavky – jednak výrazne zjednodušuje kód agenta (porovnaj Obrázok č. 25 vľavo a vpravo)

```
AgentDefTrigger(“p”,proxy,...);
MsAgentRead(“p”,&p,sizeof(p),NULL);
Agent(“space”);
for (;;) {
 pid = Receive(0,…);
 if (pid == proxy) { // configuration
  AgentRead(“p”,&p,sizeof(p),NULL);
  Agent(“space”);
 }
 else …
}
```

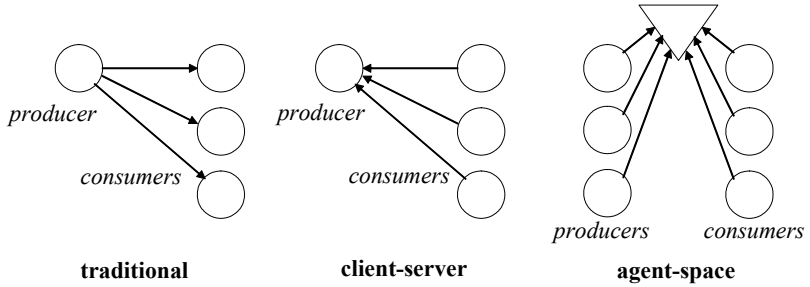
```
for (;;) {
 Receive(…);
 AgentRead(“p”,&p,sizeof(p),NULL);
 …
 Agent(“space”);
 …
}
```

Obrázok č. 25 Vplyv čistej reaktivity na konfigurovateľnosť

Vľavo konfigurovateľné reaktívne riešenie, vpravo čisto reaktívne riešenie

(10) systém možno ľahko naštartovať. Naštartovať systém s pyramidálnou architektúrou klient-server je obtiažne, nakoľko aby sme mohli štartovať ďalšieho klienta, jeho servery už musia byť naštartované. Keby sme chceli systém naštartovať tak, že naraz spustíme všetky procesy, tak musíme do každého klienta na začiatok vložiť špecifický kód, v ktorom kontrolujeme stav jeho serverov, čakáme, sledujeme timeouty a podobne. V Agent-Space proste naštartujeme prostredie, potom všetky agenty a utrasie sa to samo bez toho, že by to bolo zabezpečované špecifickým kódom (nastavovanie defaultov pri čítaní za neho nepovažujeme, lebo slúži aj iným veciam než na naštartovanie).

(11) dátový tok medzi agentami je netradičný (typu many:many). Z hľadiska dátového toku, možno na agent-space pozerať ako na ďalšiu vývojovú fázu - od klasického dátového toku, keď dodávateľ posielal odberateľom, cez klient-server, keď si odberatelia od dodávateľa odoberali, ku kombinovanému spôsobu (Obrázok č. 26). Podobne ako pri klient-server sa dodávateľ nemusí zaoberať tým koľko odberateľom ním vytvorenú informáciu používa. Avšak v agent-space je možné aj to, aby viacerí dodávatelia ten istý blok zapisovali a odberatelia o tom nemali žiadnu vedomosť. Aby sa odberateľ nemusel starať o to koľko má informácia dodávateľov, je úplná novinka. Za túto možnosť platíme tým, že prenos informácie nie je priamy a teda pri ňom dochádza k zdržaniu (to sa dá znížiť na minimum triggrami, ale v interaktívnom systéme je to len málokedy nevyhnutné).



Obrázok č. 26 Dátový tok

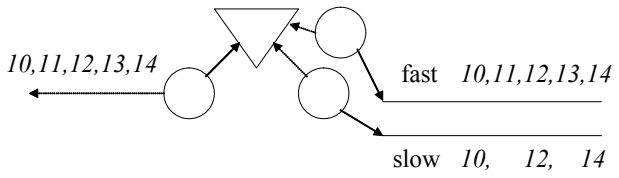
(12) horúca záloha (hot backup) sa veľmi ľahko implementuje. Z možnosti mať viac zapisovateľov rovnakého bloku plynie možnosť nenápadne produkovať obsah tohto bloku viacerými agentami. Dôvodom môže byť rôzna cesta, ktorou je tento obsah dopravovaný,

ale aj rôzny výpočet, ktorým je vyrátaný (napríklad jeden agent rýchlo ale nepresne odhadne, iný pomaly ale presne zráta).

(13) hrozí nekonzistentnosť dát. Problémom agent-space je, že sa určitá informácia môže propagovať príliš pomaly a postupne a nejaký agent potom číta nekonzistentný stav. Dá sa proti tomu samozrejme bojovať tým, že nebudeme používať väčší počet blokov, ktorých hodnota by mala byť vzájomne konzistentná. Dôležité však je, že v niektorých aplikáciách si môžeme dovoliť tento problém nevšímať. Je to všade tam, kde hodnoty blokov zodpovedajú fyzikálnym veličinám, ktoré sa spojito menia. Vtedy je odchýlka od konzistentnej hodnoty príliš malá na to, aby mohla systém ovplyvniť. Napríklad, keď na mobilnom robotovi rátame zo vzdialenosti najbližšej prekážky šikmo doľava a šikmo doprava bezpečný smer pohybu, nebude nám vadiť ak agent rátajúci odchýlku bezpečného smeru od ľavej prekážky bude pracovať s novou meranou hodnotou vzdialenosti od prekážky a so starou spočítanou hodnotou bezpečného smeru. Frekvencia prepočítaní týchto hodnôt je dostatočne vysoká na to, aby sa nová meraná hodnota líšila od starej len minimálne.

(14) hrozí strata dát. Z *(3)* jednoznačne plynie, že sa môže stať, že nejaký agent nestihne prečítať určitú hodnotu, ktorú mu iný agent poslal cez prostredie. Tento problém je len teoretický v systéme reálneho času, kde máme pod kontrolou veľkosti všetkých oneskorení. Aj tak však treba proste konštatovať, že agent-space nie je určená na to, aby niekto cez prostredie spočítaval veci ako peniaze, ale skôr spojito sa meniace hodnoty. Samozrejme, aj peniaze sa dajú v agent-space bezchybne spočítať (napr. si agenty budú potvrdzovať prevzatie, alebo budú používať špecifické meno bloku pre každú dávku), len to nie je elegantné.

(15) implicitné vzorkovanie. Hoci *(14)* sa na prvý pohľad javí ako jednoznačná nevýhoda, z iného pohľadu je to skvelá vec. Zabezpečuje totiž, že ak určitý agent nestíha spracovávať určité dáta (napríklad niečo ťažké s nimi počíta), budú tieto automaticky navzorkované bez toho, že by na to bol určený nejaký explicitný kód. To výrazne napomáha *práci systému v reálnom čase*, čo je inak dosť problém v event-driven systémoch. Keď si predstavíme mobilný robot, je pochopiteľne oveľa lepšie, keď spracúva každé druhé meranie, než keby spracúval merania spred dvoch minút miesto súčasných. Pomocou vzorkovania sa systém dokáže dobre prispôsobiť fyzickej povahe svojich zariadení. Obrázok č. 27 znázorňuje dva rovnaké agenty, ktoré prenášajú prechádzajúce hodnoty skrz rýchle a pomalé komunikačné médium. Kým ten, čo pracuje s rýchlym médiom, prenáša všetky hodnoty, druhý agent pracujúci s pomalým médiom každú druhú. Ich kódy sú však rovnaké a rozdiel v ich správaní je daný len tým, že druhý agent strávi viac času zápisom dát do média a pritom nestihne prevziať jedno meranie.

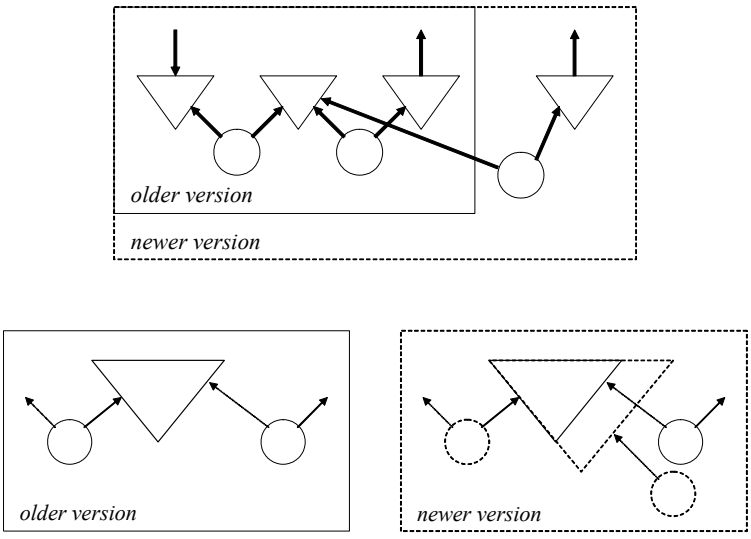


Obrázok č. 27 Implicitné vzorkovanie

(16) v systéme sa vyskytuje jalová práca. Ako sa podľa (14) môže stať, že agent nejaké čítanie vynechá, oveľa častejšie sa môže stať pravý opak – že agent číta a pritom nič nové

v prostredí nemá. A to nielen preto, že bloky v ňom boli zhodou okolností prepočítané na rovnaké hodnoty, ale aj preto, že tam naozaj nikto nič nezapísal. Toto samozrejme platí len o agentoch, ktoré sa budia na timer, tých však preferujeme. Na obhajobu tejto taktiky možno uviesť, že pokiaľ systém pracuje v reálnom čase, musí mať dostatok kapacity aby mohol spracovať každú zmenu a tá sa môže potenciálne vyskytnúť v každom zobudení agenta. Preto táto jalová práca je len čerpaním kapacity, ktorá beztak musí byť k dispozícii[12].

(17) systém je modifikovateľný. Kým všetky predchádzajúce vlastnosti sa prejavovali počas behu systému, teraz sa sústredíme na to, čo sa prejavuje počas jeho vývoja. Agent-space je navrhnutý tak, aby na prvé miesto kládol uľahčenie práce vývojára, hoci aj za cenu vzniku menej efektívneho riešenia (obsahujúceho napríklad jalovú prácu). Keďže v konečnom dôsledku zamýšľame vyvíjať komplexný systém, pôjde nám v prvom rade o podporu jeho modifikovateľnosti. Bežne sa vyskytujúce postupy vývoja [Šešera – Mičovský 1994] vedia značne uľahčiť návrh a vývoj systému, avšak pri modifikáciách to v nich riadne škrípe, lebo treba udržiavať v konzistencii veľké množstvo informácií a to sa málokedy podarí. Duchom týchto systémov je predpoklad, že je možné ich najprv vyvinúť ako návrh na papieri a potom už „len“ implementovať. Náš prístup je celkom opačný: ústredným princípom vývoja je pre nás modifikácia na stelesnenom systéme. Ideálny prípad nastáva, keď je modifikácia realizovateľná pridaním nových blokov a agentov, ktoré manipulujú nielen s nimi, ale aj s blokmi pôvodnej verzie (Obrázok č. 28 hore). Akceptovateľná je tiež situácia, keď rozširujeme určitý blok, pokiaľ zabezpečíme, aby pôvodné agenty z neho dostávali rovnakú časť ako predtým. Keď sú dáta v prostredí púhou postupnosťou bytov, je to ľahké, stačí dáta pridávať na konci buffra (Obrázok č. 28 dolu). Podobne ak sú dáta v tvare XML, dá sa to dosiahnuť rôznou verziou XSD u pôvodných a nových agentov. Najmenej radi modifikujeme systém zmenou kódu už existujúcich agentov, v takom prípade si musíme byť istý dopadom na správanie systému alebo systém podrobiť kompletnému otestovaniu.



Obrázok č. 28 Modifikácia

(18) komunikačné rozhrania sú normalizované. Veľkým nešvárom pyramidálnej architektúry klient-server je používanie rôznych rozhraní na každom serveri. Klient potom musí linkovať niekoľko knižníc obsahujúcich zaobaľovacie funkcie (enwrapping). V agent-space máme jediný server, takže – pochopiteľne – vystačíme s jednou klientskou knižnicou.

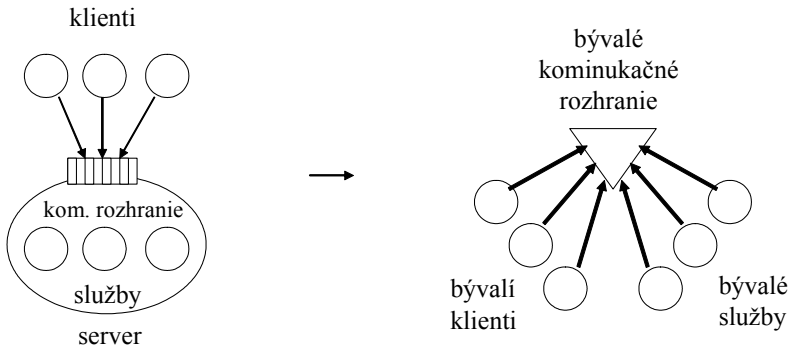
Jej rozhranie síce nebude tak kvetnaté, ale zato bude normalizované. V pyramidálnej klient-server architektúre by klient volal niečo ako

```
GetStereo(&stereo.direction, &stereo.distance)
```

zatiaľ čo v agent-space

```
AgentRead("stereo",&stereo,sizeof(stereo),NULL).
```

Dá sa na to pozerať aj tak, že pôvodné servery sa v agent-space rozpadajú a z ich služieb vznikajú noví klienti a zo serverov ostáva iba komunikačné rozhranie, ktoré sa týmto normalizuje (Obrázok č. 29).



Obrázok č. 29 Normalizácia enwrapping-u

(19) typickou organizačnou vlastnosťou systému je decentralizácia. Ako už naznačujú (7) a (10), v systéme nemá žiadny agent nejaké výhradné postavenie. Samozrejme aj v agent-space je možné nakódovať centrálne riadený systém, kde jeden agent rozdeľuje prácu ostatným, je to však načisto neprirodzené. Naopak, dobre sa kódujú systémy, kde výsledné správanie povstáva z interakcií medzi agentami. Samozrejme, ak odstránime zo systému nejaký agent, ktorý poskytuje do systému dôležitú informáciu, správanie systému nebude bez chýb, systém sa však nezrúti, nezasekne sa, len o čosi zdegraduje. Je to podobné ako keby sme mrzačili nejaký živý systém. A je to odlišné od bežného software, kde sa bežne stáva, že ak zavoláme metódu objektu, ktorý neexistuje, vyvolá sa behová výnimka a je to koniec všetkého. Naša architektúra je voči tomu odolná vďaka dôslednému uplatňovaniu „agentovej" myšlienky: agenty nik nevolá, naopak, iba oni volajú, takže ak nejaký odstránime, potreba volania zanikne spolu s ním. Nikomu nebude chýbať na to aby operoval, môže mu však – samozrejme – chýbať k tomu aby operoval správne.

Na záver tejto kapitoly uvedieme jednoduchý príklad.

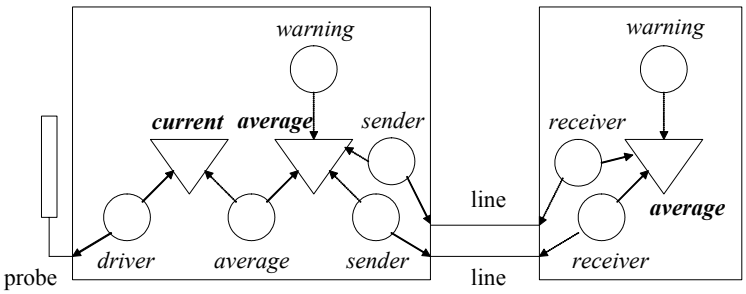
Príklad č. 2

Urobíme analogické riešenie systému, ktorý popisuje Príklad č. 1 a zobrazuje Obrázok č. 9. Pre pohodlie čitateľa zopakujeme zadanie:

Majme sondu ktorá meria určitú veličinu. Táto slúži na varovanie osôb. Meranie má značný rozptyl a preto sa kritická hodnota stanovuje pre priemer za určité obdobie. Varovanie prebieha v mieste merania a na inom vzdialenom mieste. Meranie je dôležité a preto sú tieto dve miesta prepojené dvomi nezávislými spojeniami rovnakého druhu, ktoré sa navzájom zálohujú.

Riešenie v agent-space znázorňuje Obrázok č. 30. Počíta s nakódovaním piatich programov na realizáciu ôsmich agentov:

- *driver* zabezpečuje komunikáciu so sondou: raz za určitý čas urobí meranie a meranú hodnotu vloží do prostredia ako blok ***current***
- *average* pravidelne číta current a vytvára si v sebe okno meraných hodnôt za príslušné obdobie, z ktorého potom počíta (plávajúci) priemer a zapisuje ho do bloku ***average***
- *warning* pravidelne číta hodnotu v ***average*** a ak prekročí limit, tak hlási poplach
- *sender* pravidelne číta hodnotu v ***average*** a posiela ju do linky, aby sa preniesla na vzdialené miesto
- *receiver* na vzdialenom mieste vyberá z linky, čo mu príde a dáva do lokálnej kópie ***average***



Obrázok č. 30 Príklad systému v agent-space

Na rozdiel od riešenia v pyramidálnej klient-server architektúre tu na vzdialenom mieste nepotrebujeme *average2*, čo demonštruje, akou ľahkosťou sa robia zálohované riešenia – viď (12) – proste tu pustíme pár *sender – receiver* dva krát a nestaráme sa. Jednotlivé agenty sú na sebe oveľa nezávislejšie, než procesy v pôvodnom riešení, môžeme ich ľahko naštartovať i reštartovať. Agent *driver* má ďalej výrazne jednoduchšiu štruktúru kódu ako server *driver* v pôvodnom riešení. Ten totiž musel obsluhovať klientov aj počas komunikácie so sondou (inak by sa zablokoval postupne celý systém), zatiaľ čo v agentovom prevedení sa chvíľu zapodieva so sondou, potom chvíľu s prostredím, ale nie s oboma naraz. Keby teraz napríklad došlo k poruche merania a agent *driver* by sa zablokoval pri bavení sa so sondou, konzumenti meranej hodnoty by stále mohli čítať z prostredia poslednú nameranú hodnotu, až by jej vypršala platnosť. Predstavme si, že by sa potom stalo, že agentovi *average* sa po čase vyprázdni okno a on urobí pri výpočte priemeru delenie nulou, teda sa odhalí jeho interná chyba. Tento agent sa zrúti, na čo ho možno automaticky opäť naštartovať. To nám však nepomôže, lebo sa zrúti zas a tak stále dokola. Tým pádom vyprší po čase platnosť aj bloku ***average***. Následne *warning* neprečíta jeho hodnotu, na základe čoho samozrejme nevyhlási

poplach, ale ak má rozumne zvolený default pre toto čítanie, mohol by indikovať poruchu systému. Dobrým deafultom by tu bola ľubovoľná hodnota mimo meracieho intervalu, lebo takáto sa môže aj reálne objaviť v bloku ***average*** pri takej poruche sondy, kedy síce komunikuje, ale dáva zlú hodnotu. *Warning* by takto mohol ľahko upozorniť administrátora, že systém je v poruche. Ten príde a reštartne sondu alebo ju vymení. Čo sa teraz stane? Agent *driver* sa preberie, následne sa v ***current*** objaví hodnota a agent *average* – napriek svojej internej chybe – sa chytí a bude znovu fungovať. Tým pádom sa aj *warning* dostane ku správnej hodnote a odvolá poruchu. Pozoruhodné na tom je, že software sa z dezolátneho stavu dostane sám, hneď ako je odstránená externá príčina, ktorá tento stav vyvolala. Je pravda, že by sa v tomto prípade dal systém urobiť šikovnejšie, bez chýb a podobne – aby sa do takého stavu vôbec nedostal. To ale neznižuje kvalitu architektúry, ktorá je schopná takémuto stavu odolávať. Takisto neslobodno znižovať význam tejto „robustnosti“ s poukázaním na to, že by sme mali venovať energiu radšej metódam ako tvoriť software bez chýb, než metódam ako sa z týchto chýb zotaviť. Žiadny komplexný systém bez chýb nebude a to či už interných (kde je to proste zlyhanie vývojára – ako v prípade agenta *average*) alebo externých (kde proste vývojár nevie, že má s istou možnosťou počítať – ako v prípade agenta *driver*).

Po demonštrovaní operačných schopností, zameriame sa teraz na vývojársku stránku. Ozrejmíme teda, nakoľko je tento systém otvorený ďalším modifikáciám. Môžeme to zviditeľniť na nasledovných príkladoch:

- Keby sme chceli zmeniť typ sondy, stačí nám vymeniť agent *driver* za iného, ktorý implementuje príslušný protokol. Podľa konfiguračného parametra môžeme potom pri štarte systému naštartovať toho správneho.

- Keby sme chceli zálohovať meranie, stačí nám použiť tri sondy a troch agentov *driver*. Tie budú zapisovať do toho istého bloku, čo agent average nebude tušiť, jediné čo sa mu môže stať je, že bude pracovať s hodnotami, ktoré majú väčší rozptyl. Ak sa dva agenty driver zaseknú, alebo nezmerajú (a nemajú čo do prostredia zapísať), nezapíšu nič (keby zapisovali hodnotu „BAD_VALUE“, tak by to nefungovalo – tu aj vidíme, že je lepšie nezapísať nič a radšej čítať s defaultom) a tým pádom bude systém stále pekne fungovať.

- Poľahky môžeme pridať ďalšie miesta, kde prebieha varovanie, keď pridáme na meracie miesto ďalšieho agenta *sender* a prepojíme ho s kópiu riešenia na vzdialenom mieste.

- Keď máme viac miest, môžeme medzi nimi narobiť ľubovoľne veľa prepojení a pokiaľ pri distribúcii budeme používať časové značky (t.j. že receiver zapíše dáta do prostredia s takým začiatkom platnosti, aký mali priradený v pôvodnom prostredí a prostredie odmietne zápis dát, ktorých začiatok platnosti je starší než začiatok platnosti tých, ktoré v danom bloku už má), nemusíme sa báť robiť ani cykly – distribúcia si vždy nájde cestu pokiaľ nejaká bude možná.

- Keby sme chceli pridať meranie nejakej ďalšej veličiny, stačí nám pre jej meracie zariadenie urobiť *driver2*, pustiť druhý krát *average* s iným menom bloku nad ktorým pracuje (*average* nezávisí od aplikačnej domény, takže ho ľahko možno parametrizovať). Samozrejme *warning* - ak má novú veličinu uvažovať – sa musí úplne zmeniť.

- Keby sme chceli za jazdy meniť úroveň poplachového limitu, najlepším riešením by bolo dať ten limit do osobitého bloku. V takom prípade by warning tento limit na začiatku načítal a bol by reštartnutý, keď sa zmení, aby si ho načítal znovu, alebo – ešte lepšie – by si ho čítal z bloku pri každom použití.

- Keby sme požadovali možnosť, aby užívateľ (v prípade, že stratí dôveru k sonde) definoval náhradnú hodnotu (získanú napríklad ručným meraním), stačí túto zapisovať do bloku ***current*** a použiť *write* s prioritou aby boli zápisy agenta *driver* bez efektu.

Vidíme, že i pri takom jednoduchom príklade je dosť o čom hovoriť. Venovali sme sa, pravda, čisto technickými vecami, pri ktorých by málokoho napadlo uvažovať o aplikáciách umelej inteligencie. Ako príklad sme použili monitorovací systém, ale uvidíme v ďalšom, že všetky spomínané vlastnosti sú užitočné i v takých aplikáciách, ako sú mobilné roboty, alebo simulátory živých systémov, čo už znie lepšie. Myslím, že agent-space je silne inšpirovaná živými systémami a obsahuje v sebe niektoré ich stránky. Jej ústrednou myšlienkou je budovať systém z modulov, do ktorých vkladáme jednoduché kódy s usporadúvame ich tak, že odolávajú rušivým vplyvom i za cenu toho, že nekonajú celkom efektívne a ich úspešnosť je limitovaná. Vývojár i príroda sa k takýmto kódom ľahšie dopracujú, než ich efektívnym a účinným analógiám, ktoré navyše majú tendenciu rušivým vplyvom podľahnúť. Avšak prv než rozoberieme podrobnosti aplikácii našej architektúry, budeme sa zaoberať implementačnými obtiažami, bez prekonania ktorých by sme nemohli experimentovať.

[11] Porovnaj definíciu agenta v [Doran 1992]: "an agent is a computational process than can collect information about its environment, can decide what actions to perform perhaps by reference to explicit goals, and can then act upon its environment".

[12] V určitých situáciách – napríklad pre embeded systémy - však musíme zvážiť, že hoci je táto kapacita k dispozícii, zvyšuje spotrebu energie, čo je nepríjemné.

V. Implementácia Agent-Space

Konkrétna implementácia architektúry závisí na platforme. Vo všeobecnosti však vytvárame dve veci:

- nejaké rozhranie, ktorým agent volá služby prostredia
- implementáciu prostredia, konkrétne služby
  - read
  - write
  - delete
  - registráciu triggrov
  - write a delete s prioritami
  - hromadné čítanie blokov
  - zisťovanie časovej platnosti dát, ...

Agent-Space nad SRR (IPC)

V SRR modeli bude agent klientom špeciálneho servera - prostredia (viď kapitolu SRR model posielania správ). Keďže s SRR je zviazaná s platformou QNX4, vyjadrovať sa budeme v jazyku C (RTOSy žiaľ vo vyjadrovacích prostriedkoch vždy trochu zaostávajú za obyčajnými OS). Ako rozhranie na komunikáciu s prostredím bude agentovi slúžiť klientská knižnica obsahujúca funkcie pre jednotlivé služby prostredia (Obrázok č. 31):

```
void AgentRead( char *block_name, void *buffer, int size, struct block_status *status )
void AgentWrite( char *block_name, void *buffer, int size, struct block_status *status )
void AgentDelete( char *block_name, struct block_status *status )
void AgentDefTrigger ( char *block_mask, pid_t proxy, int trigger_type )
void AgentWritePrio( char *block_name, void *buffer, int size, float prio, struct block_status *status )
void AgentDeletePrio( char *block_name, float prio, struct block_status *status )
```

Obrázok č. 31 Typické služby prostredia

Pritom:

- block_name predstavuje meno bloku s ktorým sa manipuluje
- buffer sú dáta voľného typu, ktoré sa do bloku zapisujú, prípadne z neho čítajú (Ak by nejaký agent prečítané dáta asocioval s iným typom, než s akým boli zapísané, dozvedel by sa úplné nezmysly – je však na návrhárovi systému aby k takému niečomu neprišlo.)
- size je dĺžka zapisovaných dát, prípadne očakávaná dĺžka čítaných dát (Ak jeden agent zapíše viac bytov než iný číta, čitateľ dostane len toľko bytov, koľko si zapýtal. Ak je situácia opačná, dostane čitateľ do svojho buffra len toľko bytov, koľko v bloku bolo zapísaných a stav zvyšku ostáva nedefinovaný.)
- trigger_type, označuje druh triggru, budeme sa mu venovať nižšie.
- status slúži na vrátenie doplnkových informácii: v prvom rade či sa služba podarila, alebo kód chyby, kvôli ktorej zlyhala (proces prostredie nebeží, málo pamäte, blok nie je prítomný, ...). Ďalej sú vrátené informácie o platnosti čítaných dát (odkedy a dokedy sú platné). Táto štruktúra navyše obsahuje rezervu pre ďalšie prípadné údaje.

```
struct block_status {
    int errno;
    time_t fromsec;
    int fromnsec;
    time_t tosec;
    int tonsec;
    char reserve[12];
}
```

Aby mohol čítajúci agent stanoviť spomínanú defaultnú hodnotu pre prípad, že by blok nebol v prostredí vytvorený, klientská knižnica zabezpečuje, že čítací buffer zostane v prípade zlyhania čítania nedotknutý. Tak môže do tohto buffra pred čítaním vložiť príslušný default a potom už nemusí ani pátrať či sa čítanie podarilo alebo nie. Knižnica mu to uľahčí tým, že na miesto smerníka na status môže vložiť NULL.

```
int a = 0;
AgentRead("a",&a,sizeof(int),NULL); …
if (a ==1) …
```

Na nastavenie platnosti zapísaných dát slúžia funkcie:

```
AgentSetValidity( int sec, int nsec )
AgentSetValidityFromTo( int fromsec, int fromnsec, int tosec, int tonsec )
```

ktoré platia pre najbližšiu sekvenciu funkcii AgentWrite() (keďže máme len 32 bitovú platformu, čas sa vyjadruje dvomi hodnotami: v sekundách od 1.1.1970 a nanosekundách v rámci aktuálnej sekundy). Pokiaľ sa nezavolajú, majú zapísané dáta neobmedzenú platnosť. Ak sa však definuje koľko sú dáta platné (prvá funkcia) alebo dokedy sú platné (druhá), potom po uplynutí príslušnej doby v prostredí zaniknú (ak ich pravda ešte prv neprepíše nový obsah). Ak sa definuje aj odkedy sú platné (druhá funkcia), budú čitateľné až od tejto doby. Ak sú dáta prepísané novým obsahom, informácia o platnosti pôvodných sa stráca (existujú však aj aplikácie, kde je výhodné implementovať kompletné intervalové správanie).

Možnosť vykonať viac služieb pre jeden agent bez prerušenia iným agentom je zabezpečená tým, že všetky dosiaľ uvedené funkcie len zliepajú komplexnú požiadavku na prostredie, ale neposielajú ju tam. To sa stane, až keď agent zavolá funkciu

```
int Agent (char *space_name)
```

ktorá komplexnú požiadavku pošle do prostredia a prijme od neho odpoveď, ktorú prevedie na výsledky jednotlivých služieb. Napríklad vyššie spomínaný zámok by sa realizoval nasledovne:

```
lock = 0; locked = 1;
AgentRead("lock",&lock,sizeof(lock),NULL);
AgentWrite("lock",&locked,sizeof(locked),NULL);
Agent("space");
if (lock == 0) printf("locked\n");
else printf("wait\n");
```

Pre prípad, že by ich z technických dôvodov bolo viac, je prostredie referencované menom space_name. To je to meno, pod ktorým prostredie prezrádza svoj pid, potrebný na komunikáciu. Funkcia Agent() vracia, či sa toto meno podarilo nájsť alebo nie.

Funkcie prislúchajúce jednotlivým službám spolu s funkciou Agent() teda v architektúre realizujú potrebné zaobaľovanie („enwrapping“) na strane klienta, aby jeho kód vyzeral k svetu[13], pričom iba tá posledne menovaná obsahuje volanie Send()

```
void AgentRead( ... parameters ... ) {
   struct space_request rq;
   ... fill rq with parameters ...
   ... add request rq into msgto ...
}

int Agent( char *space_name ) {
   space_id = name_locate( space_name );
   Send( space_id, &msgto, &msgfrom, sizeof(msgto), sizeof(msgfrom));
   ... fill parameters with returned values from msgfrom ...
}
```

Kód agenta sa potom bude opierať výhradne o funkcie z klientskej funkcie k prostrediu a nebude obsahovať volania žiadnych primitív SRR (Obrázok č. 32). Na úvod si agent okrem inicializácie prislúchajúcej jeho aplikačnej logike vytvorí timer, ktorý mu pravidelne „triggeruje“ proxy alebo si zaregistruje trigger, ktorý mu proxy „triggeruje“ keď sa v prostredí zmení obsah určitých blokov. (Snaha je pritom používať trigger čo najmenej, je to špinavosť, niekedy však nevyhnutná, keby musel mať timer príliš malú periódu aby bol podnet prevzatý včas, filozoficky sa na každý agent dívame ako keby triggre neexistovali.). Potom agent zaspí na volaní Receive(). Keď sa zobudí, zistí si, čo sa zmenilo v prostredí a to prevažne sekvenciou volaní AgentRead(), spočíta, čo má v prostredí zmeniť a vykoná to prostredníctvom sekvencie volaní AgentWrite(), prípadne AgentDelete(). Pritom samozrejme môže zvoliť, že nezapíše nič, alebo zapíše to, čo v prostredí už aj tak je.

```
void main () {
   ... initialization ...
   proxy = proxy_attach();                    // trigger alternatively
   timer = timer_create(-proxy)               // AgentDefTrigger(“a”,proxy, NORMAL);
   timer_set(timer,RELATIVE,0,0,1,0);         // Agent(„space“);
   for (;;) {
      Receive(proxy,0,0);                     // sleep on timer (or trigger)
      AgentRead("a",&a,sizeof(a),NULL); ...   // sense
      Agent("space");
      ... Compute b form a ...                // select
      AgentSetValidity(s,ns);                 // act
      AgentWrite("b",&b,sizeof(b),NULL); ...
      Agent("space");
   }
}
```

Obrázok č. 32 Typický kód agenta

Na záver si ukážeme ako vyzerá implementácia prostredia. Už sme spomínali, že pôjde o špeciálny server a teda sa bude jeho kód veľmi podobať kódu serveru na stane 22. Zamlčíme tentoraz fakt, že ide o server s portami (port je tu potrebný kvôli triggrom – musíme si niekde pamätať proxy cez ktoré poskytuje prostredie notifikáciu pri zmene bloku) a sústredíme sa na fakt, že prostredie spracúva komplexnú požiadavku a poskytuje špecifické služby (Obrázok č. 33).

Z implementačného hľadiska je dôležité poznamenať, že hoci sa prostredie dá implementovať aj primitívnym spôsobom, vyžaduje si použitie efektívnych dátových štruktúr a algoritmov, ak má byť schopné pojať veľký počet blokov, udržiavať ich platnosti a realizovať nad nimi triggre. Na úschovu dát musí slúžiť štruktúra schopná pristúpiť k dátam čase logaritmicky závislom od počtu blokov – v našej implementácii je to AVL strom. Ten má oproti hash tabuľke tú výhodu, že dokáže efektívne vyhľadávať aj podľa masky tvaru <prefix>*. Z toho aj plynie poučenie pre konvenciu názvov blokov – pokiaľ chceme vyjadriť, že viacero blokov patrí k sebe, mali by sme tak urobiť spoločným prefixom. Vyhľadávanie s maskou sa nám zíde v prvom rade pre triggre – keď chceme sledovať či sa zmenil niektorý blok z istej skupiny (pritom samozrejme musíme pamätať, že nielen pre každý trigger musíme nájsť v bloku všetky mená, ktoré vyhovujú jeho maske, ale že aj pre každé nové meno musíme za tým istým účelom prejsť cez doterajšie triggery). Hash taktiež potrebuje pri raste sem-tam prepracovať celú tabuľku, čo v systéme reálneho času predstavuje výkonnostný „peak“ a to neradi vidíme[14].

```
void main () {
   struct space_msg msgin, msgout;              ... initialization ...
   for (;;) {
      pid = Receive(0,&msgin,sizeof(msgin));    // receive a complex request msgin
      do {                                      // go through individual requests msg in msgin
         switch (msg.action) {
            case ACTION_READ:                   // compute msgout from msgin
               ...                              // and adjust data stored in space ...
               break;
            case ACTION_WRITE:
               ...
               break;
            case ACTION_DELETE:
               ...
               break;
            case ACTION_DEF_TRIGGER:
               ...
               break;
            case ACTION_WRITE_PRIO:
               ...
               break;
            case ACTION_DELETE_PRIO:
               ...
               break;
         }
      } while (...);
      Reply(pid,&msgout,sizeof(msgout));        // respond with data
   }
}
```

Obrázok č. 33 Typický kód prostredia

Ďalším problémom je dostatočne efektívne kontrolovanie časovej platnosti blokov. Keby sme nepotrebovali triggerovať vypršanie platnosti bloku, stačilo by pri každom čítaní, skontrolovať, či už dáta nie sú „po záručnej dobe“. Spravidla je však potrebné triggerovať každú zmenu bloku, teda aj moment, keď je z prostredia odstránený. Tak sa to deje pri najbežnejšom type triggrov – type NORMAL. Prax ukazuje, že sa to dá zvládnuť aj prehľadaním celého AVL

stromu napríklad prefixom, ale musíme stanoviť rozumnú základnú jednotku merania času. Na to, aby sme mohli povoliť jednotku na úrovni mikrosekúnd už potrebujeme efektívnu dátovú štruktúru, ktorá bloky triedi podľa platnosti, ale zároveň ich umožňuje odstraňovať podľa mena bloku, napríklad ďalší AVL strom spriahnutý s pôvodným.

Je rozumné zaviesť aj ďalšie typy triggrov a to v súvislosti s doplnkovými službami, ktoré umožňujú hromadnú manipuláciu s blokmi: hromadné čítanie, prípadne hromadné vymazanie. Tieto operácie sú vhodné napríklad na vypísanie obsahu všetkých blokov v prostredí, zrkadlenie blokov z jedného prostredia do iného a podobne, čiže skôr na veci nezávislé od aplikačnej domény. Kým hromadné mazanie sa realizuje ľahko rozšírením významu argumentu AgentDelete() na masku, napríklad AgentDelete("a*"), hromadné čítanie vyžaduje náročnejšiu implementáciu. Na rozdiel od štandardných služieb tu posielame prostrediu jednoduchú informáciu a dostávame komplexnú odpoveď. Preto potrebujeme funkcie na rozobratie tejto odpovede. Navyše z technických dôvodov sa môže stať, že informácia od prostredia je tak veľká, že nejde preniesť na jeden krát. Tým pádom nadobúda služba povahu transakcie.

```
char name[AGENT_MAX_NAME];
int size;
data *void;
block_status st;
AgentReadAll("*");                              // complete query
Agent("space");                                 // communicate with space
while (data =  AgentGet(name,&size,&st)) {      // get individual data from reply
   if (size) {
      printf("block %s present\n",name);
      if (!strcmp(name,"thing")) {              // example of data employment
         THING *thing = (THING *) data;
          printf("attribute: %s\n",thing->attr); …
      }
      …
   }
}
```

Hromadné čítanie je výhodné spojiť s triggrom a vtedy potrebujeme, aby trigger nielen poskytol notifikáciu, že niektorý blok vyhovujúci maske bol zmenený, ale aby si aj pamätal, ktoré bloky to presne boli.

```
...
AgentDefTrigger("*",proxy,MULTIPLY);            // define multiply trigger
Agent("space");
...
Receive(proxy,NULL,0);                          // wait for trigger
AgentReadTriggered("*");                        // complete query
Agent("space");                                 // communicate with space
while (data =  AgentGet(name,&size,&st)) {      // get individual data from reply
   if (size) {
      printf("block %s present\n",name);      …
   }
   else {
      printf("block %s removed\n",name);    …
   }
}
```

S týmito službami je už prostredie plne funkčné a vhodné pre najnáročnejšie aplikácie. V tejto podobe (s malými odchýlkami v názvoch funkcií) sme Agent-Space implementovali na platforme QNX4 v jazyku C, čím vznikol komerčný balík MsAgent. Tento bol v praxi použitý na niekoľko aplikácii reálneho času: letištný meteorologický systém, systém presného merania teplôt jadrového reaktora, emisný monitorovací systém, seizmologické meranie, univerzálny zberný systém z meracích miest. S výnimkou posledného o ktorom ešte bude reč, pri nich nebolo úlohou dosahovať správanie, ktoré by bolo primerané nazvať inteligentným. Nič menej, interaktívny charakter týchto systémov nás priviedol k množstvu poznatkov o detailoch tejto architektúry, ktoré by sa inak len ťažko podarilo odhaliť. S relatívnym úspechom sme potom použili rovnaké podporné prostriedky napríklad na simulovanie správania sa hmyzu (konkrétne šlo o zakladanie potomstva kutavky rodu cerceris [Lúčny 2001b]).

Príklad č. 3

Uvedieme implemetnáciu základného riešenia, ktoré navrhuje Príklad č. 2 .

```
/* Agent Driver */
void main (int argc, char *argv[]) {
   pid_t proxy = proxy_attach();
   timer_t timer = timer_create(-proxy);
   timer_set(timer,RELATIVE,0,0,1,0);
   line_open(argv[1]);
   for (;;) {
      float v;
      Receive(proxy,0,0);
      line_write(request());
      v = response(line_read());
      AgentSetValidity(1,500000000);
      AgentWrite("current",&v,sizeof(v),NULL);
      Agent("DATA");
   }
}

/* Agent Average */
void main () {
   int min = 0; pid_t proxy = proxy_attach();
   AgentDefTrigger(“current”,proxy, NORMAL);
   Agent(“DATA”); frame_init();
   for (;;) {
      float v, a; block_status t;
      Receive(proxy,0,0);
      AgentRead("current",&v,sizeof(v),&t);
      Agent("DATA");
      if (min != t.fromsec/60) {
         min = t.fromsec/60; a =  frame_average();
         AgentSetValidity(65,0);
         AgentWrite("average",&a,sizeof(a),NULL);
         Agent("DATA");
      }
      frame_insert(v,t.fromsec);
   }
}
```

```
/* Agent Warning */
void main () {
   pid_t proxy = proxy_attach();
   timer_t timer = timer_create(-proxy);
   timer_set(timer,RELATIVE,0,0,60,0);
   for (;;) {
      float a;
      Receive(proxy,0,0);
      AgentRead("average",&a,sizeof(a),NULL);
      Agent("DATA");
      if (a > LIMIT) warn();
   }
}

/* Agent Sender */
void main (int argc, char *argv[]) {
   AgentDefTrigger(“average*”,proxy,MULTIPLY);
   Agent(“DATA”);
   line_open(argv[1]);
   for (;;) {
      void *data; char name[MAX]; int size; block_status st;
      Receive(proxy,NULL,0);
      AgentReadTriggered(proxy);
      Agent(“space”);
      while (data =  AgentGet(name,&size,&st))
         if (size) line_write(protocol(name,size,st,data));
   }
}

/* Agent Receiver */
void main (int argc, char *argv[]) {
   line_open(argv[1]);
   for (;;) {
      void *data; char name[MAX]; int size; block_status st;
      data = protocol(read_line(),name,&size,&st);
      AgentSetValidity(st.tosec-st.fromsec,0);
      AgentWrite(name,data,size,NULL);
      Agent("DATA");
   }
}
```

Agent-Space nad OOP

Z možných modernejších objektových platforiem sme si na porovnanie vybrali Javu. Oproti SRR tu máme niekoľko odlišností:

- nemáme multiprocesový OS, iba multithredovú virtuálnu mašinu. Prostredie sa tým mení na samostatný pasívny objekt, agenty na objekty bežiace vo vlastnom vlákne
- máme k dispozícii prepracovanú objektovú technológiu. Rozdielom oproti tradičnému ponímaniu OOP je absencia viacnásobného dedenia a jeho náhrada rozhraniami (interface). (Tieto rozhrania popisujú ako sa budú volať metódy tried, ktoré vývojár ešte len vytvorí, zatiaľ čo samotné triedy definujú to, čo už má k dispozícii.) Ďalším rozdielom je absencia deštruktorov a prítomnosť „garbidge collector“-u.

Hlavnou inšpiráciou pre vybudovanie space v OOP je zriedkavo používaný návrhový vzor „namespace“ (použitý je napríklad v implementácií multiprocesového frameworku Echidna[15]), ktorý sa používa na referenciu objektov, ktoré boli vytvorené dynamickým inštancovaným určitej triedy na základe udania jej mena:

```
Class newClass = Class.forName(className);
Object newObject = newClass.newInstance();
```

Pokiaľ je počet takýchto objektov pevný a malý, nie je problém ich referencovať pomocou tzv. „holderov“. Pre každý takýto objekt môžeme mať zaobaľovací objekt implementujúci určité rozhranie, do ktorého sa dá vložiť skutočná dynamicky vytvorená inštancia a ktorý sprostredkuje volanie jej metód (naľavo je jedno vlákno, napravo iné):

```
Holder holder = new Holder();
...                                          holder.insert((newInterface) newObject);
holder.method();
```

Ak ich je však veľa a navyše ich plánujeme modifikáciami pridávať, je otravné pre každý takýto objekt urobiť príslušný „holder“ a preto sa použije „namespace“, ktorý je vlastne kolekciou univerzálnych „holderov“ (naľavo je jedno vlákno, napravo iné):

```
NameSpace namespace = new NameSpace();
…                                            namespace.write(“new”,newObject);
((newInterface) namespace.read(“new”)).method();
```

Pomenúvanie objektov menom teda nie je pre OOP novou, hoci dosť zriedkavou vlastnosťou. Ďalší posun od „namespace“ k nášmu „space“ je však už radikálnym útokom na štandardný model spracovania udalostí. Vo všeobecnosti v OOP možno za štandardný považovať model „signal-slot“. Konkrétne v Jave sa potom zužuje na model definovaný v rámci knižnice swing, ktorý je založený na tzv. „listeneroch“. Spočíva v tom, že u každého objektu, ktorý môže generovať udalosť, je možné sa zaregistrovať ako jej príjemca – „listener“. Môže tak však učiniť len taký objekt, ktorý implementuje rozhranie, ktoré objekt generujúci udalosť volá pri jej vzniku. Niektoré udalosti generujúce objekty používajú na toto volanie rovnaké rozhrania, vo všeobecnosti však platí, že počet týchto rozhraní sa rádovo blíži počtu typov objektov, ktoré ich volajú.

```
interface ActionListenerX {
  public void actionX();
}

class GeneratingX {
  …
  public void addListener (ActionListenerX listener) {
    listeners.add(listener);
  }
  … listeners.get(i).actionX(); … // generate event X
}

class ListenerX implements ActionListenerX {
  ListenerX (GeneratingX g) {
    g.addListener(this);
  }
```

```
  public void actionX() {
    … // process event X
  }
  …
}
```

Jediný objekt, ktorý preberá udalosti od mnohých ďalších typov objektov, potom musí implementovať mnoho „listenerovských“ rozhraní.

```
class ListenerXYZ implements ActionListenerX, ActionListenerY, ActionListenerZ  {
  public void actionX( …argX1, arg X2… ) {
    … /* process event X */
  }
  public void actionY( …argY1, argY2… ) {
    … /* process event Y */
  }
  public void actionZ( …argZ1, argZ2… ) {
    … /* process event Z */
  }
  …
}
```

Podobne ako pri klient-server, klienti tučneli zo zaobaľovacích knižníc, objekty prijímajúce udalosti modelom swing tučnejú z implementovaných rozhraní. Zoštíhliť ich možno v princípe akoukoľvek normou, ktorá pre „listenerov“ predpokladá zavedenie jediného rozhrania. To je však je možné len vtedy, ak sa relevantné dáta prenášajú opačným volaním než je tomu v štandardnom modeli. Pri tomto volaní totiž prenášame konkrétne argumenty, takže ak voláme z objektu, ktorý udalosť generuje objekt, ktorý ju prijíma, musí mať prijímateľ v sebe na to určenú špecifickú metódu – a teda je tučný. Štíhly môže byť iba keď toto volanie vykonáme naopak, teda relevantné dáta vráti metóda objektu, ktorý udalosť generuje, pričom je táto volaná z objektu, ktorý ju prijíma. Tento prijímateľ by však musel byť jasnovidný, aby vedel, kedy ju má zavolať. A tu nám opäť vyskakujú dve možnosti: aby ju volal pravidelne (timer), alebo aby sa dozvedel, že ju treba zavolať (trigger). Na to samozrejme musí implementovať určité rozhranie, ale už jediné, univerzálne. Toto sa bude zameriavať iba na zistenie toho, že sa niečo stalo, zatiaľ čo to, čo sa stalo, sa vybaví mimo tohto rozhrania.

Takto síce prijímateľa zoštíhlime, avšak málo nám to bude platné, lebo aj tak musí každý objekt, ktorú generuje udalosti, poskytovať registráciu, aby sme sa dozvedeli, že udalosť nastala. Odstrániť toto je možné len zriadením tretej entity (to bude ten náš space), ktorá proste doručenie udalosti sprostredkuje. Ani jeden z objektov medzi ktorými dátový tok zriaďujeme teda nesmie byť volaný, môže iba on sám volať. V opačnom prípade je potenciálne tučný. Dostávame sa pomaly k tomu, že agent-space je nielen jednou možnou, ale vlastne jedinou štíhlou normalizáciou spracovania udalostí v OOP.

Ako to teda bude vyzerať? Najprv definujeme rozhranie Triggerable na realizáciu prenosu informácie o tom, že sa niečo stalo:

```
interface Triggerable {
  void trigger();
}
```

Toto jednoduché rozhranie bude jediné, ktoré bude náš model spracovania udalostí potrebovať a ktoré musia všetky udalosti prijímajúce objekty implementovať. Ďalej potrebujeme ten space.

Ten musí byť v princípe jediný pre všetky objekty. Najjednoduchšie to bude zariadiť použitím návrhového vzoru „singleton“.

```
public class Space {

  private static Space singleton = new Space();
  // variables for representation of bloks and triggers

  public static Space getInstance() {
    return singleton;
  }

  protected Space() {
    // initialization of the variables
  }

  synchronized public void write (String name, Object value, long validFrom, long validTo, float prio) {
    // implementation
  }

  public void write (String name, Object value, long validFor) {
    long now = System.currentTimeMillis();
    write(name,value,now,now+validFor,0);
  }

  public void write (String name, Object value) {
    long now = System.currentTimeMillis();
    write(name,value,now,Block.FOREVER,0);
  }

  synchronized public Object read (String name) {
    Block block;
    // implementation
    return block.value;
  }

  public Object read (String name, Object def) {
    Object obj = read(name);
    if (obj == null) return def;
    return obj;
  }

  synchronized public void attachTrigger (String mask, Triggerable agent, int type) {
    // implementation
  }

  /* additional methods */
  synchronized public void detachTrigger (Triggerable agent) { /* ... */ };
  synchronized public BlockStatus readFirst (Triggerable agent) { /* ... */ };
  synchronized public BlockStatus readNext (Triggerable agent) { /* ... */ };
  public void write (String name, Object value, float prio) { /* ... */ };
  public void delete (String name) { /* ... */ };
  ...
}
```

Takto dosiahneme, že všetky agenty budú zdieľať jediný objekt – space, nakoľko nebudú môcť volať konštruktor:

```
Space space = new Space();
```

ale len statickú metódu:

```
Space space = Space.getInstance();
```

Keby sme predsa len potrebovali viac space-ov, použili by sme návrhový vzor „singleton register“, volali by sme teda:

```
Space space = Space.getInstance(„name“);
```

To by malo význam pre púšťanie viacerých nezávislých projektov pod jednou JVM, keď pri prídavných funkciách potrebujeme napríklad správne interpretovať trigger na „*“.

Čo sa týka samotnej implementácie Space, nebudeme tu zachádzať do problémov efektívnej algoritmizácie, nakoľko pri nej vystačíme s bežnými poznatkami z algortimov a dátových štrultúr. Java nám navyše poskytuje dosť silnú podporu v podobe balíka java.util ktorý obsahuje kľúčové štruktúry TreeMap<K,V> a TreeSet<V>. Musíme si dať akurát pozor na synchronizáciu.

Keďže už máme Space, môžeme pristúpiť k implementácii agentov. Pre tie je najlepšie zriadiť abstraktnú triedu, ktorú budú dediť všetky konkrétne agenty. Základnou úlohu tejto triedy bude vytvoriť pre agent jeho vlastné vlákno a zároveň zabezpečiť možnosť zablokovania a odblokovania jeho vykonávania – teda fázu *Sleep*. Na to môžeme v Jave použiť mechanizmus synchronizácie opierajúci sa o monitor, ktorý má potenciálne každý objekt a ktorý umožňuje synchronizovať dve vlákna pomocou metód wait() a notify(). To druhé vlákno, s ktorým sa bude vlákno agenta synchronizovať, môže byť jednak vláknom časovača (ktorého podporu zimplementujeme tiež v tejto abstraktnej triede) – keď ho budí timer, jednak vláknom iného agenta (nakoľko space nemá vlastné vlákno a volania jeho kódu sú rozložené do vlákien jeho klientov), keď ho budí trigger. Pritom v oboch prípadoch bude budenie agenta zabezpečené implementáciou rozhrania Triggerable.

```
public abstract class Agent extends Thread implements Triggerable {
  Object monitor;
  Timer timer;
  Space space;

  public Agent () {
    monitor = new Object();
    space = Space.getInstance();
    timer = null;
  }

  public void setTimer(int period) {   // to be called from init()
    if (timer == null) timer = new Timer(true);
    timer.scheduleAtFixedRate( new TimerTask() {
      public void run() {
        trigger();
      }
    },period,period);
  }
```

```
public void setTrigger(String name, int type) {   // to be called from init()
  space.attachTrigger(name,this,type);
}

abstract void init(); // to be overriden

abstract void sense_select_act();  // to be overriden

private int receive() {
  int ret = 0;
  synchronized(monitor) {
    try {
      monitor.wait();
    }
    catch (InterruptedException e) {
      ret = -1;
    }
  }
  return (ret);
}

public void trigger() {
  synchronized (monitor) {
    monitor.notifyAll();
  }
}

public void main () {
  init();
  for (;;) {
    receive();
    sense_select_act();
  }
}

public void run () {
  main();
}

public Object read (String name, Object def) {
   return space.read(name,def);
}

public void write (String name, Object value, long validFor) {
   space.write(name,value,validFor);
}

// further analogous methods for methods of the space

/* additional methods */
public BlockStatus readFirst () {
  space.readFirst(this);
}
```

```
  synchronized public BlockStatus readNext () {
    space.readNext(this);
  }

}
```

Toto už je celkom rozumná implementácia agent-space, s ktorá sa dá plne použiť na výskumné účely. Implementovali sme s ňou napríklad riadiaci systém mobilného robota s podvozkom na báze stavebnice Boe-bot vybaveného kamerou. V praxi sa však ukazuje, že požiadavka definovať konkrétny agent prekrytím init() a sense_select_act() je trochu „akademická“. V praxi totiž potrebujeme agenty spojiť s činnosťami ako je obsluha vstupov a výstupov, socketov, existujúcich grafických rozhraní a pod. Potrebujeme preto preberať udalosti na rôznych miestach. Preto sa hodí nedefinovať init() a sense_select_act() ako abstraktné (aby sme ich nemuseli implementovať) a priamo prekryť metódu main() (ktorú sme si predchádzajúcom kóde vytiahli z implementácie vlákna v run() už s týmto zámerom) a volať z nej receive() všade, kde potrebujeme, alebo volať úplne inú metódu blokovania (napríklad čítanie vstupu).

Z technických komentárov stojí za zmienku, že Java poskytuje väčší komfort nakoľko je podstatne modernejšia než C. Napríklad v meraní času možeme využiť 64 bitový integer a teda môžeme merať čas jediným údajom a to napr. v milisekundách. Ďalej nepotrebujeme analógiu funkcie Agent() z implementácie nad SRR, nakoľko viac príkazov môžeme bez prerušenia vykonať synchronizovaním postupnosti týchto príkazov na monitor samotného space-u. Čo sa nám urobí výrazne ťažšie, je sledovanie časovej platnosti blokov. JVM totiž nie je systém reálneho času. Preto je vhodné obmedziť sa už spomínanú stratégiu kontroly vypršania platnosti pri čítaní bloku, pokiaľ je to z hľadiska aplikácie možné. Taktiež čo sa týka povahy objektov, ktoré možno do space-u uložiť, treba poznamenať, že v prvom rade nemožno zapísať premennú základného typu, ale len objekt (teda miesto int musíme zapísať Integer) a z hľadiska hromadnej manipulovateľnosti s objektami by bolo vhodné obmedziť sa objekty implementujúce rozhranie Serializable (v Jave vnútorne mashalovateľné objekty). Problematickým je taktiež klonovanie objektov uložených v space. Pre inštancie triedy Objekt je totiž v Jave klonovanie úmyselne zakázané a ani nejaké všeobecnejšie rozhranie ako Serializable ho nepodporuje. Preto je jednoduchšia taktika vkladaný objekt neklonovať. Vývojár si potom ale musí dať pozor aby objekt prečítaný zo space-u nemodifikoval. V opačnom prípade by sa modifikácia prejavila aj u ostatných agentov a to nekontrolovateľným spôsobom. Iná možnosť je zabezpečiť kolovanie ukladaním objektov do Space v marshalovanej podobe, či už v natívnej pomocou rozhrania Serializable, v podobe danej nejakým štandardom alebo čisto v proprietárnej podobe. Zo štandardov je zaujímavou možnosťou použitie mapovania objektov do XML (XML-binding, JAXB).

Porovnanie nášho prístupu s COOP (concurrent object-oriented programming), menovite na základe dlhoročnej práce Douga Leaho [Lea 1999] ukazuje, že napriek rôznorodosti foriem COOP, je náš prístup dosť odlišný. Balík java.util.concurrent, ktorý je technickým prostriedkom COOP v Jave, však aj napriek tomu poskytuje dosť prostriedkov, pomocou ktorých by sa dala Agent-Space implementovať priamočiarejšie, avšak metodologická hodnota takého prístupu by bola nižšia.

Príklad č. 4

Uvedieme analógiu riešenia, ktoré podáva Príklad č. 3 , v Jave.
(Opäť uvádzame len podstatnú časť kódu.)

```
public class AgentDriver extends Agent {

  private String port;
  private Line line;

  public AgentDriver (String port) {
    this.port = port;
    start();
  }

  void init () {
    line = new Line(port);
    setTimer(1000);
  }

  void sense_select_act() {
    line.write(request());
    float v = response(line.read());
    write("current",new Float(v),500);
  }

}

public class AgentAverage extends Agent {

  private min = 0;
  private Frame frame;

  void init () {
    min = 0; frame = new Frame(60);
    setTrigger("current",Trigger.NORMAL);
    start();
  }

  void sense_select_act() {
    BlockStatus t = new BlockStatus();
    float v = ((Float) read("current", new Float(-1), t)).floatValue() ;
    if (min != t.validFrom/60000) {
      min = t.validFrom/60000;
      float a =  frame.getAverage();
      write("average",new Float(a),65000);
    }
    frame.add(v,t.validFrom/1000);
  }

}

public class AgentWarning extends Agent {

  public AgentWarning () { start(); }

  void init () {
    setTimer(60000);
  }
```

```
void sense_select_act() {
   float a = ((Float) read("average",new Float(-1))).floatValue();
   if (a > LIMIT) warn();
  }

}

public class AgentSender extends Agent {

  private String port;
  private Line line;

  public AgentSender (String port) {
    this.port = port;
    start();
  }

  void init () {
    line = new Line(port);
    setTrigger("average*",Trigger.MULTIPLY);
  }

  void sense_select_act() {
    BlockStatus st = readFirst();
    while (st != null) {
      line.write(protocol(st.getName(),marshall(st.getValue()),st.getValidity()));
    }
  }

}

public class AgentReceiver extends Agent {

  private String port;
  private Line line;

  public AgentSender (String port) {
    this.port = port;
    start();
  }

  void main () {
    line = new Line(port);
    for (;;) {
      ProtocolStatus st = protocol(line.read()); // read is blocking
      if (st != null)
        write(st.getName(),unmarshall(st.getValue()),st.getValidity());
    }
  }

}
```

```
public class Place1 {
  public static void main (String args) {
    new AgentDriver(args[3]); new AgentAverage(); new AgentWarning();
    new AgentSender(args[1]); new AgentSender(args[2]);
  }
}

public class Place2 {
  public static void main (String args) {
    new AgentWarning();
    new AgentReceiver(args[1]); new AgentReceiver(args[2]);
  }
}
```

Agent-Space ako middleware

Doteraz sme o architektúre Agent-Space uvažovali ako o nástroji na vytváranie jednouzlových riešení (teda sme pôsobili v rámci „concurrent systems“). Implicitnou vlastnosťou každej agentovej technológie je však schopnosť podporovať distribuované riešenia, či už v rámci LAN alebo WAN typu Internet (ide teda o „distributed systems“). Skutočne aj k implementácii Agent-Space sa dá pristúpiť ako ku implementácii špecifickej relačnej a prezentačnej vrstvy modelu ISO OSI. Pritom je možné implementáciu postaviť priamo na komunikačnej vrstve (napríklad na TCP/IP) alebo sa včleniť medzi aplikačnú vrstvu a existujúci middleware (napríklad CORBA, RMI). Je taktiež možné využiť niektorý z aplikačných protokolov internetu (napríklad http) a služby ktoré vykonáva space realizovať ako webové služby (WS) zaintegrované do web servera (toho času populárna internetová technológia). Možností je veľa, lebo sieťové programovanie je nad komunikačnou úrovňou (ktorú väčšinou obstaráva TCP/IP) veľmi rôznorodé.

Na rozdiel od riešenia pre jednu JVM, kde bolo neobyčajným používanie timerov na prenos udalosti, v distribuovanom prípade je to presne naopak: neobyčajnými sú triggre. Väčšina platforiem pre distribuované programovanie má s prenosom udalosti od servera ku klientovi vážne problémy a prípadne ho vôbec neumožňujú (typickým príkladom sú databázové aplikácie). To v podstate znemožňuje ísť v odozve dostatočne nízko, čo výrazne limituje doménu možných aplikácii. Radi by sme preto mali triggre k dispozícii aj tu.

Najelegantnejšie riešenie pre povýšenie implementácie Agent-Space z predošlej kapitoly na middleware ponúka RMI, kde môžeme v podstate priamo zabaliť riešenie pre jednu JVM a vytvoriť riešenie pre viac JVM (to už je jedno či na jednom uzle alebo v rámci siete). Musíme sa pri ňom len vysporiadať s obmedzeniami RMI. Aj tu musíme implementovať rozhranie pre spätný prenos informácie:

```
import java.rmi.*;
import java.io.*;

public interface RemoteTriggerable extends Remote {
  void trigger() throws RemoteException;
}
```

Ďalej vytvoríme rozhranie pre space, ktoré budú používať agenty na vzdialených JVM aby sa dostali na skutočnú implementáciu space na lokálnej JVM (kvôli jednoduchosti výkladu pritom ignorujeme veci ako časová platnosť a podobne):

```
import java.rmi.*;
import java.io.*;

public interface RemoteSpace extends Remote {
  void write (Object key, Object value) throws RemoteException;
  Object read (Object key) throws RemoteException;
  void attachTrigger (RemoteTriggerable listener) throws RemoteException;
  ...
}
```

Ďalej vytvoríme jeho implementáciu a pustíme ju ako RMI objekt a priradíme jej meno – napríklad – SPACE, pod ktorým je viditeľná pre ostatné JVM. Tie na ňu získajú kontakt tak, že s odvolávaním sa na jej meno kontaktujú cez TCP/IP port 1099 démona rmiregistry, ktorý im ho sprostredkuje. Tohto démona štartujeme pri spúšťaní space, ak ešte nebeží. Pre jednoduchosť sa tu obmedzujeme ne riešenie v rámci jedného uzla, pričom riešenie pre viac uzlov je rovnaké až na to, že musíme použiť meno napr. //192.168.145.1:1099/SPACE (na POSIX) alebo //NOD1:1099/SPACE (na Win32, pričom NOD1 je tu tzv. „full computer name“ v „Network ID“). Všimnime si, že na realizáciu triggrov musíme používať adaptér aby sme zo vzdialeného urobili lokálne.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;
import java.util.*;
import java.io.*;

public class RemoteSpaceImplementation extends UnicastRemoteObject implements RemoteSpace {

 Space space;

 public RemoteSpaceImplementation() throws RemoteException {   // super();
  space = Space.getInstance();
 }

 public void write (Object key, Object value) throws RemoteException {
  space.write(key,value);
 }

 public Object read (Object key) throws RemoteException {
  return space.read(key);
  return obj;
 }

 public void attachTrigger(String name, final RemoteTriggerable remoteAgent) throws
 RemoteException {
  Triggerable agentAdaptor = new Triggerable() {
   public void trigger() {
    remoteAgent.trigger();
   }
  }
  space.attachTrigger(name,agentAdaptor);
 }
```

```
  public static void main(String[] args) {
    System.setProperty("java.security.policy","policy");
    try {
      LocateRegistry.createRegistry(1099);
    }
    catch (RemoteException e) {
      System.out.println("probably registry already running");
    }
    System.setSecurityManager(new RMISecurityManager());
    String bindName = "SPACE";
    try {
      RemoteSpace space = new RemoteSpace();
      Naming.rebind(bindName, space);
    }
    catch(Exception e) {
      System.out.println("space not launched");
    }
  }

}
```

Pri tejto implementácii narážame aj na problém všetkých distribuovaných systémov – problém bezpečnosti. Vymedziť klientov, ktorí majú právo manipulovať so space sa dá v RMI pomocou tzv. „policy“ súboru. Pokiaľ nás bezpečnosť nestraší, definujeme obsah súboru policy ako:

```
grant {
// Allow everything for now
permission java.security.AllPermission;
};
```

Keď teraz odpálime RemoteSpaceImplementation, naštartuje sa nám rmiregistry a JVM realizujúca space. Teraz si môžeme vyvinúť agenty, ktoré budú so space manipulovať a to z iných JVM. Paradoxne to nebudú RMI klienti, ale rovnako ako space RMI servery – to kvôli triggrom, aby ich vedel space vo vhodnej chvíli zobudiť. Blokovať sa pritom budú lokálnymi synchronizačnými mechanizmami.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*; import java.util.*; import java.io.*;

public class RemoteAgent extends UnicastRemoteObject implements RemoteTriggerable, Runnable {

  RemoteSpace space;
  Object monitor = new Object();

  public RemoteAgentListener() throws RemoteException {
    try {
      space = (Space) Naming.lookup("SPACE");
    } catch(Exception e) {
      System.out.println("probably space not present");
    }
  }
```

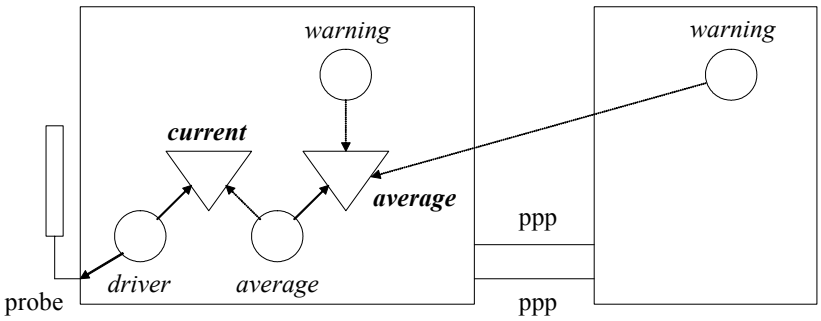
```
 public setTrigger(String name) {
  try {
    space.attachtrigger(name,this);
  } catch(Exception e) {
    System.out.println("trigger not attached");
  }
 }

 public setTimer(int period)      { /* the same as in the Agent definition */ }
 public void main()               { /* the same as in the Agent definition */ }
 public void init()               { /* the same as in the Agent definition */ }
 public void sense_select_act()   { /* the same as in the Agent definition */ }

 public void run() {
  main();
 }

 public void trigger() throws RemoteException {
  synchronized (monitor) {
    monitor.notifyAll();
  }
 }

 public void receive() {
  synchronized(monitor) {
    try {
      monitor.wait();
    } catch (InterruptedException e) {
    }
  }
 }
}
```

Konkrétne agenty potom odvodíme od tejto triedy. Napríklad:

```
public class RemoteAgentExample extends RemoteAgent {

 public RemoteAgentExample () {
     new Thread(this).start();
 }

 public void init() { /* ... */  }
 public void sense_select_act() { /* ... */  }

 public static void main(String[] args) {
  System.setProperty("java.security.policy","policy");
  System.setSecurityManager(new RMISecurityManager());
  try {
    new RemoteAgentExample();
   } catch (Exception e) {
   }
 }

}
```

Príklad č. 5

S takouto implementáciu by sme mohli prerobiť Príklad č. 4 tak, že na prenos blokov z jedného miesta na druhé nebudeme používať dvojicu agentov sender – receiver, ale *warning* na vzdialenom mieste bude priamo pristupovať na space v mieste merania (Obrázok č. 34). Dualita siete by sa potom zariadila na úrovni TCP/IP (z liniek by sme urobili dve ppp spojenia a zabezpečili príslušné smerovanie).



Obrázok č. 34 Príklad distribuovaného riešenia na báze middleware

Daňou za použitie RMI však je, že takýto space je prístupný iba z platformy Java. Pritom v distribuovanom programovaní sa pozitívne hodnotí multiplatformnosť. Tú môže poskytnúť napríklad RMI-IIOP, ktoré natívny marshalling Javy nahradzuje štandardom, ktorý predpisuje CORBA. Každé takéto riešenie je založené na mapovaní natívneho objektu do určitého štandardného formátu. To mapovanie sa dá zároveň použiť ako klonovací mechanizmus, teda miesto natívnych objektov môžeme v space držať mapované klony. Voľba určitého existujúceho middlewaru o ktorý agent-space oprieme, nám do istej miery robí jasno i v tom, aký formát pre tieto klony použiť. CORBA tu ponúka POA (portable object adaptor), RMI mashallovaný objekt, v prípade WS by sme použili XML dokument v tvare SOAP. S výnimkou RMI však všetky tieto možnosti majú ďaleko od jednoduchosti z hľadiska ich používania v rámci aplikačného kódu, preto je každopádne rozumné požadovať, aby sa na aplikačnej úrovni používala proste istá skupina objektov a na prezentačnej úrovni ju vnútorne mapovať na príslušný zvolený formát. Kľúčovú rolu pre toto mapovanie tu zohráva prítomnosť introspektívneho modelu (reflection model) na zvolenej platforme, čím rozumieme schopnosť získať za behu systému k ľubovoľnému objektu jeho popis – premenné, metódy, nadradenú triedu, implementované rozhrania, ... Objekty, ktoré možno do space-u vložiť, sa dajú identifikovať pomocou toho, že sú odvodené od určitej triedy, alebo že implementujú určitý interface. Pritom význam tohto dedenia či implementovania rozhrania bude len značkovací: bude dokladovať pre mapovací mechanizmus schopnosť objektu byť namapovaný. Iná možnosť (tú používame v príklade nižšie) je definovať mapovateľnosť sémanticky a schopnosť byť mapovaný detekovať počas samotného procesu mapovania.

Príklad č. 6

Jeden z možných spôsobov mapovania môžeme definovať nasledovnou statickou triedou:

```
import java.lang.reflect.*;
import java.util.*;
```

```
public class Mapping {

public static String Object2XML (String name, Object obj) {
  return Object2XML(name,obj,0,new ArrayList<Object>());
}

public static String Object2XML (String name, Object obj, int depth, List<Object> objs) {
  Class cl = obj.getClass();
  if (cl.isInterface()) return("");
  if (cl.isPrimitive() || isPrimitive(cl))
    return(tab(depth) + "<var type=\""+ className(cl) + "\" name=\"" + name + "\">" + obj +
           "</var>\n");
  String str;
  objs.add(obj);
  if (cl.isArray()) {
    Class itemCl = cl.getComponentType();
    str = tab(depth) + "<array type=\"" + className(itemCl) + "\" length=\"" +
          Array.getLength(obj) + "\" name=\"" + name + "\">\n";
    for (int i = 0; i < Array.getLength(obj); i++) {
      if (!objs.contains(Array.get(obj,i))) {
        str += Object2XML(name,Array.get(obj,i),depth+1,objs);
      }
    }
    str += tab(depth) + "</array>\n";
  }
  else {
    str = tab(depth) + "<object type=\"" + className(cl) + "\" name=\"" + name + "\">\n";
    for (;;) {
      Field[] field = cl.getDeclaredFields();
      for (int i = 0; i < field.length; i++) {
        field[i].setAccessible(true);
        if (field[i].getName().startsWith("this")) continue;
        try {
          if (!objs.contains(field[i].get(obj)))
            str += Object2XML(field[i].getName(),field[i].get(obj),depth+1,objs);
        }
        catch (Exception e) {
          str += tab(depth+1) + "<unknown type=\"" + className(field[i].getType()) + "\" name=\""
                      + field[i].getName() + "\"/>\n";
        }
      }
      Class supercl = cl.getSuperclass();
      if (supercl == null) break;
      if (supercl == cl) break;
      cl = supercl;
    }
    str += tab(depth) + "</object>\n";
  }
  objs.remove(obj);
  return(str);
}
```

```
public static boolean isPrimitive (Class cl) {
  String clName = cl.toString();
  return (
    clName.equals("class java.lang.Boolean")      || clName.equals("class java.lang.Byte")  ||
    clName.equals("class java.lang.Character")    || clName.equals("class java.lang.Class") ||
    clName.equals("class java.lang.Double")       || clName.equals("class java.lang.Float")  ||
    clName.equals("class java.lang.Integer")      || clName.equals("class java.lang.Long")  ||
    clName.equals("class java.lang.Short")        || clName.equals("class java.lang.String") ||
    clName.equals("class java.lang.Void")
  );
}

static String tab (int depth) {
  String ret = "";
  for (int i=0; i<depth; i++) ret += " ";
  return ret;
}

static String className (Class cl) {
  String ret = cl.toString();
  if (ret.startsWith("class ")) ret = ret.substring(6);
  if (ret.startsWith("java.lang.")) ret = ret.substring(10);
  return ret;
}
```

Uvažujeme potom, že máme definované nasledovné triedy (obe sú v podstate obyčajné štruktúry vyjadrené ako objekty – sú teda tzv. „simple objects“):

```
class Address {
  String street;
  int number;
  String city;
  public Address (String street, int number, String city) {
    this.street = street;
    this.number = number;
    this.city = city;
  }
}

public class Citizen {
  String name;
  int age;
  Address[] address;
  public Citizen (String name, int age, Address[] address) {
    this.name = name;
    this.age = age;
    this.address = address;
  }
}
```

A teraz mapujme jednu inštanciu triedy Citizen pomocou nášho mapovacieho mechanizmu:

```
public class MappingExample {
  public static void main (String args[]) {
    System.out.println(
      Mapping.Object2XML("Citizen Andy",
        new Citizen("Andy",33,new Address[]{
          new Address("Hudobna",6,"Banka"),
          new Address("Cavojskeho",1,"Bratislava")
        })
      )
    );
  }
}
```

Obdržíme nasledovný XML dokument:

```
<object type="Citizen" name="Citizen Andy">
 <var type="String" name="name">Andy</var>
 <var type="Integer" name="age">33</var>
 <array type="Address" length="2" name="address">
  <object type="Address" name="address">
   <var type="String" name="street">Hudobna</var>
   <var type="Integer" name="number">6</var>
   <var type="String" name="city">Banka</var>
  </object>
  <object type="Address" name="address">
   <var type="String" name="street">Cavojskeho</var>
   <var type="Integer" name="number">1</var>
   <var type="String" name="city">Bratislava</var>
  </object>
 </array>
</object>
```

Tento kód má rovnaký informačný obsah ako daný objekt, ale poľahky ho dostaneme cez TCP/IP socket na akúkoľvek platformu, kde ho môžeme:

- namapovať naspäť, ak je to rovnaká platforma
- namapovať na útvar podobného významu, ak je to možné
- pracovať s ním v tvare XML, ktorý má podporu na každej mysliteľnej platforme
- zviditeľniť ako XHTML v internetovom prehliadači pomocou XSLT

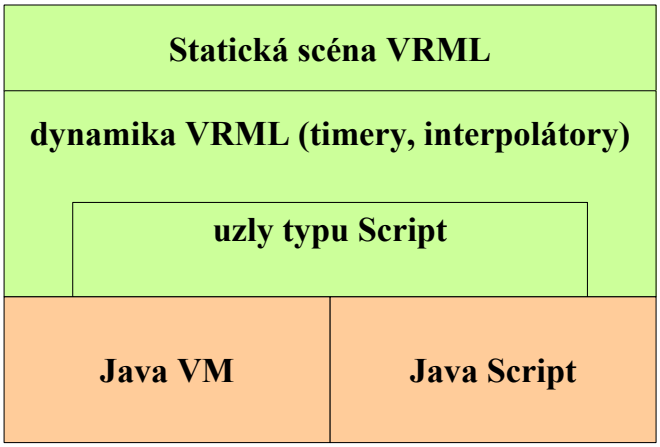
Či teraz do space uložíme priamo tento kód (a teda „podistribuujeme“ situáciu aj na lokálnej virtuálnej mašine) alebo ho bude používať len ako výstupný tvar, keď budú tieto dáta opúšťať lokálnu platformu, je voliteľné. Samozrejme, reálne implementácie by mali umožniť viac spôsobov naraz.

Agent-Space vo VRML

Hoci jazyk VRML (Virtual Reality Modelling Language, [Žára 1999]) je – čo do rozsahu – neporovnateľný s vyššie uvažovanými oblasťami programovania, pre naše potreby je veľmi dôležitý a preto mu venujeme samostatnú kapitolu. Totiž, keď sa opierame o biomimetické metódy a modelujeme nejaký biologický systém v počítači, výsledok je často závislý od kvality simulátora, ktorý používame. Najkvalitnejšie simulátory sú práve tie, ktoré sú založené na virtuálnej realite. Hoci virtuálna realita neznamená, že to musí byť práve VRML (v súčasnosti

prevládajú proprietárne binárne formáty), je predsa len základom tejto oblasti. Navyše je to platforma s voľne dostupnými implementáciami (CosmoPlayer, Blaxxun, Cortona).

VRML je jazyk postavený na rámcovej reprezentácii, akurát rámce nazýva uzlami a rubriky atribútmi. Je pôvodne určený predovšetkým na modelovanie statických 3D scén, pozostávajúcich jednak z objektov ideálnych tvarov (kváder, guľa, kužeľ, valec), jednak z nepravidelných objektov definovaných napríklad pomocou množiny rovných povrchových plôch tvaru mnohouholníka. Obsahuje však aj dynamické prvky a to jednak vstavané, ktoré umožňujú vyjadriť dynamiku priamo vo VRML (má na to špeciálne uzly), jednak otvorené, ktoré umožňujú doplniť schopnosti tohto jazyka o dynamiku naprogramovanú v jazyku JavaScript alebo Java (pomocou uzla Script) (pozri Obrázok č. 35). Práve túto programovateľnú dynamiku Agent-Space dokáže výrazne obohatiť. Samotné VRML disponuje totiž vhodnou modulárnosťou kódu, ale kooperáciu medzi modulmi prenecháva na generovanie udalostí, ktoré sú smerované z jedného modulu do druhého. Vďaka tomu sa prakticky nedá urobiť zdieľanie nejakej informácie viacerými modulmi, ak ju generuje viac než jeden modul. Na to by bolo nutné urobiť v scéne nejaký tajný objekt, ktorý túto informáciu obsahuje ako hodnotu atribútu. A to nie je veľmi pekné.



Obrázok č. 35 Prostriedky dynamiky vo VRML

Náročnú kooperáciu viacerých modulov potrebujeme nastoliť práve vtedy, keď modelujeme nejakú mobilnú kreatúru, robot či živočícha, pohybujúcu sa v 3D scéne a riadenú automatizovaným riadiacim systémom. VRML je vytvorené na generovanie takéhoto pohybu, ale len z vopred zadaných dát. Pokiaľ chceme tieto dáta priebežne generovať z interakcie modelovanej kreatúry a jej okolia v scéne, dostávame sa na samotnú hranicu možností VRML. V prvom rade sa nám tu zídu rozšírenia VRML umožňujúce vnímať stav okolia modelovaného objektu (t.j. nielen stav pomyselného užívateľa – tzv. avatara, čo je súčasťou štandardu): detekovanie kolízie, zisťovanie vzdialenosti najbližšej prekážky v určitom smere a pod. My sme pre naše účely použili implementáciu VRML v plugine Cortona pre webový prehliadač IE, ktorá toto poskytuje. Jednoznačne však oceníme, ak budeme mať aj podporu vzájomnej kooperácie medzi modulmi realizujúcimi dynamiku. No a na túto podporu je implementácia agent-space priam stvorená [Lúčny 2004b].

VRML samotné s niečím takým nepočíta a je len šťastnou náhodou, že agent-space tu vôbec ide implementovať. Príčinou je púhy fakt, že implementácia VRML, ktorá by púšťala pre každý uzol Script samostatnú JVM, by bola veľmi neefektívna (hoci by nebola v rozpore s normou). Všetky scripty preto bežia pod jedinou JVM a my môžeme využiť jej statickú pamäť na realizáciu space-u. Brzdí nás v tom len fakt, že nami zvolená konkrétna implementácia je

príliš závislá na platforme MS Windows a používa MicroSoft JVM, ktorej vývoj ostal na verzii Java 1.1. Vďaka tomu nemôžeme použiť väčšinu dnes bežných konštruktov jazyka Java a musíme sa uspokojiť s primitívnejšou implementáciou.

Hoci virtuálny model vo VRML pôsobí dojmom reálneho času, od systému reálneho času má veľmi ďaleko. Čas je totiž len emulovaný tým spôsobom, že prehliadač prekresluje 3D scénu maximálnou možnou rýchlosťou. Nemožno tu teda získať pravidelné časové impulzy, iba si na časovej osi kontrolovať, kam padol okamih aktuálneho prekreslenia scény. Nemožno sa teda ani spoľahnúť, že do každej zvolenej periódy padne určité prekreslenie. Naštastie s výkonným počítačom a kvalitnou grafickou kartou sa dajú voliť celkom slušné periódy časovania. Avšak konkrétna hodnota závisí od zložitosti scény. Pri súčasnom hardware (rok výroby 2003) sa pri jednoduchších scénach dá spoľahnúť na časovanie zhruba 20Hz, pri zložitých len okolo 4Hz. To je na hranici použiteľnosti, každopádne však možne produkovať simuláciu spomalenú oproti reálnemu času v určitom pomere. Vzhľadom na Moorov zákon možno v tomto ohľade do budúcnosti pozerať optimisticky. Každopádne bude pri VRML vždy platiť, že únosnú frekvenciu si treba zakaždým experimentálne preveriť.

Prostredie implementujeme veľmi podobne ako v prípade bežnej implementácie v Jave. Bude teda poskytovať všetky bežné funkcie space-u. Rozdiel bude spočívať v synchronizácii. Keďže jednotlivé scripty púšťa VRML plugin v prehliadači, spúšťa ich načisto sekvenčne a nemusíme mať obavy zo súčasného prístupu z viacerých vlákien. Na druhej strane však budeme musieť použiť úplne iný blokovací mechanizmus pre agenty. Pre prostredie z toho vyplýva, že sa buď zaobídeme bez triggrov, alebo, že ich budeme implementovať prostredníctvom generovania udalosti v 3D scéne (teda vo VRML, nie v JVM). Podobne na timery nebudeme môcť používať schopnosti JVM, ale VRML. VRML timery veľmi dobre podporuje, hoci ich realizácia je úplne iná: nie timer generuje udalosť, ale pri kreslení sa prehliadač pozerá, kde sa trafil súčasný okamih do periódy timeru. My budeme chytať stav, keď sa trafí za nábežnú hranu a v tom momente budeme vykonávať jeden prechod cyklom agenta. Čo sa týka časovej platnosti dát, ostávame pri taktike, že dáta odstránime z prostredia pri prvom čítaní po uplynutí ich platnosti. Avšak čas budeme musieť počítať spôsobom typickým pre VRML, teda v reálnych číslach vyjadrujúcich sekundy od začiatku simulácie. Takto môžeme vyvinúť Space, ktorý bude singleton pracujúci nad statickou pamäťou:

```
public class Space {
  ...
  public static Space getInstance() {...}
  protected Space() {...}
  public void write (String name, Object value, double validFrom, double validTo) {...}
  public void write (String name, Object value, double validFor) {...}
  public void write (String name, Object value) {...}
  public Object read (String name) {...}
  public Object read (String name, Object def) {...}
  public void attachTrigger (String mask, Triggerable agent, int type) {...}
  ...
}
```

Rozhranie Triggerable bude klasické:

```
interface Triggerable {
  void trigger();
}
```

Funkciu trigger budeme volať na zobudenie príslušného agenta ako predtým, avšak v nej budeme volať funkcie na generovanie udalostí vo VRML, nie synchronizačný mechanizmus JVM. Problémom je nasmerovanie udalosti na správny agent a neostáva iné, než tieto udalosti nasmerovať do špeciálneho scriptu - "triggerovača", ktorý všetky tieto triggre vezme dohromady a nasmeruje kam patria. Aby to však mohol urobiť, trigger musí so sebou priniesť informáciu, pre koho je určený. A na to mu bude v agentovi slúžiť funkcia getName(), poskytujúca určité jedinečné meno, ktoré agentovi priradíme priamo v scéne.

Používajúc takýto space, môžeme agenty realizovať ako špecifické scripty VRML. Pritom budeme musieť nielen definovať ich triedu pre JVM, ale aj sémantický obsah uzlov vo VRML, ktoré im prislúchajú. Definovanie triedy pre konkrétny agent si uľahčíme definovaním všeobecnej triedy Agent, od ktorej všetky agenty odvodíme:

```
import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class Agent extends Script implements Triggerable {

 private Space space;
 private SFString triggerEvent;
 public static double timestamp;
 private String name;

 public void initialize() {
  space = Space.getInstance();
  triggerEvent = (SFString) getEventOut("trigger");
  name = ((SFString) getField("name")).getValue();
  timestamp = 0.0;
  init();
 }

 public void processEvent(Event e) {
  if (e.getName().equals("tick")) {
   double t = ((ConstSFTime)e.getValue()).getValue();
   if (t > timestamp) timestamp = t;
   sense_select_act();
  }
  else sense_select_act(e);
 }

 public void init() {
 }

 public void sense_select_act() {
 }

 public void sense_select_act(Event e) {
 }

 public String getName() {
  return name;
 }
```

```
 public void trigger() {
  triggerEvent.setValue(getName());
 }

 public double time() {
  return timestamp;
 }

 public void write (String name, Object value, long validFrom, long validTo) {
  space.write(name,value,validFrom,validTo);
 }

 public void write (String name, Object value, long validFor) {
  space.write(name,value,validFor);
 }

 public void write (String name, Object value) {
  space.write(name,value);
 }

 public Object read (String name) {
  return space.read(name);
 }

 public Object read (String name, Object def) {
  return space.read(name,def);
 }

 public void setTrigger(mask) {
  space.attachTrigger(mask,this,0);
 }

}
```

Každej inštancii XY nejakej konkrétnej špecializácie tejto triedy musíme potom v scéne vyčleniť script tvaru:

```
DEF AGENTXY Script {
 ...
 field SFString name „AgentXY“
 eventIn SFTime tick
 eventOut SFString trigger
 url "AgentXY.class"
 mustEvaluate TRUE
}
```

Tento uzol, reprezentujúci agent v scéne, musí byť ďalej budený časovačom s určitou periódou v sekundách <PER>, pričom tento časovač je tiež súčasťou scény (viaceré agenty ho však môžu zdieľať):

```
DEF TIMERXY TimeSensor {
 cycleInterval <PER>
 loop TRUE
}
```

Ďalej musí byť súčasťou scény príslušné smerovanie udalosti pre budenie agenta nábežnou hranou časovača:

```
ROUTE TIMERXY.cycleTime TO AGENTXY.tick
```

ako aj smerovanie triggra do „triggerovača“:

```
ROUTE AGENTXY.trigger TO TRIGGERFACTORY.trigger
```

a samotný „triggerovač“, ktorý musíme napísať na mieru všetkých agentov XY1, XY2, ..., ktoré majú byť triggrované:

```
DEF TRIGGERFACTORY Script {
 eventIn SFString trigger
 field MFNode triggerable [ USE AGENTXY1 USE AGENT XY2 … ]
 url "vrmlscript:
   function trigger (name) {
     var i;
     for (i=0; i<triggerable.length; i++)
       if (name == triggerable[i].name)
         triggerable[i].tick = 0.0;
   }
 "
}
```

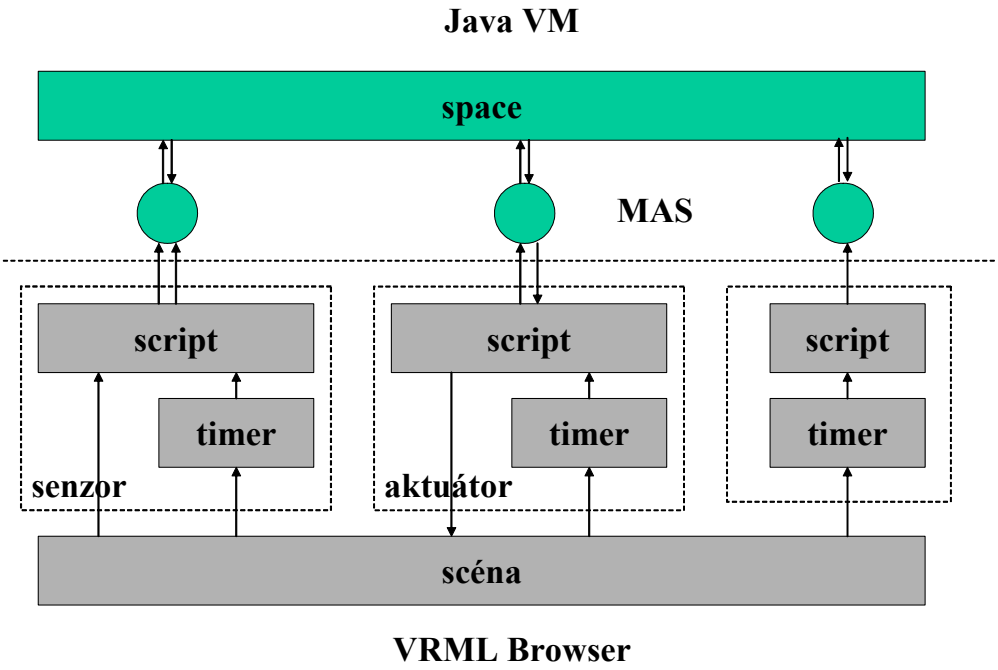
Keď teraz budeme chcieť urobiť agent, ktorý pracuje výlučne nad prostredím, stačí urobiť príslušnú inštanciu. S tým rozdielom, že z metódy init() budeme nastavovať iba triggre, nakoľko timery sú zadefinované vo VRML scéne (dalo by sa to formálne doplniť, keby sme veľmi chceli, nakoľko ľubovoľný uzol – a teda i timer – sa dá z javy vygenerovať a pridať do scény).

```
public class AgentXY extends Agent {

 public void init() {
   ... setTrigger("mask"); ...
 }

 public void sense_select_act() {
   ... obj1 = read("name1",def1); ...
   ...
   ... write("name2",new Obj2(val2));
 }

}
```

Keby sme však mali len takéto agenty, bolo by nám to celé nanič. Niektoré z agentov musia byť aj prepojené so scénou. Tie, ktoré budú reprezentovať senzory, budú mať v zápise uzla script definovanú aj vstupnú udalosť (eventIn), do ktorej sa bude v scéne smerovať udalosť pochádzajúca zo štandardnej dynamiky VRML. Túto naša mašinéria obslúži funkciou sense_select_act(Event e), v ktorej možno z parametra zistiť nejaký aktuálny údaj zo scény. Metóda sense_select_act(Event e) sa teda volá nie pravidelne, ale iba keď sa v scéne vygeneruje príslušná udalosť. Formálne musíme zachovať takémuto agentovi aj vstupnú

udalosť tick, v reáli ale nemusíme do nej žiadny výstup z časovača nasmerovať. Alternatívne možno miesto eventIn v scripte definovať field, ktorú cez USE previažeme s určitým uzlom siete, v init() si získame prístup k tomuto uzlu a túto hodnotu potom priebežne zisťujeme v sense_select_act(), keď príde tik od časovača.

Aktuátory urobíme o niečo ľahšie, pričom tu opäť máme dve možnosti. Prvá je, že generujeme výstupnú udalosť, ktorú potom v scéne smerujeme do príslušného uzla. Nato v init() získame prístup k reprezentácii tento udalosti a potom v sense_select_act() nastavujeme jej hodnotu. Druhá možnosť je navlas rovnaká ako v prípade senzorov, akurát miesto zisťovania hodnoty atribútu určitého uzla, ku ktorému získame prístup, túto hodnotu nastavujeme. Takto vybudujeme architektúru, ktorú zviditeľňuje Obrázok č. 36.



Obrázok č. 36 Riadiaci systém na báze agent-space konajúci v 3D scéne vo VRML

Príklad č. 7

Ukážeme si najjednoduchší príklad, aký sa vôbec dá vymyslieť. Uvádzame práve taký, lebo aj toho kód je dosť zložitý. Ide o riadiaci systém guličky, ktorý simuluje jej fyzikálne správanie: pri kolmom náraze na prekážku sa odrazí. Značná časť kódu sa opiera o schopnosti Cortony detekovať náraz, ktoré sú nad rámec VRML, takže pri inom prehliadači by sa implementovali inak, prípadne by sa vôbec nedali implementovať. Konkrétne: náraz sa tu detekuje pomocou objektu Collidee, ktorý je rozšírením knižníc JavaScriptu pod Cortonou. Keďže tento objekt nie je k dispozícii z Javy a na druhej strane JavaScipt nedokáže pristupovať do statickej pamäti JVM, kde máme space, musíme informáciu o náraze vyniesť do scény ako udalosť a nasmerovať ju do objektu (tzv. adaptéra), ktorý reprezentuje daný senzor v Jave.

```
#VRML V2.0 utf8

DEF OBSTACLE Shape {
 geometry Box {}
}
```

```
DEF MOVINGBODY Transform {
 translation -5 0 0
 children DEF BODY Shape {
  geometry Sphere {}
 }
}

DEF TIMER TimeSensor {
 loop TRUE
 cycleInterval 0.1
}

DEF SENSOR Script {
 eventIn SFTime tick
 field SFNode holder USE MOVINGBODY
 field SFNode body USE BODY
 eventOut SFInt32 bump
 url "vrmlscript:
  function tick() {
   var c = new Collidee();
   c.body = body;
   c.position = new SFVec3f(holder.translation);
   c.orientation = new SFRotation(holder.rotation);
   if(c.moveTo(holder.translation, holder.rotation)) bump = 0;
   else bump = 1;
  }"
}

DEF SENSORADAPTOR Script {
  field SFString name „Sensor“
  eventIn SFTime tick
  eventOut SFString trigger
  url "Sensor.class"
  mustEvaluate TRUE
  eventIn SFInt32 bump
}

DEF CONTROLLER Script {
  field SFString name „Controller“
  eventIn SFTime tick
  eventOut SFString trigger
  url "Controller.class"
  mustEvaluate TRUE
}

DEF ACTUATOR Script {
  field SFString name „Actuator“
  eventIn SFTime tick
  eventOut SFString trigger
  url "Actuator.class"
  mustEvaluate TRUE
  field SFNode actuator USE MOVINGBODY
}
```

```
ROUTE TIMER.cycleTime TO SENSOR.tick
ROUTE SENSOR.bump TO SENSORADAPTOR.bump
ROUTE TIMER.cycleTime TO CONTROLLER.tick
ROUTE TIMER.cycleTime TO ACTUATOR.tick

# Táto vrml scéna používa nasledovné class-y implementované v jazyku Java

import vrml.*;
import vrml.field.*;
import vrml.node.*;

public class Sensor extends Agent {

  public void sense_select_act(Event e) {
    if (e.getName().equals("bump")) {
     ConstSFInt32 bump = (ConstSFInt32) e.getValue();
     write("bump",bump);
    }
  }

}

public class Controller extends Agent {

  private SFVec3f dir;

  public void init() {
    dir = new SFVec3f(0.1f,0f,0f);
  }

  public void sense_select_act() {
    ConstSFInt32 bump = (ConstSFInt32) space.read("bump", new ConstSFInt32(0));
    if (bump.getValue() > 0) dir.setValue(-dir.getX(),-dir.getY(),-dir.getZ());
    space.write("motion",dir);
  }

}

public class Actuator extends Agent {

  private SFVec3f pos;

  public void init() {
    SFNode node = (SFNode) getField("actuator");
    Node base = (Node) node.getValue();
    pos = (SFVec3f) base.getExposedField("translation");
  }

  public void sense_select_act {
    SFVec3f motion = (SFVec3f) space.read("motion",new SFVec3f(0f,0f,0f));
    pos.setValue(pos.getX()+motion.getX(),pos.getY()+motion.getY(),pos.getZ()+motion.getZ());
  }

}
```

Keď teraz spustíme VRML scénu v prehliadači, uvidíme guličku, ktorá letí na kváder a odrazí sa od neho. Toto samozrejme ide urobiť aj v štandardnej dynamike VRML. Toto riešenie sa však dá upraviť tak, že kváder zavesíme na polohový senzor (tým pádom s ním môžeme pohybovať). Keď potom pohneme kvádrom tak, že guličká do neho nenarazí, guličká preletí. Keď ho necháme v ceste, guličká sa odrazí. A toto už so štandardnou dynamikou VRML nedokážeme.

V uvedenom príklade vystupuje iba jeden senzor, jeden obyčajný agent a jeden aktuátor. Pri zložitejšej úlohe máme samozrejme z každého viac kusov. Dajú sa robiť pomerne zložité veci. Vo VRML sme napríklad implementovali robot prehľadávajúci kancelárske priestory (išlo o reimplementáciu známeho robota ALLEN). Pri nich využívame ďalšie rozšírenia VRML, ako je napríklad meranie vzdialenosti najbližšej prekážky vo zvolenom smere, ktoré Cortona podporuje pre zmenu na úrovni Javy.

Záver kapitoly

Dalo by sa ešte veľa hovoriť o implementačných záležitostiach, nakoľko tu nejde o vec triviálnu, ale už by to bolo nad rámec, ktorý sme ochotní pripustiť. Totiž, napriek tomu, že je táto kapitola najrozsiahlejšia, nepovažujeme ju za ťažiskovú. Pojednáva v podstate o dočasných hodnotách (kto vie ako sa bude agent-space implementovať o 10 rokov...), pričom tie večné sú práve mimo nej. Vyzneli by však príliš platonicky, keby čitateľ nedostal do ruky návod ako ich pretaviť do reálnych aplikácii. Ak by sa k tomu niekto odhodlal, táto kapitola mu významne pomôže a v tom je jej zmysel. Som ďaleko od názoru, že nemá význam sa zapodievať čisto filozofickými otázkami, avšak táto práca je každopádne o programovaní reálnych aplikácií. Preto agent-space má v tejto dobe aj komerčné implementácie. Prvú sme vytvorili v roku 1996 na platforme QNX4 pod názvom MsAgent (pozor to Ms tu znamená MicroStep, nie MicroSoft) a hoci nikdy nebola šírená ako krabicový software, použili sme ju v rade komerčne úspešných produktov. V súčasnosti prebieha vývoj druhej implementácie a to na platforme Java. Okrem toho bolo vytvorených niekoľko nekomerčných implementácii pre účely výskumných projektov o ktorých ešte bude reč.

[13] t.j. a) aby bol kód realizujúci aplikačnú logiku čo najviac oddelený od kódu, ktorý je vynútený aplikačným rozhraním technických prostriedkov, ktoré používa, b) aby sa v kóde neopakovali rovnaké charakteristické vzory a sekvencie

[14] lebo tým doženieme systém k prepínaniu procesov (v procesore) „proti ich vôli“, čo je menej efektívne ako keď sa prepínajú „dobrovoľne“, napr. následkom zavolania blokujúcej primitívy

[15] echidna.sourceforge.net

VI. Význam Agent-Space pre umelú inteligenciu

Na tomto mieste výkladu už máme do detailov predstavenú architektúru agent-space ako určitý nástoj. Zameriame sa teraz na otázku ako a na čo ho použiť. Tak, ako sa pri riešení problému nájdenia minima zo sto čísel nedá pochopiť na čo sú v programovaní dobré objekty, tak ani nikto nepochopí, na čo sú dobré agenty, kým sa nezačne zaoberať vhodnými problémami. Obvykle sa v súčasnosti za vhodné považujú distribuované problémy, kde ide o budovanie systémov zložených z mnohých uzlov siete

- ktorej štruktúra sa môže meniť
- kde jednotlivé uzly môžu pribúdať a ubúdať
- kde sa má nájsť nejaký súlad ako použiť zdroje, ktorými tieto uzly disponujú k prospechu celku

(Často používaným príkladom je riadenie výrobného procesu v továrni, schopné adaptovať sa na zmeny v požiadavkách a podmienkach výroby.) Je to však skôr preto, že na túto problematiku existuje spoločenská objednávka. Ja mám rád slogan „the computer is the network“ a nevidím podstatu problému v prítomnosti počítačovej siete. Vďaka tomuto názoru môžem pracovať na rádovo vyšších časových frekvenciách a nemusím zápasiť s Deutschovými klammi[16]. Ako som už spomenul, za vhodné problémy považujem dostatočne zložité riadiace a monitorovacie systémy (napríklad riadiaci systém mobilného robota alebo simulovaného hmyzu). Pritom upozorňujem, že hoci názvy sú dva, oblasť je jediná. (Keď je v aute systém čo zabrzdí, keď detekuje, že hrozí zrazenie chodca, tak hovoríme o riadiacom systéme, zatiaľ čo keď ten systém iba zapíska a upozorní vodiča, že treba zabrzdiť, hovoríme o systéme monitorovacom. Netreba myslím dodávať, že kumšt je to zhruba rovnaký).

Keď hovoríme o „dostatočne zložitých“ problémoch, uvedomujeme si, že je to pojem vágny. Našťastie tu však máme druhý vágny pojem „umelej inteligencie“, ktorá všeobecne slúži na označenie problémov, ktoré nespadajú pod bežnú rutinu: príroda na ne riešenia vynašla, ale my ešte celkom nie. Budeme sa teda zaoberať práve týmito problémami a ukážeme, že agenty nie sú vhodné len kvôli potrebe

- púšťať riešenie na viacerých CPU
- presťahovávať programy z jednej CPU na inú

a podobne, ale že **z agentov dokážeme stavať riadiace systémy, ktoré môžeme**

- **nastaviť tak, aby sa adekvátne správali v určitom prostredí**
- **adaptovať na zmeny tohto prostredia**.

Pokiaľ poslednú vetu porovnáme s prevládajúcou psychologickou definíciou inteligencie[17], chýba nám len jedna podstatná drobnosť: aby sa adaptovali samé. Takýto mechanizmus v tejto práci čitateľ nenájde. Podľa jej autora je však diskutabilné, či také niečo vôbec v prírode existuje. Isté prejavy učenia a tvorivosti je totiž možné pripísať skôr zobudeniu sa vrodených schopností, než získaniu schopností nových. Je možné aj to, že v prírode sme svedkami získania nových schopností výlučne pri krížení a mutácii, teda pri vzniku nových jedincov. Radi by sme sa na učenie pozerali tým spôsobom, že sa určité dáta, ktoré v jedincovi neboli, nasťahovali do určitého vopred pripraveného chlievika. Podobne na tvorivosť, že v určitom vopred pripravenom chlieviku sa toľko náhodne generovalo, až to zabralo. Jedinec tak získal z vonkajšieho pohľadu novú schopnosť, ale keby nebolo toho vopred pripraveného chlievika, nič také by sa nestalo. V skutočnosti si jedinec iba nakŕmil dátami mechanizmus, ktorým už bol vybavený. A to nemám na mysli nejaký všeobecný mechanizmus. Je napríklad známe, že v jednej rodine v Anglicku sa vyskytuje mutácia jediného génu, ktorého nositeľ sa nenaučí gramatickú stavbu vety [Hamer – Copeland 2003] – hoci vie komunikovať, rozpráva asi ako indiáni z kníh od Karla Maya. Žiadna iná neschopnosť sa u nich neobjavuje. To podľa mňa svedčí zreteľne o tom, že človek má niekde v sebe chlievik presne určený na tento účel. A keď

práve tento chlievik chýba, práve túto vec sa človek nenaučí. Tým nechcem povedať, že na jednu schopnosť nemôže byť pripravených viac chlievikov, ani že jeden pripravený chlievik nemôže z princípu slúžiť u jedného jedinca jednej a druhého inej schopnosti[18].

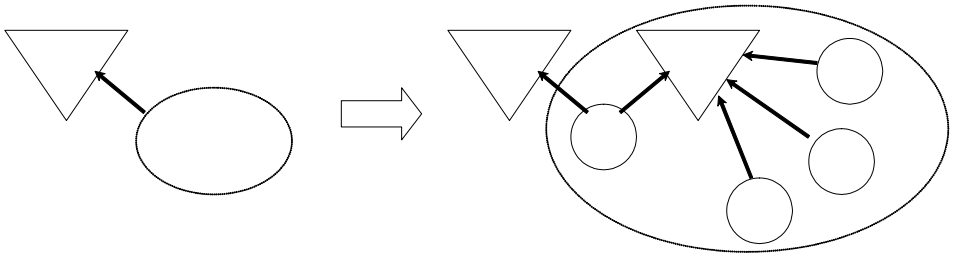
Na základe týchto predpokladov si preto dovoľujem vyhlásiť, že v porovnaní schopností agentov s definíciou inteligencie toho nechýba až tak veľa a že **agenty predstavujú** vlastne **vhodný prostriedok na tvorbu „inteligentných" systémov**.

V ďalšom sa zameriame na analógie medzi umelými systémami postavenými podľa architektúry agent-space a systémami živými. Budeme ich skúmať jednak v rovine všeobecnej, jednak na konkrétnych projektoch, ktoré sme realizovali. Pritom – napriek filozofovaniu – nebudeme nikdy zabúdať na posilňovanie našej reálnej schopnosti tvoriť umelé systémy podobné živým. Aj keby sme teda pri všeobecných úvahách skĺzli niekam preč, vždy ostane niečo, čo je hmatateľne užitočné.

Biologická relevantnosť

Popíšme teraz našu predstavu o tom, v ktorých vlastnostiach sa architektúra agent-space podobá živým systémom.

Hierarchia: Ako sme už v úvode spomenuli, východiskovým bodom tejto architektúry bola snaha zachytiť hierarchiu, ktorú v prírode pozorujeme, či presnejšie: ktorú používame na jej opis. Molekuly vytvárajú membrány a organely, bunky vytvárajú tkanivá, tkanivá orgány a orgány organizmus. Na každej úrovni máme určité jednotky, ktoré medzi sebou interagujú, pričom charakter tejto interakcie je lokálny. Problémom je, že hoci veríme, že deje na určitej úrovni sú dôsledkom dejov na nižšej úrovni, tento vzťah až na jednotlivé prípady zanedbávame a postulujeme zákonitosti v rámci každej úrovne. Keď ale takýto reálny svet začneme prenášať do počítača, kde nestačí prírodu pochopiť, ale kde je ju potrebné modelovať, potrebujeme veľmi presne stanoviť ako deje na nižších úrovniach opisu vplývajú na vyššie úrovne. V tomto opise musíme byť veľmi presný, lebo v počítači na rozdiel od reálneho sveta nie sú zákony prírody zabudované a všadeprítomné. To my ich musíme zabudovať a sprístupniť. Architektúra agent-space povstala z pragmatického riešenia tejto požiadavky. Jej ústredným princípom je **premena hierarchie na zapuzdrenie** (Obrázok č. 37). Podľa toho princípu si každý agent, konajúci v prostredí vyššej úrovne, možno predstaviť ako multiagentový systém nižšej úrovne. Tento pozostáva z agentov medzi ktorými prebieha interakcia pomocou iného prostredia, ktoré nie je prístupné vyšším úrovniam. Kľúčovú úlohu potom zohráva niekoľko málo agentov, ktoré operujú nad oboma prostrediami a prenášajú tak stav štruktúr používaných na opis nižšej úrovne do stavu štruktúr používaných na opis vyššej úrovne[19].



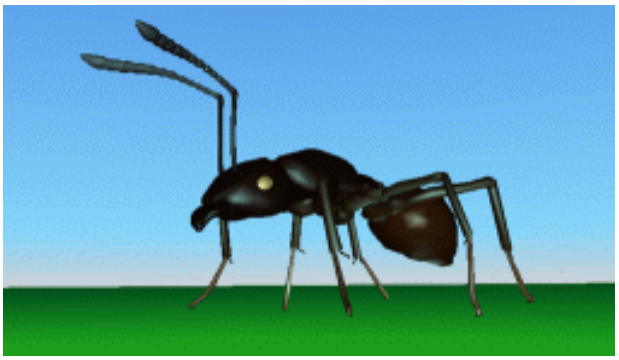
Obrázok č. 37 Premena hierarchie na zapuzdrenie

Tento pohľad na hierarchiu je pragmatický v tom, že nevyžaduje dotiahnuť vzťah medzi vyššou úrovňou a nižšou ad absurdum, kedy na vyššej úrovni nič nové nepostulujeme, lebo opis nižšej úrovne je tak dokonalý, že z neho všetko čo sa odohráva na vyššej úrovni vyplýva. Technicky výhodný je v tom, že architektúra v sebe neobsahuje žiadne špeciálne konštrukty na definovanie hierarchie. Hierarchia je prítomná len v mysli návrhára a technicky je zachytená len v tom, že isté skupiny blokov sú prístupné len istým skupinám agentov (tým, ktoré patria na tú istú hierarchickú úroveň, alebo zabezpečujú prepojenie s nižšou hierarchickou vrstvou), teda zapuzdrením. Vývojár systém implementuje v rozvinutej, nezapuzdrenej forme a hierarchickú, zapuzdrenú formu používa ako prostriedok, ktorý mu pomáha pri návrhu systému. Takto je možné pri návrhu pracovať s ľubovoľným počtom úrovní opisu systému, teda presne tým spôsobom, akým odhaľujeme zákonitosti prírody. Pritom pre každú úroveň používane rovnaké softwarové prostriedky. To je v súčasnosti výnimočné: v modeloch, ktoré poznáme[20] sa buď pracuje na jedinej úrovni opisu, alebo sa pre každú používajú špecifické prostriedky. S našou architektúrou môžeme preto pravdepodobne vytvárať zložitejšie a zaujímavejšie modely.

Príklad č. 8

Predstavme si, že by sme chceli modelovať pohyb mravca[21] (Obrázok č. 38). Telo mravca vytvorené z jednotlivých hnátov by sme vyjadrili vo VRML a rovnako by sme vyjadrili scénu, v ktorej sa pohybuje. Aby sme teraz takýto model oživili, musíme implementovať virtuálny svet s tromi úrovňami opisu:

- na fyzikálnej úrovni musíme implementovať pôsobenie tiažovej sily, prejavujúce sa pohybom modelu do rovnovážnych stavov (napríklad pád na brucho pri zdvihnutí všetkých nôh) a silovým pôsobením v kĺboch (ak na končatinu pôsobí príliš veľká sila, poddá sa)
- na biologickej úrovni musíme implementovať telo, napríklad povahu svalov (kĺbom hýbu u hmyzu poväčšine dva alebo tri svaly), tvar kĺbu definujúci stupne voľnosti, senzory ťahu a tlaku a pod.
- na neurálnej úrovni musíme implementovať samotný riadiaci systém, ktorý prijíma údaje zo senzorov a generuje príkazy pre svaly.



Obrázok č. 38 Modelovanie umelého pohybu

Je tu vhodné používať hierarchiu: na hornej úrovni máme ako dáta vzruchy pre svaly, na strednej sily ktoré pôsobia v kĺboch a na spodnej úrovni uhly ohybu a ich umiestnenie v priestore. Pri konverzii jednej reprezentácie na druhú dochádza k obojsmernému ovplyvňovaniu, napríklad náraz na prekážku znemožní svalu napnúť sa požadovaným spôsobom, zatiaľ čo normálne práve toto napnutie určuje polohu končatiny v priestore. Toto ovplyvňovanie realizujú práve tie agenty, ktoré pracujú na viacerých úrovniach opisu. Použitie

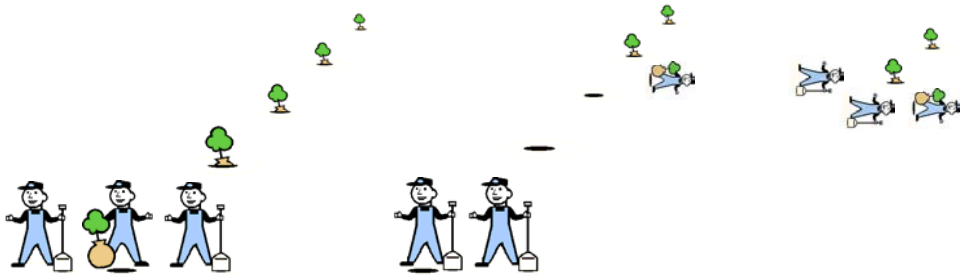
viacerých úrovní vedie v princípe k vyjadreniu rovnakej informácie viacerými spôsobmi so zabezpečením prevodu jedného formátu do druhého. Na prvý pohľad sa to môže zdať zbytočné – veď prečo by sme nemohli pracovať nad jediným formátom? Odpoveď spočíva v tom, že raz je nám výhodnejšie použiť jeden a inokedy druhý formát. Programovanie modelu sa vďaka tomu podobá metóde akou poznávame jeho prírodný vzor. Samozrejme ak raz uznáme, že hierarchia je osožná, netreba dodávať, že je veľmi prospešné implementovať všetky úrovne rovnakými softwarovými prostriedkami. To architektúra agent-space umožňuje. Je na druhej strane pravdou, že zavedenie jednotného nástroja znemožňuje oprieť sa o už existujúce špecifické nástroje. Napríklad pre modelovanie pohybu mravca by sme mohli využiť ODE, ktoré už je k dispozícii. Preto tvorba prvých modelov s jednotnými prostriedkami vyžaduje viac práce (a to záujemcov odrádza), nič menej to neznamená, že by to bola horšia metóda.

Diverzifikácia: Hoci štruktúra systému je na každej úrovni normalizovaná, charakter dát s ktorými sa pracuje nie je vôbec vymedzený. Dokonca sa počíta s tým že bude rôzny. Za normalizáciu štruktúr teda neplatíme normalizáciou reprezentácie, ako je tomu napríklad pri neurónových sieťach[22]. To je presne naplnenie myšlienky Marvina Minského, ktorý sa v [Minsky 1986] vyjadril, že cieľom umelej inteligencie by nemalo byť hľadanie optimálnej reprezentácie poznatkov, ale optimálnej organizácie, ktorá umožňuje v systéme používať rôzne spôsoby reprezentácie. Je pravdepodobné, že aj príroda používa rôzne spôsoby. Aspoň zatiaľ v tomto smere nikto žiadny kameň mudrcov neobjavil.

Decentralizácia: Živým systémom spravidla pripisujeme vlastnosť masívneho paralelizmu. Vskutku mnohé pozorovania svedčia o tom, že ak živý systém lokálne poškodíme, nikdy neprejde do úplnej pasivity. To nasvedčuje, že v systéme nie sú prítomné (centrálne) moduly, ktoré by budili k aktivite ostatné. V systéme môžu byť len moduly, ktoré produkujú tak dôležité dáta, že aktivita ostatných modulov bez nich nevedie k efektu v globálnom správaní systému. Žiadna časť systému teda nikdy neprestane byť aktívna, potlačený môže byť len jej vplyv na výsledné správanie. Je len našou hypotézou, že toto sú vlastnosti živých systémov, celkom nepochybne sú to však vlastnosti architektúry agent-space.

Príklad č. 9

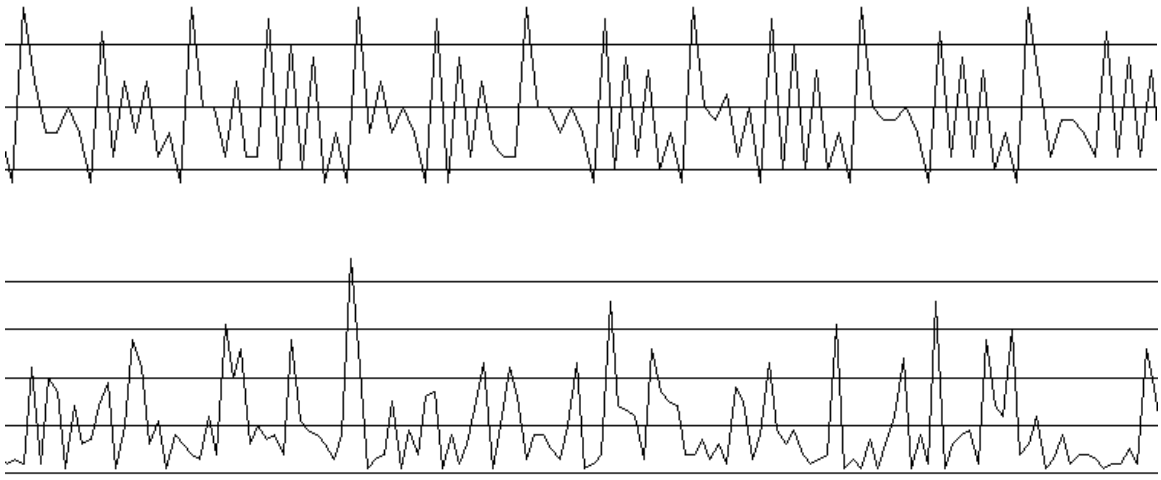
Priblížme si povahu agent-space na jednom úsmevnom príklade. Predstavme si, že máme urobiť systém na sadenie stromčekov. Povedzme, že bude pozostávať z troch modulov: kopania jamky, kladenia stromčeka a zahrabania. Otázka teraz spočíva v tom ako budeme organizovať tieto moduly do systému. Klasicky by sme to urobili zrejme centrálnym riadením, ktoré by cyklicky budilo k činnosti moduly v poradí: kopanie jamky, kladenie stromčeka, zahrabanie. S agent-space by sme to mohli urobiť napríklad tak, že načasujeme tri agenty, aby sa v čase t = 3k kopala jamka, v čase t = 3k+1 kládol stromček a v čase t = 3k+2 zahrabávalo. Tak, alebo onak, za normálnych podmienok je výsledok rovnaký: oba systémy sadia stromčeky. Rozdiel medzi nimi vypláva na povrch až v momente, keď napríklad pokazíme modul na kladenie stromčeka. Klasický systém s centrálnym riadením narazí pri vykonávaní tohoto modulu na problém a buď skončí s vrátením nejakého chybového kódu, alebo sa zasekne, zacyklí a pod. Každopádne prestane nielen sadiť stromčeky ale aj produkovať akúkoľvek činnosť. Naproti tomu decentralizované riešenie bude ďalej kopať jamky a ďalej ich bude zahrabávať, akurát bez stromčeka ☺ (Obrázok č. 39). Nechceme tým pravda povedať, že to druhé riešenie je lepšie, len že sa viac ponáša na prírodu - je biologicky relevantnejšie.



Obrázok č. 39 Porovnanie poruchy v decentralizovanom a centrálne riadenom systéme

Naľavo bezporuchový beh, v strede porucha decentralizovaného systému, vpravo porucha systému centrálne riadeného

Iregularita: V systéme vybudovanom podľa architektúry agent-space sú určité prvky voľnosti: neurčujeme napríklad presné poradie vykonávania inštrukcií jednotlivými agentami a nechávame ho na OS alebo VM. Skúsenosti hovoria, že za ustálených vonkajších podmienok má systém tendenciu utriasť svoju činnosť na periodické vykonávanie určitej postupnosti inštrukcií, zatiaľ čo za meniacich sa vonkajších podmienkach sú tieto zmeny premietnuté do neustále sa meniaceho poradia. Na MAS vyjadrený v agent-space sa teda dá pozrieť aj ako na monolytický systém, ktorý mení svoj algoritmus v závislosti od stavu vonkajšieho prostredia. Dobrým indikátorom týchto zmien je globálna aktivita systému, ktorú možno rozumne definovať ako počet zápisov do prostredia za určitú časovú jednotku. Obrázok č. 40 znázorňuje, že dynamika vonkajšieho prostredia sa výrazne podpisuje na iregularite dejov vo vnútri systému.



Obrázok č. 40 Vplyv prostredia na irregularitu globálnej aktivity systému

Hore je zmeraná aktivita na simulátore systému UDCS/QNET v ktorom je vonkajšie prostredie simulované periodickým dejom. Dole je zmeraná aktivita na reálnej inštalácii tohto systému bežiaca v reálnom prostredí

Iregularitu systému môžeme zvýšiť používaním generátora pseudo-náhodných čísel a v niektorých prípadoch je to žiadúce. Pritom náhodu môžeme použiť nielen pri vetvení, ale obzvlášť rafinovaný spôsob je premietnutie náhody do fázového posunu frekvencií časovača jednotlivých agentov, alebo dokonca do samotnej frekvencie. (Dobrým príkladom použitia tejto stratégie je práve UDCS/QNET, viď nižšie).

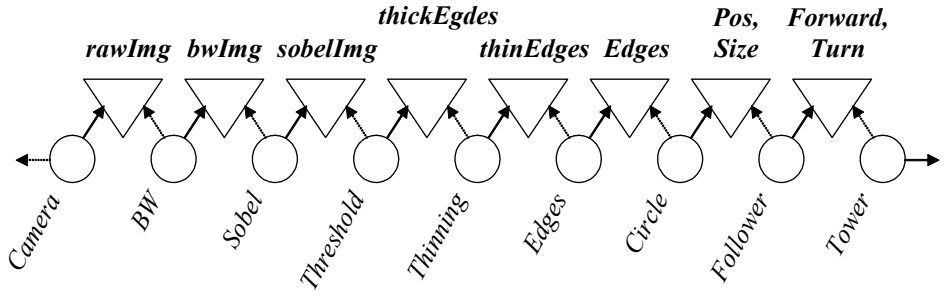
Zálohovanie: Príroda nám poskytuje niekoľko ukážok zálohovania, ako sú napríklad párové orgány (obličky, pľúca, vaječníky). Navyše niekedy sa vyskytujú prípady, keď počas evolúcie preberá jeden orgán činnosť druhého: napríklad slezina tvorí červené krvinky počas embryonálneho štádia a neskôr túto funkciu preberá kostná dreň, z čoho možno usudzovať, že slezina je evolučne starší krvotvorný orgán. Ešte vypuklejšie to pravdepodobne bude na molekulárnej úrovni, kde je zálohovaný každý gén s výnimkou tých, ktoré sú na chromozóme XY. Pri troche odvahy by sme sa na základe toho mohli domnievať, že živé systémy sa budú dobre modelovať takými organizačnými štruktúrami, ktoré umožňujú ľahký prechod od bežného riešenia k zálohovanému, od sériového k paralelnému. Agent-Space podporuje tento prechod lepšie, než ktorákoľvek iná, nám známa architektúra. Je to vďaka tomu, že bez akýkoľvek úprav môžeme pre určitý blok v systéme pridávať ako jeho čitateľov, tak aj zapisovateľov. Túto schopnosť môžeme vopred posilniť, keď nebudeme mená blokov s ktorými agent manipuluje v kóde fixovať, ale budeme ich brať ako parametre.

Príklad č. 10 [Lúčny 2004d][23]

Robot „PingPong“, ktorého riadiaci systém bol vyvinutý na báze agent-space na platforme Java, je vybavený dvomi servomotormi zabezpečujúcimi pohyb dopredu a dozadu ako aj otáčanie sa na mieste a kamerou, ktorá sníma scénu v smere jeho pohybu. Robot má za úlohu sledovať pingpongovú loptičku (Obrázok č. 41). Podvozok so servomotormi ako aj jeho bezdrôtové spojenie na vežu pripojenú k PC sú realizované na báze stavebnice Boe-bot od Paralax. Ako kamera je použitá bezdrôtová XCam2 od X10 skvelá hlavne nízkou spotrebou. Tieto dve zariadenia sú na PC integrované do jednotného riadiaceho systému na platforme Java: veža cez RS232 a javax.comm, kamera cez USB a JMF. Pod JVM sme implementovali agent-space a v nej sme vytvorili riadiaci systém ako kolekciu reaktívnych agentov.

Obrázok č. 41 Robot PingPong

Jadrom tohto riadiaceho systému je obyčajný „pipeline“. Farebný obraz prijímaný z bezdrôtovej kamery (rawImg) je skonvertovaný na čiernobiely (BwImg), na ten je aplikovaný Sobelov operátor, čo zvýrazní hrany (sobelImg). Potom na základe určitého prahu začiernime v obraze všetko, čo netvorí hrany (thickEdges), tie premeníme odbúravaním na čiary (thinEdges) a z takto upraveného obrazu vyberieme zoznam relevantných úsečiek (Edges). Z týchto sa snažíme zistiť bod, ktorý by mohol byť stredom loptičky (robíme priesečník kolmíc na ich stred) a ten otestujeme, či sa okolo neho nachádza významná časť kružnice. Ak niečo nájdeme, vieme na obraze určiť polohu stredu loptičky ako aj jej veľkosť. Ak je loptička príliš vpravo vydávame príkaz na otáčanie sa doprava, ak príliš vľavo, doľava. Ak je loptička príliš veľká, cúvame, ak príliš malá, ideme dopredu. No a na konci spracovania je vysielanie týchto príkazov do podvozku robota. Tento obyčajný „pipeline“ je však realizovaný neobyčajným spôsobom. Každú zložku postupného spracovania vykonáva jeden agent, pričom medzivýsledky spracovania sú bloky v prostredí (Obrázok č. 42).

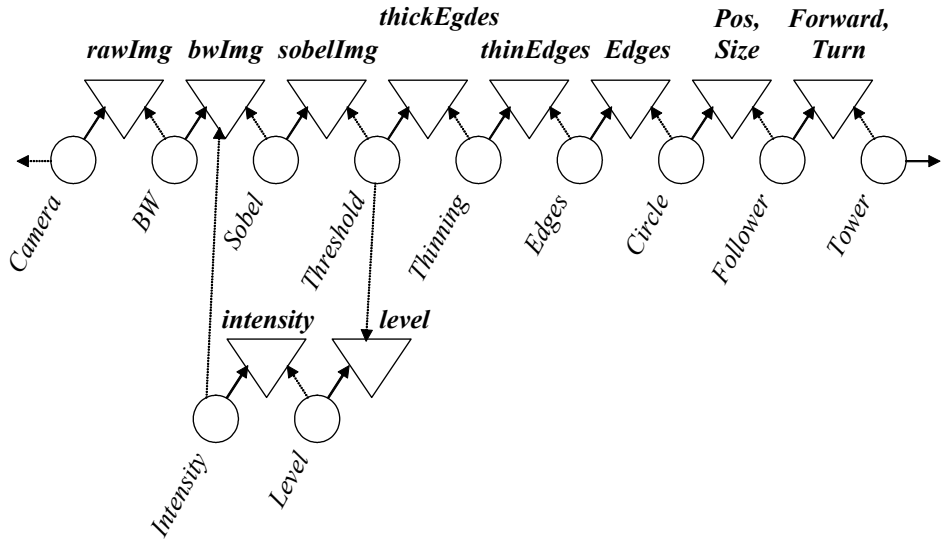


Obrázok č. 42 Riadiaci systém robota PingPong

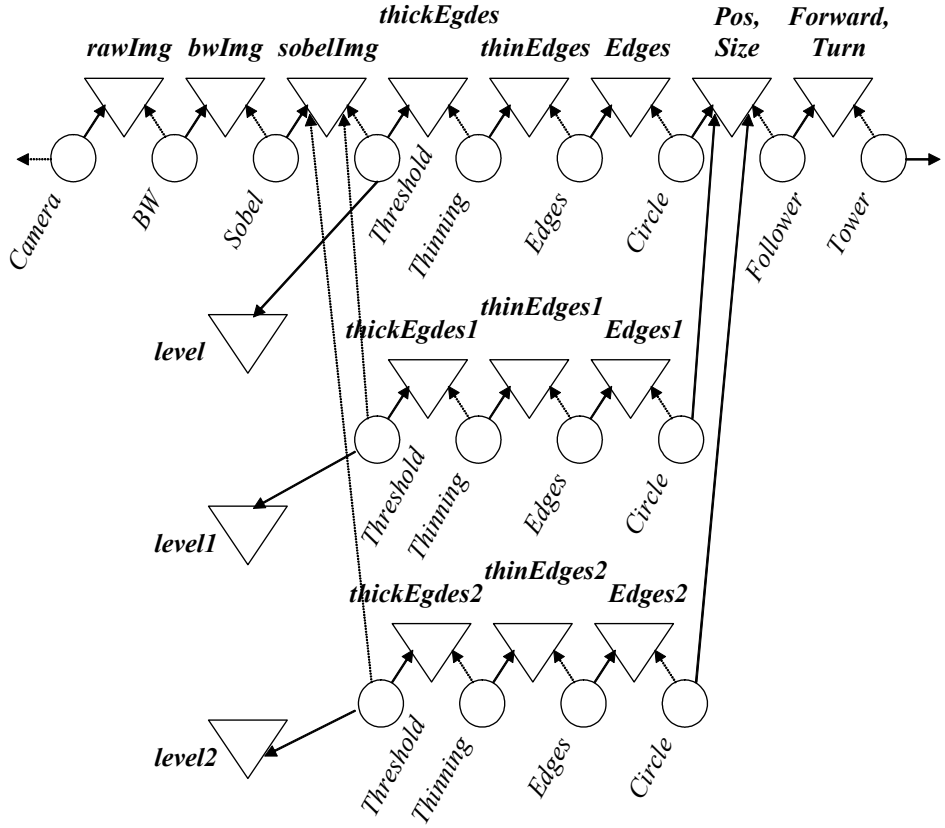
V tejto fáze vývoja agent-space prináša iba logické odčlenenie jednotlivých fáz spracovania (to by ale poskytla ľubovoľná forma modularity), možnosť pridávať a uberať agenty, ktoré medzivýsledky zobrazujú (tu je určitá výhoda, lebo pri zapnutí a vypnutí zobrazenia nemusíme meniť kód agenta, ktorý medzivýsledok počíta a tento kód vôbec nebude obsahovať podporu tohto zobrazenia vďaka čomu bude prehľadnejší) a ešte jeden zaujímavý efekt spôsobený implicitným vzorkovaním: agenty nemusia pracovať s rovnakou frekvenciou. Hoci by sme sa snažili, aby celý proces spracovania prebiehal sekvenčne a každý agent budili triggerom na výsledok činnosti predchádzajúceho agenta, rozpoznávanie loptičky (agent Circle) je časovo náročné a nebude stíhať so snímaním obrazu čo má 15 Hz. Systému to však vôbec nebude vadiť. Každý blok sa bude meniť s maximálnou frekvenciou, ktorú bude stíhať: rawImg bude mať svojich 15 Hz, Edges už len 8 Hz a Pos 2Hz. Tým pádom Tower, ktorý reálne stíha vysielať príkazy 4Hz (kamera používa WIFI a podvozok rádiomodem, preto ten rozdiel), čiže jeho kapacita bude vlastne nevyužitá (ale k dispozícii, keby bloky Forward a Turn raz zapisoval niekto iný).

Takto zimplementovaný robot sleduje loptičku za optimálnych podmienok celkom uspokojivo. Čoskoro však zistíme, že kvalita sledovania závisí od svetelných podmienok: že guličku robot nevidí keď je šero ani keď je príliš silné svetlo. Kľúčom k tomu javu je zvolená hodnota prahu v agentovi Threshold – v šere zoberieme za hrany príliš málo a pri silnom svetle príliš veľa. Agent-space nám v tejto chvíli umožní poľahky systém modifikovať. Môžeme napríklad upraviť agent Threshold, aby bral prahovú hodnotu z prostredia. Teda ju bude pri každom použití z prostredia čítať napriek tomu, že sa mení len tak často, ako sa menia svetelné podmienky (tu meníme agent z reaktívneho na čisto reaktívny). Potom môžeme hodnotu toho

nového bloku odvodzovať z priemernej intenzity obrazu. Na to pridáme agenty Intensity a Level (Obrázok č. 43).

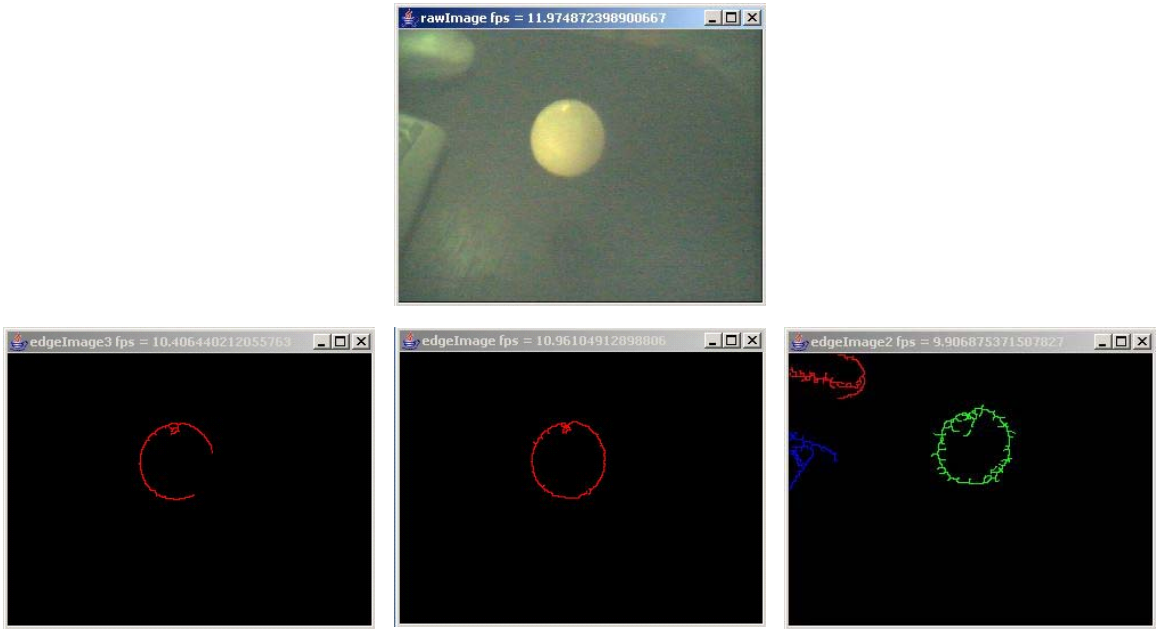


Obrázok č. 43 Modifikácia riadiaceho systém robota PingPong na princípe čistej reaktivity



Obrázok č. 44 Modifikácia riadiaceho systém robota PingPong na princípe zálohovania

Avšak nájsť tento vzťah medzi priemernou intenzitou a optimálnou prahovou hodnotu nie je priamočiare: v oblasti kde je loptička môže byť šero, zatiaľ čo inde je jasno a je zle. A tu prichádza ku slovu využitie zálohovania. Ľahko zariadime, že sa bude súčasne skúšať viac prahových hodnôt a výsledok tejto činnosti sa bude akumulovať. Vďaka agent-space na to nemusíme prakticky nič programovať, stačí agenty inak naštartovať (ak sme názvy blokov, nad ktorými agenty pracujú už zaviedli ako parametre) (Obrázok č. 44). Robot sa následne pozerá na svet troma očami (Obrázok č. 45).



Obrázok č. 45 Vnímanie pingpongovej loptičky robotom PingPong

Hore obraz z kamery, dolu zdetekované hrany objektov pri vysokej, strednej a nízkej prahovej hodnote

Pozoruhodné na tomto riešení je to, že výsledky každej vetvy sa úplne jednoducho zapisujú na to isté miesto (Pos a Size), nepridávame žiadny modul, ktorý by z viacerých hodnôt vypočítaných v jednotlivých vetvách urobil výslednú. Jedna hodnota proste prepisuje druhú, pričom víťazia tie, ktoré si na ďalšie spracovanie stihne agent Follower prevziať. Môže to samozrejme fungovať len vďaka tomu, že ak agent Circle žiadnu loptičku nenájde, nič do blokov Pos a Size nezapíše. To je ale v agent-space všeobecná stratégia – mnohoročná skúsenosť nám hovorí, že zapísať do prostredia niečo ako BAD_VALUE je vždy chyba. Keď potrebujeme hodnotu v prostredí rušiť, treba to urobiť tak, že ju zapíšeme iba s istou časovou platnosťou, takže ak ju neobčerstvíme novým zápisom, sama sa z prostredia vytratí.

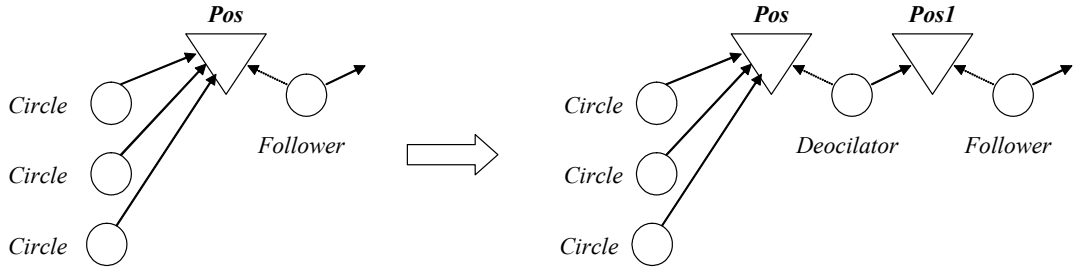
Druhá otázka je, čo by sa dialo, keby dve rôzne línie našli loptičku na inom mieste. Pokiaľ by našli reálne tú istú loptičku len by tam bol drobný rozdiel vo výpočte polohy, nedialo by sa nič vážne. Iná situácia by nastala, keby šlo o dve rôzne loptičky, napríklad by bola jedna vľavo a jedna vpravo. Vtedy by boli príkazy podvozku nekonzistentné. Presne to isté sa koniec koncov môže stať už pôvodnému riešeniu bez zálohovania. Samozrejme, dá sa to ošetriť. Dostávame sa tým k ďalšej biologicky relevantnej vlastnosti.

Oscilácia a deoscilácia: Sú dobre známe experimenty – ako je Neckerova kocka – ktoré jednak svedčia o tom, že v mysli často zápasia viaceré protichodné názory, jednak o tom, že je tam prítomný mechanizmus, ktorý sa vie prikloniť k jednému z nich. Oba mechanizmy sa

typicky vyskytujú v riešeniach podľa agent-space: prvé vďaka zápisu viacerých agentov do rovnakého bloku, druhé vďaka časovej platnosti a časovo orientovanému spracovaniu údajov.

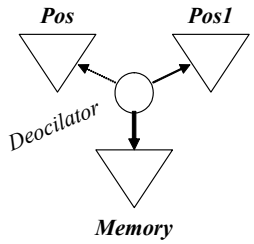
Príklad č. 11

Vráťme sa k predchádzajúcemu príkladu a uvažujme čo sa deje keď do obrazu umiestnime dve loptičky. Môže sa pritom stať, že jedna rozpoznávacia línia nájde jednu a druhá druhú loptičku. Tým pádom sa hodnota bloku ***Pos*** mení rýchlejšie ako aprílové počasie. Keby sme nechali teda agent *Follower* konať podľa ***Pos***, robot by sa mykal spôsobom na hony vzdialenom od inteligencie. Pomerne ľahko to však môžeme zachrániť vložením agenta-deoscilátora, ktorý sa prikloní k jednej z možností a tú dáva do prostredia ako ***Pos1***. Ak sa hodnota v ***Pos*** iba málo líši od toho čo si pamätá, považuje to za dôsledok vlastného pohybu robota alebo pohybu loptičky. Preto ju berie za novú hodnotu ***Pos1*** a zapamätá si ju. Ak sa hodnota v ***Pos*** líši príliš, ignoruje ju. Až keď už dlhší čas nedostal žiadne potvrdenie, že zapamätaná loptička stále v scéne je, prikloní sa k inej ktorá sa ponúka (Obrázok č. 46).



Obrázok č. 46 Oscilácia a deoscilácia

Na deoscilátore je zaujímavé, že sa ľahšie implementuje ako čisto reaktívny agent, ktorý si pamätá svoju voľbu v prostredí, teda v bloku – nazvime ho ***Memory*** (Obrázok č. 47). Túto voľbu číta pri každom čítaní ***Pos*** a zapisuje pri každom zápise ***Pos1*** s určitou časovou platnosťou. Ak sa mu potom dlhší čas neponúkne v Pos príbuzná hodnota, ktorú by zapísal do ***Pos1***, platnosť voľby v bloku ***Memory*** vyprší. Tým pádom deoscilátor nič z ***Memory*** neprečíta a bude spokojný s prvou ***Pos***, ktorú dostane.



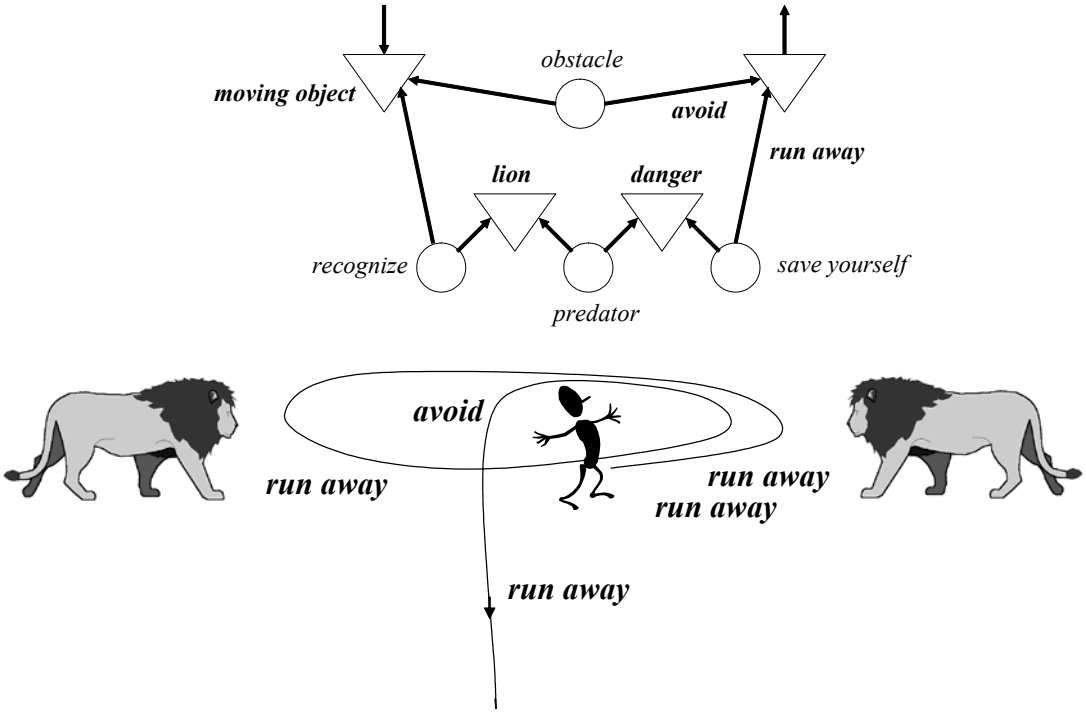
Obrázok č. 47 Deoscilátor na báze čistej reaktivity

Blok ***Memory*** sa potom dá použiť na to, aby sme prinútili systém zamerať sa na inú loptičku – stačí ho v príhodnej chvíli vymazať pomocou *delete*. Iný zaujímavý spôsob ako to dosiahnuť, je zastaviť vstup do systému, napríklad zablokovať blok sobelImg zavolaním služby *write* s prioritou na čas za ktorý vyprší platnosť bloku ***Memory***. (Komu sa zdá tento spôsob neprirodzený, nech sa pozrie na Neckerovu kocku, počká kým ju uvidí zhora, a potom nech si skúsi vyvolať jej obraz zdola a nech si pritom zmeria, koľko mu to bude trvať.)

Fluidita: Pod týmto pojmom (ktorý si požičiavame od Douglasa Hofstadtera) máme na mysli schopnosť systému nielen vnútorne oscilovať medzi viacerými možnosťami, ale využiť túto osciláciu na prekonanie určitého uviaznutia. (Pritom správanie systému možno prirovnať k nezmáčanlivej kvapaline - ako je ortuť, ktorá sa snaží tiecť až nájde miesto kde sa dá vytiecť – od toho ten pojem fluidity - tekutosti.) Tento jav nastáva, keď je významný rozdiel vo frekvenciách s akými sa do určitého bloku zapisujú hodnoty od rôznych producentov. Hodnota zapísaná s malou frekvenciou má malú šancu prejaviť sa v systéme, ale pri troche šťastia môže natoľko zmeniť globálny stav systému, že nadobro umlčí svojich silnejších konkurentov.

Príklad č. 12

Uvedieme si primitívny príklad (verím, že existujú aj zložitejšie a zaujímavejšie). Predstavme si systém, v ktorom o smer jeho pohybu zápasia dve skupiny agentov. Jedna zabezpečuje vyhýbanie sa prekážke odbočením vľavo, druhá únik pred levom urobením čelom vzad. Prvá je aktívna, keď máme pred sebou prekážku, druhá keď máme pred sebou leva. Pokiaľ nie sú aktívne, nezapisujú ako povel na otočenie nič. Pokiaľ aktívne sú, zapisujú každá svoj povel – prvá „doľava“, druhá „čelom vzad“ do toho istého bloku (ktorý je už povedzme priamo previazaný s aktuátorom). Prvá však s oveľa menšou frekvenciou než druhá (Obrázok č. 48 hore). Takže systém ide normálne rovno, keď narazí na prekážku, odbočí doľava a keď narazí na leva, urobí čelom vzad. Pritom je nenulová, ale dosť nízka šanca, že pri levovi odbočí doľava, lebo hoci lev je tiež prekážka, reakcia na leva je väčšinou rýchlejšia, než reakcia na ľubovoľnú prekážku.



Obrázok č. 48 Ukážka fluidity

Čo sa teraz bude diať, keď sa systém ocitne medzi dvomi levmi? Keď urobí čelom vzad pred prvým, ocitne sa tvárou k druhému a tým pádom v rovnakej situácii ako predtým. Mechanizmus, čo ho má chrániť pre levom, ho len privádza k druhému levovi a tak pobehuje medzi nimi. S trochou šťastia (a ak na to bude mať dostatok času) sa v ňom však presadí

mechanizmus, ktorý ho normálne chráni pred nárazom do prekážky a tak systém odbočí doľava. Tým pádom sa vymaní z uviaznutia medzi levmi a už sa doňho nevráti. Organizácia jeho vnútorných mechanizmov založená na fluidite, ktorá mu trochu znižuje úspešnosť úteku pre jedným levom, ho zachráni v prípade, že sú levy dva. Ide tu o znižovanie dokonalosti mechanizmu (ale nie zase nejaké dramatické, povedzme zo 100% na 98%) za účelom dosiahnutia jeho aplikovateľnosti za širších podmienok, ktoré nemusia byť dopredu stanovené. Fluidita je teda prostriedok na zvýšenie aplikovateľnosti za cenu zníženia úspešnosti. Pritom toto zníženie sa dá istým spôsobom ovládať. Keď systém zdetekuje, že uviazol, môže zvýšiť frekvencie všetkých svojich časovačov. Frekvencie výstupov, ktoré majú fyzikálnu povahu, ostanú samozrejme rovnaké, a teda pravdepodobnosť, že si fluidita nájde cestu, stúpne toľko krát, koľko krát zrýchlime časovanie. V prírode je také niečo možné samozrejme len za cenu väčšej spotreby energie, ale snaha vyburcovať systém a zavolať si tak fluiditu na pomoc sa tu pravdepodobne objavuje viac než logicky správne postupy. Napríklad chrobák prevrátený na chrbát sa nesnaží pomaly húpať zľava doprava a sprava doľava a postupne sa tak rozkolísať (čo je z hľadiska fyziky asi najlepší spôsob), ale urputne myká nohami čo to dá, aby premenil svoj rovnovážny stav na chaotický, v rámci ktorého sa možno nájde cesta do iného – želaného – rovnovážneho stavu.

Brikoláž a emergencia: Nevýhodou centrálneho riadenia je prílišná determinovanosť podmienok za ktorých dokáže rozumne operovať. Pracuje absolútne spoľahlivo keď podmienky zodpovedajú predpokladom, avšak pokiaľ sa ukáže, že sa v predpokladoch na niečo zabudlo, spravidla v systéme nie je nič, čo by bolo schopné vzniknutú situáciu ošetriť. Naopak, pokiaľ necháme obdobné riadenie vzniknúť z interakcie viacerých agentov, výsledok spravidla poskytuje rozumné správanie systému aj v situáciách, s ktorými sa pri návrhu nepočítalo. Využíva sa tu fakt, že z pomerne jednoduchých zákonov sa dajú spravidla odvodiť zložité, ďalekosiahle a prekvapivé závery, zatiaľ čo keď miesto týchto zákonov používame iba ich momentálne známe dôsledky, náš model je spravidla neúplný. Posun od centralizovaného k decentralizovanému zodpovedá snahe generovať správanie počítačového systému na rovnakom princípe ako je správanie reálneho sveta generované z jeho zákonov. Podobenstvom je napríklad modelovanie prúdenia kvapaliny: môžeme to urobiť na základe modelu ideálnej kvapaliny (Bernoulliho rovnica) – čo zodpovedá centrálnemu (globálnemu) prístupu, alebo pomocou definovania pravidiel pohybu pre jednu časticu kvapaliny – čo zodpovedá decentralizovanému prístupu. Keby sme vo fáze návrhu nemali poňatia o tom, že existuje turbulencia, oba prístupy by boli rovnako účinné. Z toho prvého však turbulenciu nikdy nedostaneme, zatiaľ čo pri druhom sme ju nevedomo implicitne zaviedli.

Výhodou centrálnych systémov sú samozrejme ich nesporne menšie nároky na výkon počítača. Nie vždy je však pomer potrebných výkonov tak obludný ako v uvedenom podobenstve s prúdením kvapaliny. Ide o prípady, kedy sa globálny systém skladá z pomerne malého počtu lokálnych jednotiek.

Keď sa decentralizovanému systému vystavenému podmienkam, z ktorými sa nepočítalo, podarí túto situáciu zvládnuť, možno pritom pozorovať jednu z dvoch nasledujúcich možností. Prvou možnosťou je, že nejaký konkrétny mechanizmus vedome vložený do systému za konkrétnym účelom, začal pracovať pre iný účel, na ktorý nebol stvorený, ale na ktorý zhodou okolností funguje – to nazývame brikoláž. Druhou možnosťou je, že sa takýto mechanizmus objaviť nepodarí, lebo kľúčovú rolu zohráva interakcia viacerých mechanizmov – to nazývame emergencia. Možno teda povedať, že brikoláž je triviálnym prípadom emergencie (emergencia, kde sa na „interakcii“ podieľa jediný mechanizmus).

Príklad č. 13 [Lúčny 2004a]

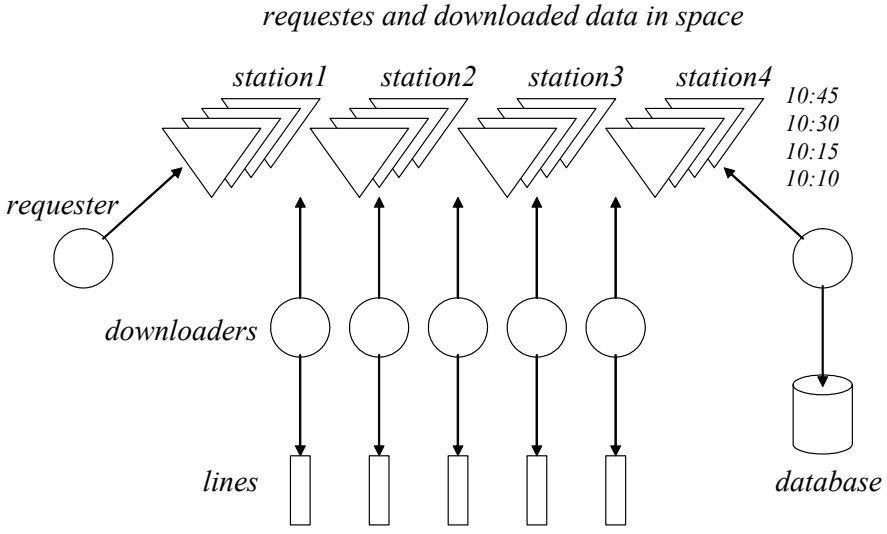
UDCS/QNET je komerčný systém na zber údajov so siete automatických hydrometeorologických staníc. Tieto stanice vykonávajú pravidelné merania s určitou periódou (napríklad 15 minút). Úlohou systému je zozbierať tieto údaje kompletne a čo najskôr: potrebné sú aktuálne aj historické dáta. Spojenie je realizované pomocou PSTN a GSM siete. Stanice sú spravidla vybavené GSM modemom, zatiaľ čo centrálny systém ich obvoláva cez PSTN (dial-up) linky. Staníc je rádovo 100 a liniek na centrálnom systéme rádovo 10.

Táto triviálna úloha sa stáva zaujímavou keď zvážime všetky fyzikálne faktory, ktoré robia zber nespoľahlivým:

- GSM sieť môže byť dočasne preťažená, kvôli čomu nie je možné vytvoriť so stanicou spojenie. GSM sieť je poruchová a prenesené údaje môžu byť poškodené šumom. Niektoré retranslačné stanice GSM siete môžu byť istý čas mimo prevádzky (napríklad sú v púšti a po vyčerpaní paliva v agregáte sa musí čakať na jeho doplnenie, ktorému zase môže brániť piesočná búrka)
- Stanice sú napájané zo solárnych kolektorov, takže nie je zaručené, že majú dostatok energie, aby vytvorili GSM spojenie, prípadne ani na to, aby urobili meranie (napríklad je solárny panel zanesený pieskom, alebo je opotrebovaný akumulátor, ktorý stanicu napája počas noci)
- Prenosová kapacita systému je obmedzená. Jedno obvolanie stanice trvá istý čas. V prípade, že sa so stanicou dá spojiť je to okolo 40 sekúnd, v prípade, že je nedostupná, je to okolo 80 sekúnd.

Je tu teda niekoľko protichodných faktorov: keď chceme získať určité dáta a nepodarí sa nám to, môže to byť tým, že sa momentálne nedá dovolať a dáta sa dajú získať neskôr, ale aj tým, že stanica nemeria a dáta sa získať vôbec nedajú.

Netvrdíme, že sa nedá napísať centrálny systém, ktorý by sa s týmto vysporiadal, ale ľahkosť s akou sa dá naprogramovať účinné decentralizované riešenie vyráža dych (Obrázok č. 49).



Obrázok č. 49 UDCS/QNET – decentralizovaný systém, na kt. môžno pozorovať brikoláž

Základným rysom systému je, že pre každé konkrétne údaje z určitého času a stanice je do prostredia generovaný jeden blok reprezentujúci túto požiadavku. Tieto bloky majú len určitú platnosť (napríklad tri dni), takže pokiaľ nie sú včas vybavené, sú zabudnuté. Nad týmito

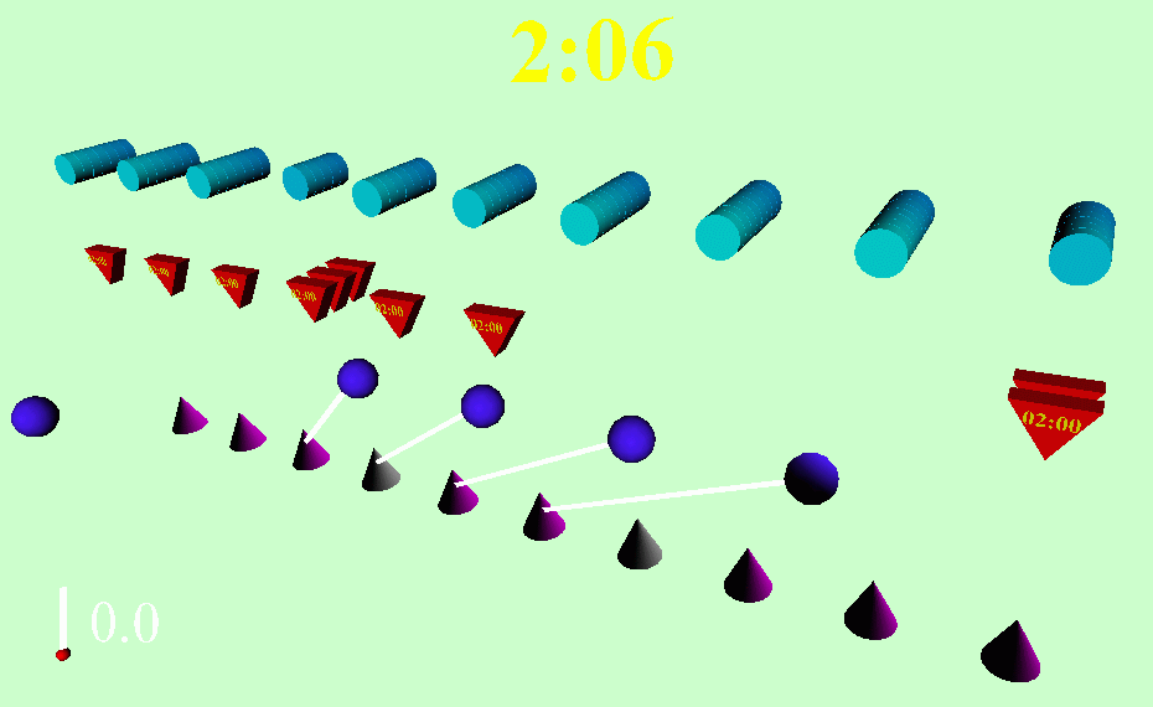
blokmi pracujú homogénne agenty ovládajúce jednotlivé sériové linky, ktoré sú schopné vykonať zavolanie stanice a stiahnutie dát. Každý takýto agent pri zobudení s pomerne malou periódou (napr. 30 sekúnd) začne prehľadávať požiadavky. Keď nejakú nájde, pozrie sa, či táto stanica už nie je vybavovaná (na to slúži blok používaný ako zámok, viď implementáciu agent-space nad SRR). Ak nie je, označí ju ako vybavovanú. Zistí si všetky požiadavky na túto stanicu a pokúsi sa o spojenie. Ak sa mu to podarí, pýta sa postupne stanice na každú požiadavku. Tie požiadavky, na ktoré dostane korektnú odpoveď (dáta alebo informáciu, že sa nenamerali) z prostredia vymaže a zároveň tam zapíše získané dáta, po ktorých snorí agent, ktorého úlohou je prevziať ich a zapísať do databázy. Keď sa už na všetko stanice opýta, zloží spojenie, označí stanicu ako nevybavovanú a ide spať. To urobí aj pri každej výnimke, ktorá sa vyskytne: napríklad pri prerušení spojenia. Pomôže ešte, ak každý agent po zobudení vykoná sleep na náhodný počet sekúnd, nie príliš veľa, ale dosť na to aby pristupovali k požiadavkám v náhodnom poradí - PSTN linky tak budú približne rovnomerne zaťažené. Ostáva vyriešiť, aby sa na stanicu, ktorá je nedostupná, nesnažili neustále volať všetky agenty: to sa dá manipuláciou s časovou platnosťou požiadaviek: ak sa ich nepodarí vybaviť a odmazať, posunie im agent začiatok platnosti o pauzu, ktorú chceme medzi dvomi pokusmi mať (túto musíme voliť s ohľadom na energetické možnosti stanice ako aj na celkovú kapacitu PSTN liniek). Ďalším detailom je riešenie toho, aby systém nepreferoval určitú stanicu. Na to je vhodné, aby agent preferoval staršie požiadavky pred novšími. Teraz stačí, aby sme jedným agentom generovali zodpovedajúce požiadavky. Všetko ostatné sa odvádza týchto jednoduchých správaní. Globálne správanie pritom zahrňuje mechanizmy:

- systém sťahuje dáta zo všetkých staníc
- ak nejaké dáta prídu zašumené, bude ich systém opäť žiadať
- ak bola nejaká stanica nedostupná, po obnovení spojenia z nej systém stiahne všetko, čo sa stiahnuť nepodarilo
- oneskorenie prítomnosti dát v databáze oproti ich prítomnosti na staniciach je minimálne (v rámci určitej malej tolerancie)

Teraz nastal čas prejsť k demonštrácii brikoláže a emergencie. Počas vývoja tohto systému sme predpokladali, že kapacita PSTN liniek, s ktorými systém operuje, je dostatočne veľká na to, aby sa všetky stanice obvolali počas každej periódy merania, ba je dokonca rádovo väčšia. Keď však k systému pridávame postupne ďalšie a ďalšie stanice, môže sa stať, že ani za optimálnych podmienok táto kapacita nestačí (stačí samozrejme na to, aby sme dáta vôbec niekedy stiahli, čo je dané tým, že čas na vybavenie dvoch požiadaviek pri jednom volaní je podstatne menší ako čas na dve volania vybavujúce jednu požiadavku – to je kvôli tomu že väčšinu času spotrebujeme na nadviazanie spojenia). Budeme však musieť systém nejako upraviť, aby sa s nepredpokladaným nedostatkom kapacity vysporiadal? Takmer s istotou možno tvrdiť, že centrálne riadený systém by sa v takomto prípade musel preprogramovať, hoci v drobnostiach. Jeho typická štruktúra by totiž mala vonkajší cyklus prechádzajúci cez časové periódy a tento cyklus by sa teraz začas oneskorovať voči skutočnému času. Predvedeného decentralizovaného riešenia sa však netreba ani dotknúť. Prejaví sa totiž brikoláž: mechanizmus, ktorý zaručoval v pôvodnom riešení, že dáta, ktoré sa na prvý pokus nepodarilo stiahnuť, sa stiahnu neskôr, zabezpečí, že sa neskôr stiahnu aj dáta, ktoré sme sa stiahnuť vôbec nepokúsili kvôli nedostatku kapacity. Jediné čo sa teda vo výslednom správaním zmení, bude oneskorenie prítomnosti dát v databáze, ktoré vystúpa nad jednu periódu merania. Inak všetky dáta prídu a dokonca oneskorenie bude rovnomerne rozložené a bude dosahovať minimálnu potrebnú hodnotu. Takto úhľadne napísané sa to môže zdať ako trivialita, ale v MicroStep-MIS, kde sme boli reálne vystavení tejto situácii, sme si o tom urobili poradu, ktorej prekvapivým výsledkom bolo, že netreba urobiť vôbec nič.

Ďalšia zaujímavá situácia vznikla, keď sa ukázalo, že PSTN linky, ktoré systém používa, nie sú spoľahlivé, ako sa predpokladalo. Môže tu dôjsť jednak k poruche dial-up

modemu, ale hlavne k neúmyselnému fyzickému prerušeniu linky, či už v ústredni alebo v mieste systému. Je teda možné, že sa systém snaží volať z linky odkiaľ sa nikam nedá dovolať a keby sme použili inú, tak sa to podarí. Ako sa systém vysporiada s týmto nepredpokladaným stavom? Ukazuje sa, že ak nejaká PSTN linka vypadne (a agent cez ňu volajúci sa stane neúčinným), jej funkciu prevezmú ostatné linky (agenty). Táto linka samozrejme spôsobí oneskorenie prítomnosti dát v databáze, ale pravdepodobnosť toho, že bude väčšie než jedna perióda merania, je malá (pravdepodobnosť, že sa dáta oneskoria o k periód bude presne n^{-k}, kde n je počet liniek). Príčinou implicitnej implementácie tohto správania bola naša explicitná snaha rozložiť používanie liniek rovnomerne (čo má vzhľadom na predplatené minúty nielen estetický, ale aj ekonomický prínos), sama o sebe by však nebola stačila a preto tu skôr môžeme hovoriť o emergencii než o brikoláži, lebo ťažíme z početnosti agentov. Kým príklad s brikolážou by sa dal použiť aj keby sme mali iba jeden agent na jednej linke, toto funguje iba vtedy, keď máme aspoň dvoch.



Obrázok č. 50 Ukážka zo simulátora systému UDCS/QNET vytvorená vo VRML

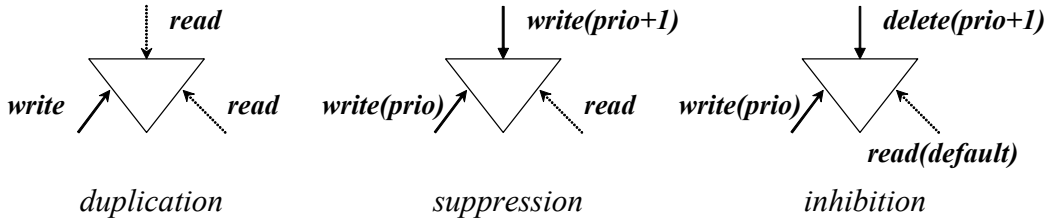
Modré guličky predstavujú agenty, červené trojuholníky požiadavky, tyrkízové peniažky stiahnuté dáta, kužele stanice (fialové dostupné, čierne nedostupné), biele úsečky momentálne spojenia. Posunovač (vľavo dole) umožňuje nastaviť poruchovosť siete.

Vzťah k subsumpčnej architektúre

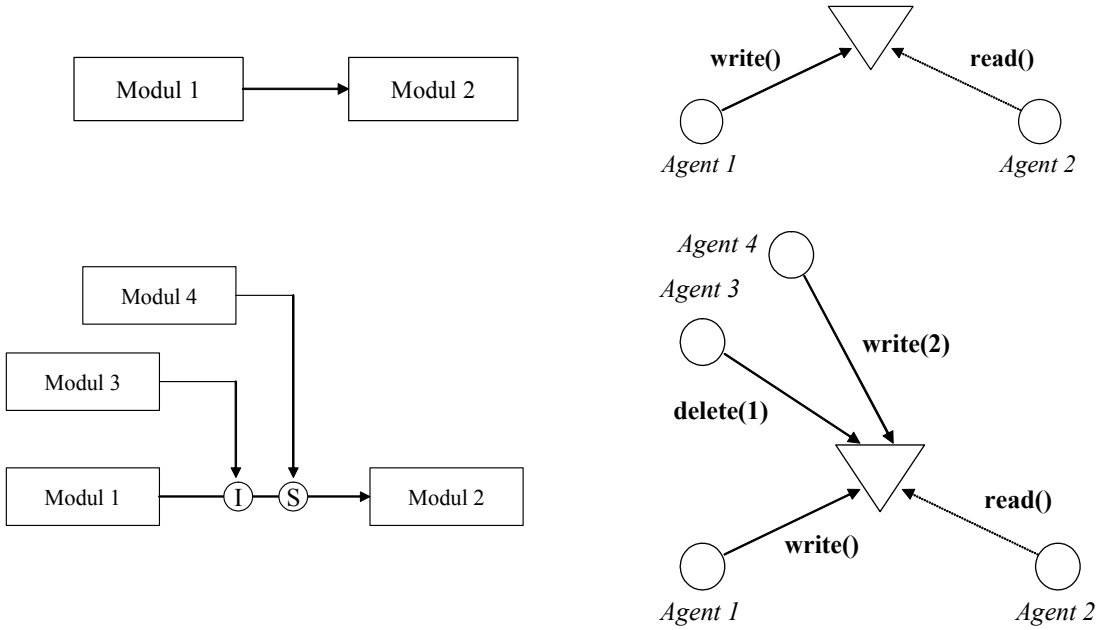
Zo všetkého, čo bolo kedy vymyslené, sa agent-space najviac ponáša na subsumpčnú architektúru. Pôsobí to na prvý pohľad prekvapujúco, nakoľko subsumpčná architektúra je všeobecne považovaná za architektúru na tvorbu jedného agenta a agent-space je nástroj na tvorbu multiagentového systému. Táto nesymetria je však iba zdanlivá, pretože agent-space vznikla z iniciatívy poňať agent ako multiagentový systém, jednouzlové ako distribuované a pod. Subsumpčná architektúra bola pre agent-space od začiatku vzorom a do značnej miery

ju agent-space kopíruje, či skôr sa ju snaží prefomulovať pomocou nových výrazových prostriedkov. Aký je teda vzťah medzi vzorom a výsledkom? Pokiaľ skúmame povrchové rozdiely, najvýraznejší je v aplikačnej oblasti. Subsumpčná architektúra je v podstate zviazaná výlučne s mobilnou robotikou, zatiaľ čo agent-space chce byť univerzálny programovací prostriedok pre všetky interaktívne systémy. V tomto zmysle je agent-space jednoznačne univerzálnejšia. Je však univerzálnejšia aj na úrovni svojich vyjadrovacích schopností?

Zhruba platí, že keď máme nejaké riešenie vyjadrené v subsumpčnej architektúre, dokážeme ho priamočiaro prepísať do agent-space. Moduly pritom prevádzame na agenty a komunikačné spojenia na bloky v prostredí. Časovanie modulov premieňame na časovanie agentov, vnútornú pamäť v moduloch na vnútorný stav agentov alebo ďalšie bloky. Mechanizmy subsumpcie nahradzujeme pristupovaním viacerých agentov k jednému bloku. Odpočúvanie sa mení na čítanie bloku, supresia na prepísanie bloku, inhibícia na vymazanie bloku (Obrázok č. 51 a Obrázok č. 52).



Obrázok č. 51 Realizácia mechanizmov subsumpcie v agent-space

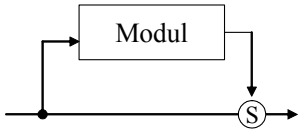


Obrázok č. 52 Transformácia subsumpčnej architektúry na agent-space

Hore: moduly sa transformujú ako agenty, komunikačné vedenie ako bloky. Dolu: supresory a inhibítory sa transformujú pomocou rozšírenia agent-space o priority (ktoré sú uvádzané ako argument týchto služieb)

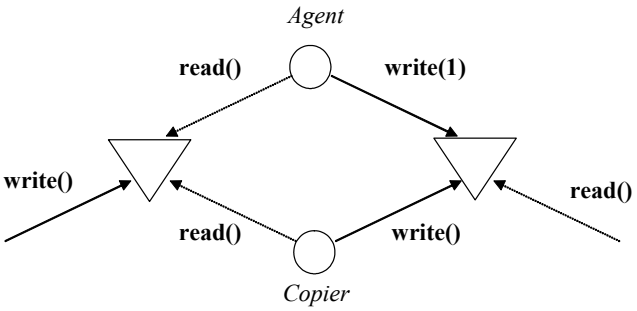
Vyskytujú sa však aj ťažkosti. V prvom rade vyplývajú zo striktnej priority vtierania vyšších vrstiev do nižších, ktoré sú v subsumpčnej architektúre dané poradím inhibítorov a supresorov na komunikačnom vedení. Základný koncept Agent-space nepozná žiadne priority, je to pre neho niečo cudzie. Priorita sa však dá doplniť ako nepovinný parameter pre služby write() a delete(). Túto prioritu si v bloku zapamätáme a to s ohľadom na časovú platnosť definovanú pri volaní služby (tu vidíme dôvod, prečo aj delete() potrebuje platnosť). Po túto dobu budeme brániť meniť obsah bloku všetkým agentom, ktoré k nemu pristupujú s menšou prioritou. Defaultnou prioritou, ktorá sa použije v prípade, keď prioritu pri volaní služby neuvedieme, bude nula. Priorita bude reálne číslo, aby sme aspoň teoreticky mohli vždy vložiť medzi dve tretiu. Hoci sme už v kapitole o implementácii uvádzali priority v prototypoch, dosiaľ sme ich prakticky nepoužili, ani sme sa nezaoberali ich implementáciou. Tá je našťastie triviálna – na základe zapamätanej priority niektoré write() a delete() na bloku nevykonáme.

Ďalší problém predstavujú štruktúry, ktoré aplikujú odpočúvanie a supresiu na tom istom komunikačnom vedení (Obrázok č. 53). V subsumpčnej architektúre je takáto štruktúra prípustná, hoci prakticky nepoužívaná.



Obrázok č. 53 Vedenie s odpočúvaním a supresorom v subsumpčnej architektúre

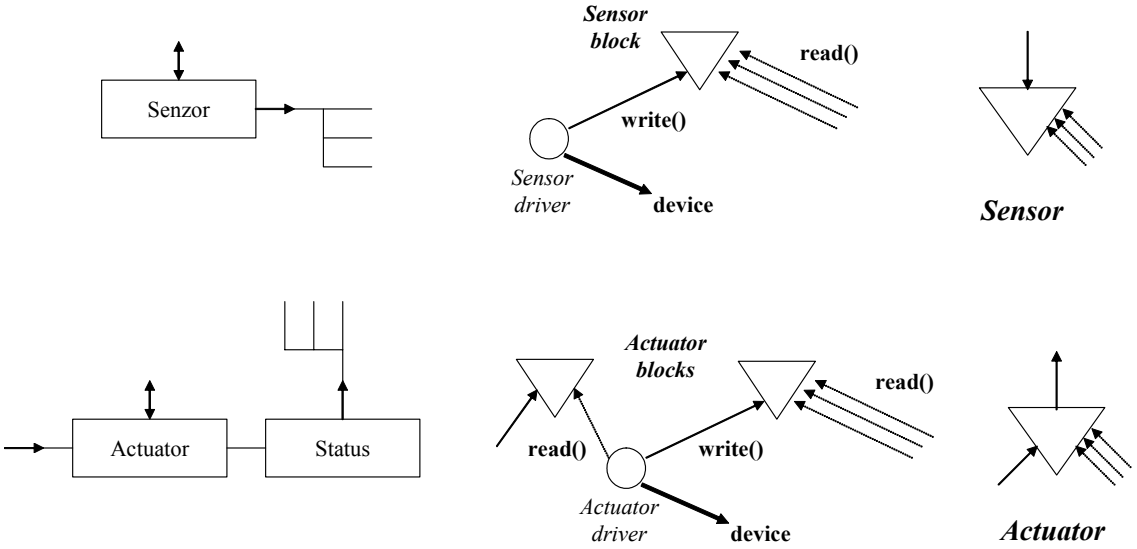
Pre agent-space je to problém, lebo tu by sme museli do jedného bloku zmestiť dve rôzne hodnoty. Nevieme to urobiť ináč, než zdvojením bloku a pridaním agenta, ktorého úlohou je kopírovať obsah jedného do druhého. Transformovať sa to teda dá, ale elegantné to nie je (Obrázok č. 54).



Obrázok č. 54 Transformácia vedenia s odpočúvaním a supresorom do agent-space

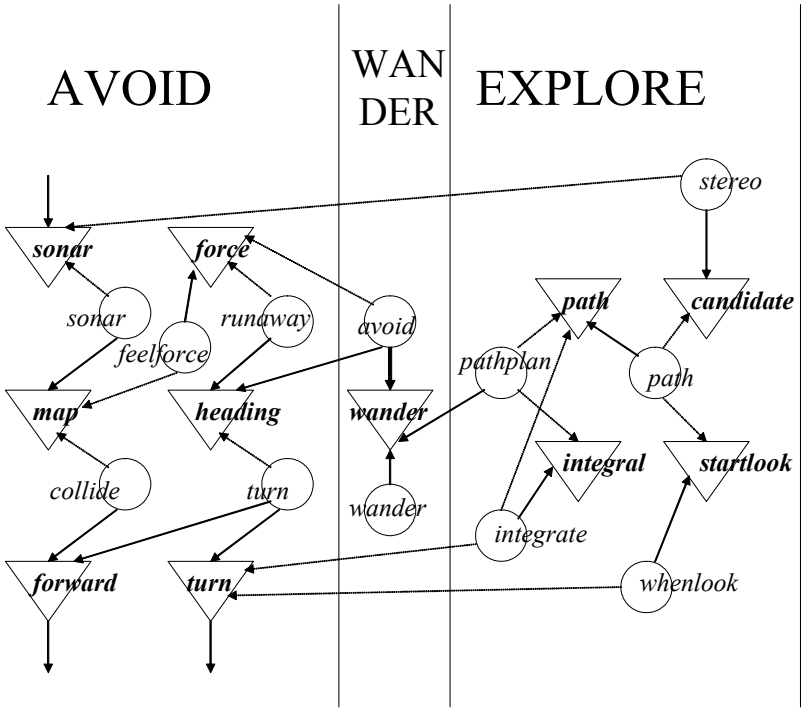
Posledný problematický bod predstavujú vstupy a výstupy. Je to spôsobené hlavne tým, že v tomto mieste kríva samotná subsumpčná architektúra. Táto predpokladá, že vstupy a výstupy budú realizované pomocou špeciálnych modulov. Tým pádom je však skutočný vstup alebo výstup nedostupný pre odpočúvanie inými agentami a to je niekedy potrebné. V riešeniach sa potom objavujú moduly, ktorých jedinou úlohou je poskytovať túto informáciu vyšším vrstvám čo je v rozpore so samotnou podstatou subsumpčnej architektúry. Tá má totiž odstrániť potrebu budovania takýchto rozhraní. Príkladom takýchto nepodarených modulov je modul Status v robotovi ALLEN (viď Obrázok č. 15). Pri transformácii do agent-space je potrebné takýto modul (na rozdiel od ostatných modulov) eliminovať na blok, ktorý zapisuje agent

zodpovedajúci modulu, ktorý ovláda príslušný vstup alebo výstup. Vstupy a výstupy budú teda po transformácii reprezentované dvojicou blok-agent, pričom z hľadiska ostatných agentov bude zaujímavý skôr blok, než agent (Obrázok č. 55).



Obrázok č. 55 Transformácia vstupov a výstupov zo subsumpčnej arch. do agent-space

Hore transformácia vstupu, dolu transformácia výstupu. Vľavo vyjadrenie v subsumpčnej architektúre, v strede analogické vyjadrenie v architektúre agent-space, vpravo jeho skrátený zápis (porovnaj Obrázok č. 24).



Obrázok č. 56 Reimplementácia robota ALLEN v agent-space

Príklad č. 14

Reimplementujeme teraz robota ALLEN (viď Obrázok č. 15) do agent-space (ako platformu sme použili VRML). Štruktúru, ktorá vznikne pri transformácii znázorňuje Obrázok č. 56. Popíšeme si, ako by sme túto štruktúru dostali postupne, uplatňujúc princípy sumbsumpčnej architektúry na štruktúry agent-space. Vyvíjať budeme teda inkrementálne, po vrstvách:

I. vrstva AVOID. Pre túto vrstvu požadujeme, aby robot nenarážal do prekážok.
   I.a. Ako prvé zimplementujeme aktuátor **forward**. Zavedieme tento blok a zvolíme štruktúru a sémantiku jeho dát dávajúce ostatným agentom možnosť zapísať jeden z príkazov: dopredu, dozadu, stop. Implementujeme agent – driver, ktorý povely v tomto bloku realizuje na motoroch robota. Defaultnou činnosťou, ktorú bude tento agent vykonávať, keď v bloku nebude žiadny príkaz, bude pohyb vpred. Už v tejto fáze dokážeme robot pustiť a pozorovať jeho správanie. Výsledkom bude, že robot pôjde rovno a potom bude narážať do prekážky.
   I.b. Implementujeme senzor **sonar** a údaje z neho usporiadame pomocou agenta *sonar* do mapy, ktorá pre každý výsek okolia robota po 30º určuje vzdialenosť najbližšej prekážky v danom smere. Smer sa pritom chápe relatívne (robot nemá kompas). Toto pole zapíšeme do bloku **map**.
   I.c. Pridáme agent *collide*, ktorý zastaví **forward** zapísaním príkazu vzad, keď zdetekuje v **map**, že hrozí zrážka. Robot teraz pôjde rovno a pred prekážkou cúvne, opäť kúsok pôjde dopredu, opäť cúvne, atď.
   I.d. Teraz nám pôjde o to, aby robot nielen nenarazil, ale prekážku obišiel. Pridáme aktuátor **turn**, ktorý dokáže vykonávať na motoroch povely doľava, doprava, rovno. Pritom tieto pohyby majú charakter plynulosti, nie je to príkaz typu „doľava o 5º“, ale „odteraz sa toč doľava až kým nepoviem rovno“. Defaultnou hodnotou je rovno.
   I.e. Pridáme agent *feelforce*, ktorý vyhodnotí smer, v ktorom najviac hrozí zrážka a zapíše do bloku **force**.
   I.f. Pridáme agent *runaway*, ktorý zavelí uberať sa opačným smerom, ak sa tento hrozivý smer nachádza v smeroch dopredného pohybu robota (napr. 270º - 90º). Tento povel zapíše do bloku **heading**.
   I.g. Zriadime agent *turn*, ktorý povel v **heading** realizuje tým, že zapíše stop do bloku **forward** a potom príslušný povel doľava alebo doprava na blok **turn**. Toto otáčanie má spätný vplyv na obsah map a tým pádom aj na heading, takže vzhľadom na časovanie agentov sa hodnota odklonu požadovaná v heading postupne zmenšuje až nakoniec úplne zruší. Robot sa teraz hýbe rovno, až príde ku prekážke, pri nej zatočí a potom sa pohybuje opäť rovno. To, či sa od prekážky odrazí alebo či bude sledovať jej tvar závisí od rozpätia smerov, ktoré považujeme za hrozivé. Vrstva AVOID je týmto pádom realizovaná. Za malú brikoláž môžeme považovať správanie robota, ktoré sa objaví, keď na neho vyštartujeme s nejakým objektom, ktorý sa pohybuje rýchlosťou menšou, než je rýchlosť robota. Vtedy sa mu robot vyhne. Toto vyhýbanie sa pohybujúcim objektom sme implicitne zimplementovali realizovaním mechanizmu na obchádzanie statických prekážok.

II. Vrstva WANDER. Robot teraz síce nenaráža do prekážok, ale má tendenciu pohybovať sa po cyklickej trase (najmä ak ho realizujeme v simulátore, kde naň nepôsobí šum prostredia). Aby sme to odstránili, pridávame túto vrstvu.
   II.a. Zavedieme agent *wander*, ktorý sa budí s pomerne veľkou periódou a s určitou pravdepodobnosťou navrhne, aby sa šlo určitým náhodne vygenerovaným smerom. Tento smer zapíše do rovnomenného bloku **wander**.

II.b. Pridáme agent *avoid*, ktorý si overí v bloku **force**, či nehrozí zrážka. Pokiaľ nehrozí, zapíše potom návrh agenta *wander* do bloku **heading**. Tým využíva mechanizmus na vyhýbanie sa prekážkam pre náhodné otočenie z vrstvy AVOID. Agent *turn* pochopiteľne považuje tento povel za povel od agenta *runaway*. Výsledkom je, že sa robot sem-tam náhodne otočí. Agent *avoid* ani nemusí použiť pri volaní write() vyššiu prioritu, ktorá by znemožnila agentovi *runaway* potlačiť jeho vplyv, lebo na základe odpočúvania bloku **force** je *avoid* aktívny iba vtedy, keď *runaway* nemá žiadny dôvod byť aktívny. To nám zároveň dáva istotu, že pridanie vrstvy WANDER nepokazí činnosť vrstvy AVOID, iba ju vhodne doplní.

III. Vrstva EXPLORE. V tejto chvíli je už pohyb robota zaujímavý a nepredikovateľný, ale robot má tendenciu zdržiavať sa na jednom mieste. Za nič na svete napríklad neprejde dverami. Preto pridávame túto vrstvu, ktorá by mala zariadiť, že sa z času na čas presunie na iné pôsobisko.

III.a. Pridáme teda agent *whenlook*, ktorý s príkazov, ktoré idú na efektor turn štatisticky usúdi, že robot už dostatočne preskúmal terajšie pôsobisko a nastal čas presunúť sa inam. Túto potrebu indikuje v bloku **startlook**.

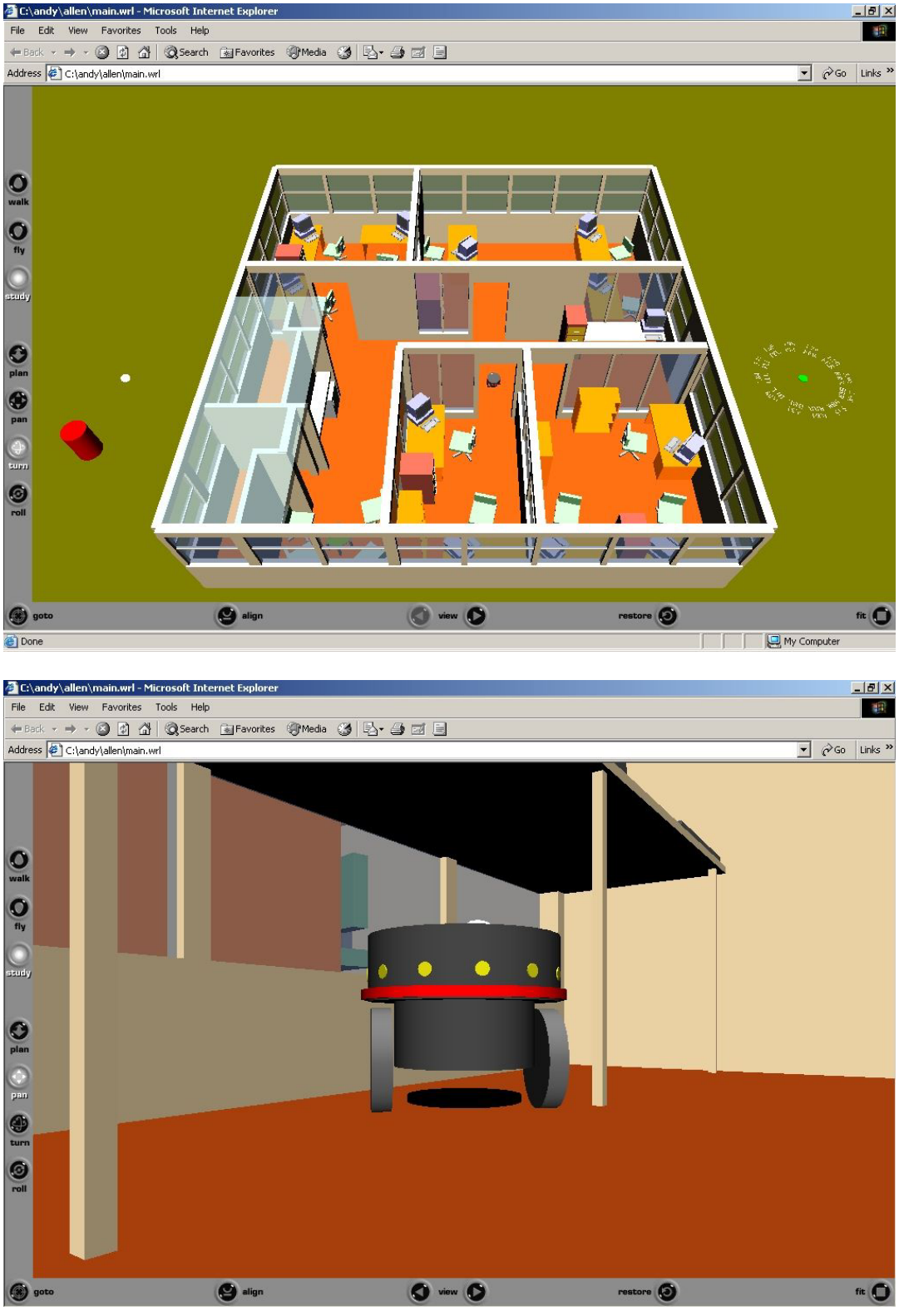
III.b. Zavedieme agent *stereo*, ktorý vyhľadáva z údajov zo sonarov taký smer, v ktorom je možný dlhý priamy pohyb, napríklad prechod do inej miestnosti. Tento smer zapíše do bloku **candidate**.

III.c. Pridáme agent *path*, ktorý v prípade, že máme súčasne požiadavku na presun aj vhodného kandidáta na smer, zvolí tento smer tým, že ho prepíše do rovnomenného bloku **path**.

III.d. Keďže je to relatívny smer po vykonaní malého pohybu, by nám bol už len málo platný. Preto si musíme aspoň približne od tejto chvíle počítať do bloku integral o koľko sme sa odchýlili. To realizuje agent *integrate* sledujúci aktuátor **turn**. Pokiaľ robot používa servomotory alebo krokové motory a behá po vhodnom podklade, tak sa to vcelku dá (a ešte lepšie to ide v simulátore).

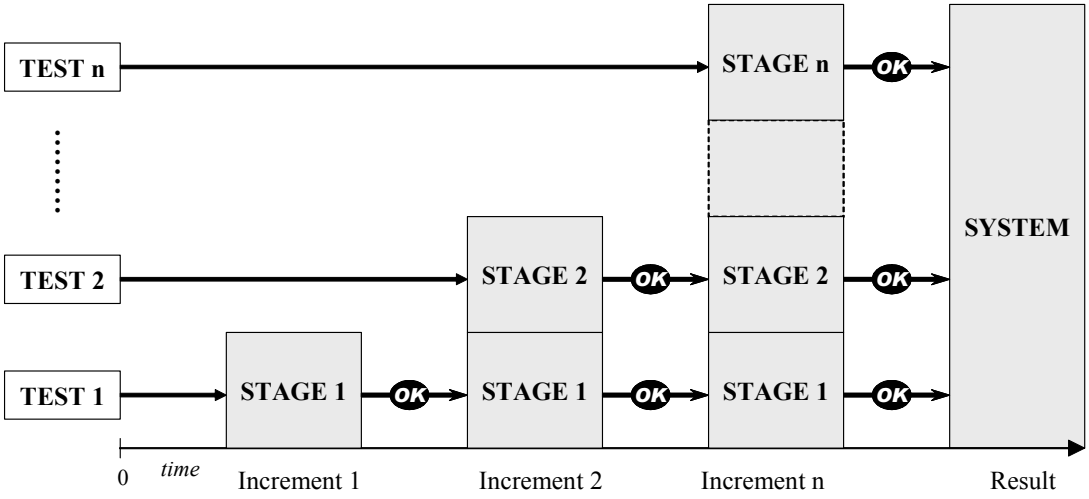
III.e. Teraz už stačí pridať agent *pathplan*, ktorý z blokov **path** a **integral** vie spočítať o koľko by sa bolo treba otočiť, aby sme boli natočení vo zvolenom smere presunu z jedného pôsobiska na druhé. Tento agent potom využije vrstvu WANDER a navrhne tento smer ako náhodné otáčanie – zapíše ho do bloku **wander**, čím realizuje otáčanie riadené. Výhoda tohto postupu je v tom, že systém neprestane obchádzať prekážky, ale bude pritom držať zvolený smer (hoci môže dôjsť k istému posunutiu). Presun sa zastaví vypršaním platnosti bloku **path**. V tomto prípade je výhodnejšie aby *pathplan* umlčal *wander* a teda na blok **wander** zapísal s vyššou prioritou. Nič menej, aj keby tak neurobil, EXPLORE by neprestala fungovať. Iba by sa do správania robota vkradlo akési záhadné správanie: sem tam by sa zvrtol, akoby chcel preč a následne by sa zvrtol úplne opačne akoby si uvedomil, že to bol zlý nápad. Takéto správanie je tu neúčelné a tak sa zdá vhodné ho pomocou priority eliminovať, ale na druhej strane, keď pozorujeme hmyz, zisťujeme, že je preň typické (entomológovia ho poznajú pod názvom „turn alternation behaviour“ [Migita 2004]) a sú známe prípady, keď má svoj účel. Tak si každý môže vybrať či preferuje efektivitu alebo biomimetiku.

S týmito tromi vrstvami už robot prechádza prakticky celým priestorom, ktorý v prípade našej reimplementácie predstavoval 3D model laboratória PRISM. Samozrejme povaha tohto priestoru sa nám premietla do viacerých konštánt v kódoch agentov a riešenie odladené pre toto prostredie by hypoteticky v inom prostredí nemuselo fungovať. Neskúšali sme to otestovať v nejakej inej scéne, ale predpokladám, že keby mala charakter kancelárskych priestorov, tak to bude chodiť.



Obrázok č. 57 Ukážky zo simulátora v ktorom bol ALLEN reimplementovaný

Z uvedeného príkladu jednoznačne vyplýva jedno poučenie. Subsumpčná architektúra môže byť v agent-space nielen vyjadrená, ale môže pre agent-space poskytnúť adekvátnu metódu vývoja, založenú na inkremenálnom vývoji zdola-nahor. Na základe tejto metódy rozdelíme vývoj systému na inkrementy, pričom pre každý vopred definujeme len to, ako ho otestovať, nie ako ho urobiť. V porovnaní s bežným softwarovým inžinierstvom tu stanovíme nad tzv. prípadmi použitia (use-case) akúsi hierarchiu a do implementácie sa pustíme bez toho, že by sme všetky z nich premietli do návrhu popisujúceho ako bude systém zostrojený. Návrh a implementáciu urobíme len pre prvú sadu testov. Keď výsledok prejde otestovaním, začneme systém rozširovať od inkrement pokrývajúci ďalšiu sadu testov. Po jeho implementácii musíme systém otestovať nielen pre túto novú sadu, ale pre všetky doterajšie sady testov (nemusíme to robiť iba vtedy, ak máme istotu, že pridaný inkrement sa za podmienok konania predchádzajúcich testov na globálnom správaní neprejavuje). Tak je naša vývojová metóda otvorená voči dodatočnému pridávaniu prípadov použitia, čo býva pri štandardných metódach vývoja značný problém. Po úspešnom pridaní všetkých inkrementov dostávame výsledný systém (Obrázok č. 58).



Obrázok č. 58 Inkrementálny vývoj zdola-nahor

Kardinálnym problémom tohto prístupu je využívanie starších inkrementov novšími. Pri postupe zdola nahor totiž nebudujeme žiadne rozhrania ktoré by použitie toho, čo už máme, tým, čo ešte len vyvinieme, neskôr umožnili. Agent-space sa tomuto problému úspešne bráni tým, že implementované časti komunikujú skrz bloky, ktoré sú všeobecne prístupné a možno sa z nich dozvedieť čo sa v systéme deje, ako aj možno cez ne činnosť systému ovplyvniť. Analogický trik používa subsumpčná architektúra, len miesto nepriamej komunikácie používa zdvojovanie vedenia medzi modulmi, supresory a inhibítory. Vďaka tejto analógii môžeme používať pri vývoji systému v agent-space rovnaké triky ako v rámci subsumpčnej architektúry. Videli sme to už v predchádzajúcom príklade a potvrdíme to príkladom, ktorý nevznikol transformáciou riešenia v subsumpčnej architektúre, ba dokonca by ani nešiel do tejto architektúry pretransformovať.

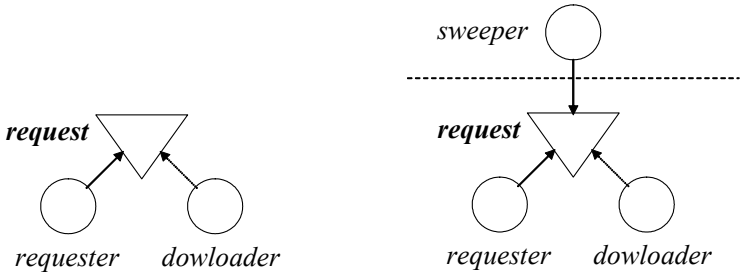
Príklad č. 15

Systém UDCS/QNET, ktorého základnú časť predstavil Príklad č. 13, samozrejme prešiel testovaním, ale reálne podmienky sa v tomto prípade dali len aproximovať. Po nasadení a dlhodobejšom behu bolo zistené, že sa v malom počte prípadov stáva, že určité dáta stanica

zmeria, ale do historickej databázy sa nedostanú. Šlo o takmer zanedbateľné množstvo, na ktoré - pri zvážení toho, že podstatne viac dát chýba preto, že sa skutočne nezmerajú – sa dalo v pokoji aj zabudnúť. Predsa však bola snaha zdokonaliť zber, pokiaľ by to nebolo nákladné a neohrozilo by to spoľahlivosť prevádzky systému. K zdokonaleniu sa dalo postaviť dvomi spôsobmi.

Prvý – klasický – by diktoval, aby sa buď hľadala nejaká chyba v súčasnej implementácii alebo nejaký rozdiel medzi predpokladanými a skutočnými podmienkami. Na základe toho by sa dospelo k pochopeniu prečo k chybe dochádza a potom by sa systém upravil tak, aby sa s tým vysporiadal.

Druhý spôsob spočíval v ponechaní systému tak, ako je a v pridaní pomerne primitívneho inkrementu, ktorý by zariadil, aby sa systém pokúsil chýbajúce dáta stiahnuť ešte raz. Hoci sa každému exaktne zmýšľajúcemu človeku musí pri takomto plátaní otvárať nožík vo vrecku, tento spôsob poskytoval presne to, čo sme potrebovali. Agent-space podporuje takéto zmýšľanie tým, že takéto záplaty sa na systém pridávajú extrémne ľahko: systém netreba kompilovať, preinštalovať, ani len reštartnúť ho netreba. Proste pustíme ďalšie agenty a je to. V tomto prípade stačí jediný agent (nazvime ho *sweeper*), ktorý bude prehľadávať databázu a generovať požiadavky pre chýbajúce dáta. Tie zapíše do prostredia spôsobom ako to robí agent *requester* (Obrázok č. 49). Zvyšok systému s nimi bude pracovať ako keby to boli dosiaľ nevybavené požiadavky. Ku kolízii *sweepera* s *requesterom* nemôže prísť, lebo ten zapisuje len aktuálne požiadavky. Preto *sweeper* nemôže pôvodný systém pokaziť. Sweeper musí samozrejme databázu prehľadávať tak, aby žiadal o určité chýbajúce dáta len raz a platnosť požiadaviek musí nastaviť tak, aby nevyvolali hneď zavolanie na danú stanicu, ale aby sa vybavenie týchto požiadaviek zviezlo pri stiahnutí najbližších aktuálnych dát (Obrázok č. 59).



Obrázok č. 59 Rozšírenie systému UDCS/QNET pomocou „záplaty“

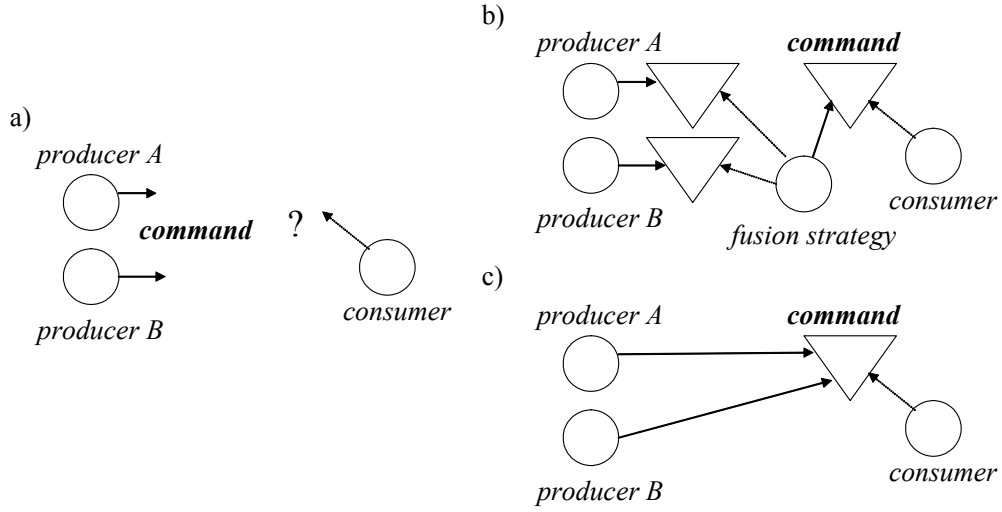
S tými, ktorým takéto plátanie nie je po chuti, si v ďalšom radi položíme otázku či náhodou príroda pri vývoji živých systémov nekoná nevedome tak, ako keby vedome plátala. Teraz sa však ešte na chvíľu vráťme ku vzťahu agent-space a subsumpčnej architektúry. Už sme ukázali, že subsumpčnú architektúru do agent-space pretransformovať možno a dokonca je to pre ňu veľkým prínosom. Natíska sa teraz otázka ako je to s opačnou transformáciou. Už sme pravda naznačili, že existujú systémy, ktoré sa pretransformovať nedajú, ostáva však vyšpecifikovať, ktoré to sú a prečo sa nedajú. Navyše treba zvážiť, že subsumpčná architektúra je tu v dvojnásobnej nevýhode: jednak poznáme jej kritiku a jednak do agent-space – ako do vlastného diela – môžeme podľa potreby pridávať chýbajúce mechanizmy (tak ako sme pridali napríklad prioritu).

Začnime pohľadom na tradičnú kritiku subsumpčnej architektúry a povedzme si ako sa s ňou vysporadúva agent-space. Pripomeňme, že ide o tri výčitky: neprístupnosť k vnútornému stavu, nemožnosť porušenia úrovne kompetencie a nemožnosť dátovej fúzie [Rosenblatt - Payton 1989] (viď kapitolu Predchádzajúce práce).

Čo sa týka neprístupnosti k vnútornému stavu, je výčitka úplne na mieste. Všimnime si napríklad v reimplementácii robota ALLEN, že blok **force** je z hľadiska AVOID redundantný, zmysel nadobúda len v spolupráci s vrstvou WANDER. Ako vieme čo má byť v blokoch a čo môže ostať vo vnútornom stave agentov? Pokiaľ niečo zabudneme vo vnútornom stave, už sa k tomu z neskorších vrstiev nedostaneme. Bolestným ale účinným riešením tu je zakázať aby nejaký vnútorný stav agenty vôbec mali. Teda vyžadovať, aby všetky agenty boli čisto reaktívne. Ak programátor dodrží toto pravidlo, nedostane sa neskôr do problémov. Dodržiavať toto pravidlo je niekedy bolestné, ale spravidla sa to oplatí. Žiadny programátor ho však z dôvodu lenivosti nedokáže plniť na 100%, veľkým pokrokom je však už to, keď čistú reaktivitu preferuje. Paradoxne, čisto reaktívne agenty, ktoré sa pre svoje obmedzené schopnosti zdajú byť na prvý pohľad nevhodnými stavebnými jednotkami systémov, sú pri agent-space tými najvhodnejšími.

Čo sa týka nemožnosti porušenia úrovne kompetencie, ani samotná subsumpčná architektúra na tom nie je v tomto ohľade tak zle, ako jej kritici pripisujú. Napríklad pri robotovi ALLEN pri presune z jedného pôsobiska na druhé, ktoré riadi vyššia vrstva EXPLORE, je v prípade natrafenia na prekážku pohyb robota prevzatý vrstvou AVOID. Je to síce vďaka tomu, že vyššia vrstva odstúpi tak povediac dobrovoľne, ale porušenie kompetencie to je. Agent-space je na tom ešte lepšie. Pri nej sa energia musí vynakladať práve na neporušovanie kompetencie (write() a delete() s prioritou), jej porušovanie je základná vlastnosť. Okrem toho priority umožňujú nielen vyšším vrstvám ovplyvniť nižšie, ale aj nižším zabezpečiť sa proti vplyvu vyšších. Navyše priorita, s ktorou agenty volajú write() a delete() sa môže meniť podľa situácie a tak má zmysel hovoriť skôr o skorších a neskorších vrstvách, než o nižších a vyšších (z hľadiska kompetencie). Vrstvy sú dané len poradím v rámci vývoja, nie samotnou kompetenciou.

No a čo sa týka nemožnosti dátovej fúzie, túto kritiku načisto odmietame. Bolo by samozrejme fajn, keby sa hodnoty v blokoch dokázali nielen navzájom prepisovať, ale aj nejako vzájomne integrovať (napríklad spočítavať). Nevyhnutne by to však viedlo k používaniu veľmi obmedzeného množstva dátových typov, ktoré by bolo možné do blokov uložiť a to zásadne odmietame. Preto jednoznačne preferujeme prepisovanie hodnôt a pokiaľ je nevyhnutná ich fúzia, nech sa realizuje pomocou viacerých blokov a agenta, ktorý ju vykonáva (Obrázok č. 60).



Obrázok č. 60 Problém dátovej fúzie

a) problém, b) riešenie na základe fúzie, c) odporúčané riešenie

Čo by sme boli schopní pripustiť (možné je to však len pri implementácii v rámci OOP), by bolo obmedzenie hodnôt, ktoré do blokov možno napísať na objekty odvodené od určitej triedy, pre ktorú by bola definovaná metóda na fúziu dvoch jej inštancií. Túto by volalo prostredie a jej defaultná definícia by bola prepísanie starého novým. Kto by chcel nejakú konkrétnu fúziu, mohol by vo vlastnej réžii túto metódu prekryť v závislosti od konkrétnej podtriedy. Pokladáme to však za zbytočné a nečisté, každopádne sa z toho kľuje problém časovej synchronizácie.

Vráťme sa teraz k vymedzeniu toho, čo sa nedá v subsumpčnej architektúre vyjadriť, respektíve čo ide vyjadriť len hypoteticky. V prvom rade sa z princípu nedajú vyjadriť riešenia pracujúce s blokmi, ktorých mená sa v čase menia. V subsumpčnej architektúre by to viedlo k nekonečnému počtu komunikačných vedení. Príklad č. 13 opisujúci systém UDCS/QNET je ukážkou takéhoto systému, lebo aby sa požiadavky nepomiešali, každá musí mať v mene ID stanice aj čas. Reálne neprekonateľné potiaže vyvolá však aj veľký konečný počet blokov, ako aj akákoľvek štruktúra využívajúca, že určitý agent pracuje s blokom, ktorého meno si prečítal v inom bloku. V týchto prípadoch by zodpovedajúce moduly subsumpčnej architektúry museli mať privedený na vstup obrovské množstvo komunikačných vedení a následne by ich vnútorný kód bol nezmyselne zložitý.

Vo všeobecnosti zhruba platí, že kým je určité riešenie v agent-space založené len na kooperácii medzi agentami, hľadá sa jeho analógia v subsumpčnej architektúre pomerne ľahko. Akonáhle sa však v riešení objavia prvky konkurencie medzi agentami (obzvlášť konkurencie medzi rovnocennými agentami), nastanú problémy. Rovnocenná konkurencia totiž volá po rozšírení subsumpčnej architektúry o spájanie komunikačného vedenia. Supresor síce vedenia spojí, ale nie rovnocenne. Museli by sme preto na každý takýto spoj použiť špecifický modul a aj to je možné len v rámci jednej vrstvy. Dosiahnuť rovnocennú konkurenciu dvoch častí ležiacich v rôznych vrstvách v subsumpčnej architektúre vždy znamená vytiahnuť ich do najvyššej vrstvy a potlačiť ich v nižších vrstvách.

Nakoľko sú tieto nedostatky subsumpčnej architektúry závažné? (O koľko je agent-space reálne lepšia?) Existujú reálne problémy, pre riešenie ktorých sú vhodné štruktúry agent-space netransformovateľné alebo len ťažko transformovateľné do subsumpčnej architektúry? Štruktúry, ktoré využívajú premenlivé mená blokov, sa vyskytujú v každom riešení, ktoré musí akumulovať historický priebeh nejakých dát. A takých problémov je neúrekom. Naproti tomu štruktúry používajúce konkurenciu sa nezdajú byť na prvý pohľad reálne užitočné. Veď načo by sme do nášho systému zavádzali princípy, ktoré vedú k tomu, že jedna zložka systému súperí s inou? Nebolo by rozumnejšie zariadiť, aby radšej spolupracovali?

Na túto otázku existuje jednoduchá odpoveď: v umelých systémoch by to snáď bolo aj rozumnejšie, ale pri modelovaní systémov živých sa bez toho nezaobídeme. Rad príkladov poukazuje na to, že konkurencia je nielen v živých systémoch prítomná, ale občas (hlavne za naaranžovaných podmienok, ktoré sa normálne nevyskytujú) je schopná vyplávať na povrch v podobe zaujímavých, až údiv vyvolávajúcich, zlyhaní. Neveríte?

Príklad č. 16 [Lúčny 2002]

Skúste prosím čo najrýchlejšie odpovedať na nasledujúce dve otázky:

Koľko je do tucta korunáčok ?

A koľko päťdesiathaliernikov ?

Pokiaľ tento experiment nepoznáte, poznačte si odpoveď. Skúste sa prípadne opýtať aj svojich kolegov.

Vtip je v tom, že 98% odpovedí na tento test je 12, 24 (môj diplomant Andrej Kružic na to vykonal celkom seriózny experiment konzultovaný s psychológom Imrichom Ruiselom). Pokiaľ ste aj vy odpovedali 12, 24, prečítajte si otázky ešte raz a zrejme uznáte, že jediná správna odpoveď na ne je 12, 12. Tak aj odpovedajú tie 2% a mizivý počet tvoria žiadne a exotické odpovede (žiadnych je málo preto, lebo sme vhodne nastavili časový limit, do ktorého bolo treba odpovedať a exotických preto, že naozaj len málokto odpovie napríklad 60, 60 alebo 12, 60 – hoci aj to sa stáva). (Pozor inak na to, že tento experiment sa nedá premigrovať do iného jazyka. S američanmi napríklad máme tú skúsenosť, že na otázku s dolármi a štvrťdolármi – poldoláre žiaľ nemajú – neodpovedajú 12, 48 ale 12, 3 a navyše sa im to zdá v poriadku. Nemáme ich však takú vzorku aby sme niečo o tom s istotou tvrdili.)

Podstatné na tomto experimente je zamyslieť sa, prečo k zlyhaniu dôjde. Zrejme je to preto, že nejaká časť mysle zaoberajúca sa peniazmi je príliš aktívna a v tomto prípade konkuruje časti, ktorá zodpovedá za všeobecné počty.

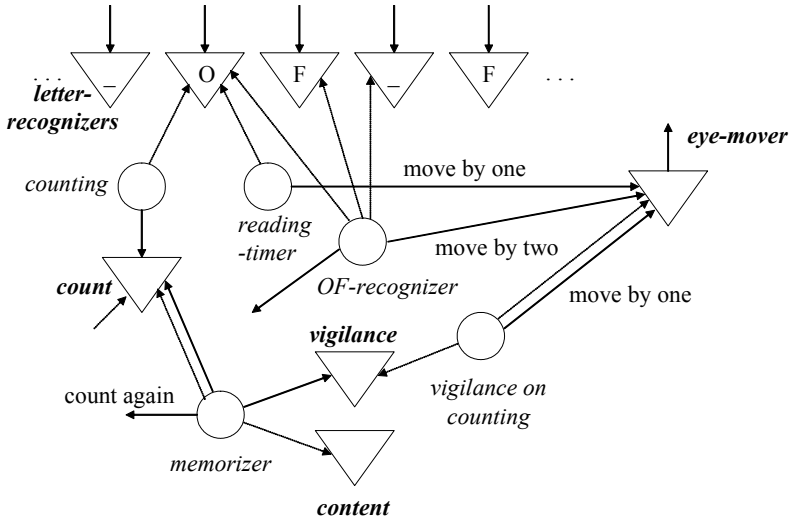
Pokiaľ by sme sa snažili modelovať túto situáciu subsumpčnou architektúrou, zrejme by sme prisúdili peniazom vyššiu úroveň a zriadili supresiu na výstup z počtov. Tak by sme ale 12, 12 nikdy nedostali. Tieto odpovede by sme museli prisúdiť tomu, že dotyční majú inú štruktúru mysle. Tie 2% však netvoria žiadni géniovia, ani ľudia, ktorí nemajú vzťah k peniazom, je to skôr náhoda. S agent-space nie je problém modelovať túto situáciu aj za predpokladu, že všetci majú rovnakú štruktúru mysle a je navyše možné naladiť frekvencie zúčastnených agentov tak, aby sa percentuálne zastúpenie jednotlivých odpovedí zhodovalo s nameranými údajmi (podrobnosti sú v [Lúčny 2002]).

Príklad č. 17

Podobný príklad: spočítajte počet písmen F v nasledujúcom texte[24]:

```
FINISHED FILES ARE THE RE-
SULT OF YEARS OF SCIENTIF-
IC STUDY COMBINED WITH THE
EXPERIENCE OF YEARS
```

Koľko vám vyšlo?



Obrázok č. 61 Modelovanie počítania písmen v texte

Napočítali ste tri, štyri, päť alebo šesť? Vskutku je ich tam šesť, ale tri z nich (tie čo sú súčasťou OF) sú „neviditeľné“. Pritom testovaná skupina ľudí sa podľa odpovedí zvyčajne rozdelí na štyri približne rovnako veľké skupiny a nezáleží dokonca ani na tom, či vedia po anglicky alebo nie (hoci obyčajne sa tento jav vysvetľuje práve tým, že F v OF sa po anglicky číta ako V – s ohľadom na nezávislosť výsledku na znalosť jazyka je však toto vysvetlenie neudržateľné). Akurát by mali byť dospelí a mali by vedieť dobre čítať (napríklad prváčikovia čo sa učia čítať, vždy napočítajú šesť). Podľa nášho názoru dobré vysvetlenie spočíva v tom, že o posun pozornosti na jedno písmenko súťažia viaceré mechanizmy. Hoci chceme čítať po písmenách nepodarí sa úplne vypnúť vyššie úrovne spracovania textu, ktoré v záujme zvýšenia rýchlosti preskakujú určité písmená, vnímajú ich viac naraz alebo ovplyvňujú rýchlosť presunu pozornosti. Prečo práve OF robí takéto problémy a napríklad ON nie? Nevieme, ale domnievame sa, že v tom má prsty podobnosť F a E, respektíve dlhší čas potrebný pre rozpoznanie F, ktorý je následkom tejto podobnosti. Každopádne, keď nemáme veľké tlačené písmená, tento efekt nenastáva. Keby sme ale model založili len na vzťahu rýchlosti presunu pozornosti po texte a času potrebného na rozpoznanie F, tak by napočítal buď tri, alebo šesť. Náš model (Obrázok č. 61) musí zahrnúť nejaký prvok náhodnosti, ktorý raz F započíta a raz nezapočíta, pričom pomer pravdepodobností je cca 1:1. S agent-space sa to dá namodelovať práve zápisom dvoch konkurenčných agentov do rovnakého bloku (na obrázku sú to *reading-timer* a *OF-recognizer* zapisujúce do bloku ***eye-mover***). Keď tieto agenty majú rovnakú frekvenciu, je pravdepodobnosť prevzatia hodnoty jedného z nich 0.5^{25}.

Príroda konkurenciu nielen používa, ale má na to aj dobrý dôvod. Poznáme prípady, kedy logicky správna kooperačná štruktúra je zložitejšia než konkurenčná štruktúra, ktorá nie je logicky správna, ale jej úspešnosť za bežných podmienok dosahuje takmer 100%. Za nastrojených špeciálnych podmienok potom môže úspešnosť dramaticky poklesnúť. Napríklad pri experimente s korunáčkou logicky správna štruktúra musí pracovať minimálne nad štyrmi blokmi, zatiaľ čo štruktúra založená na konkurencii vystačí s tromi blokmi [Lúčny 2002]. Evolučne môže byť teda výhodnejšie prikloniť sa k jednoduchšej štruktúre, ktorá nefunguje stopercentne, ale

- fylogenéza sa k nej môže dopracovať na menší počet mutácii
- ontogenéza ju dokáže vytvoriť na menší počet prechodných štádií
- počas svojho pôsobenia je energeticky menej náročná

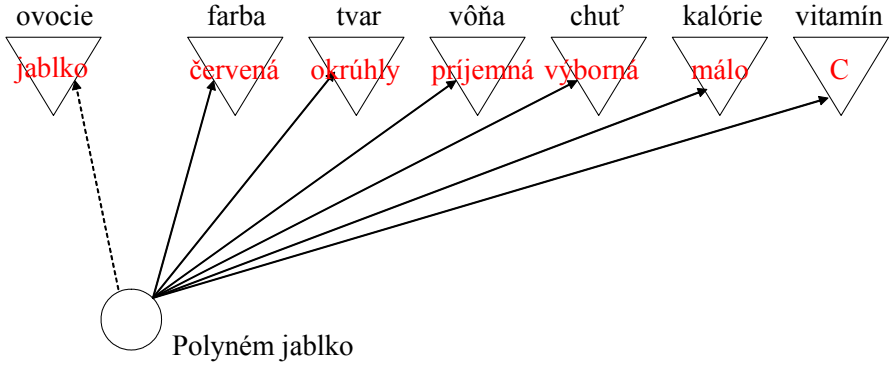
Práve takéto štruktúry sú tie, ktorá sa v subsumpčnej architektúre vyjadrujú len veľmi ťažko, prípadne sa to ani principiálne nedá (netvrdíme však, že by sa nedala rozšíriť aby to možné bolo). Napriek tomu je subsumpčná architektúra veľmi významným míľnikom na ceste k takýmto štruktúram. Neviem si predstaviť, že by sme sa k nim bez jej znalosti vôbec dopracovali.

Vzťah k Minského spoločenstvu mysle

Okrem subsumpčnej architektúry bola pre agent-space silnou filozofickou motiváciou práca Marvina Minského týkajúca sa modelovania mysle ako spoločenstva agentov [Minaky 1986]. Táto motivácia spočívala v prvom rade v tom, že Minského model leží niekde uprostred medzi neurónovou sieťou a mašinériou založenou na logike. Ide tu jednoznačne o komputačno-symbolický prístup, ale nie je odkázaný na logickú inferenciu. A to sú presne parametre, ktoré má aj agent-space. Z tejto podobnosti vyplýva možnosť vyjadriť Minského štruktúry z K-spojení (viď kapitolu Predchádzajúce práce) ako vzory správania sa reaktívnych agentov nad blokmi v prostredí. Relevantnosť týchto štruktúr sa dá potom vyhodnocovať podľa toho, ako

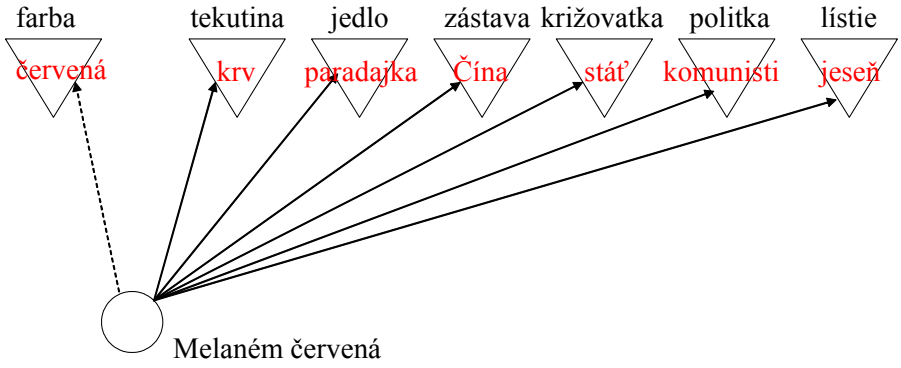
často sa objavuje ich potreba pri realizácii riadiacej štruktúry modelov živých systémov. Pokúsime sa preto v agent-space vyjadriť aspoň tie Minského štruktúry, ktorým vieme v modeloch podľa agent-space pripísať nejaký význam.

Polynémy sú štruktúry, ktoré na základe predmetu aktivujú atribúty, ktoré sa k nemu viažu. V agent-space je polyném možné realizovať jediným agentom, ktorý na základe hodnoty určitého bloku nastavuje hodnoty viacerých blokov. Pritom sémantika týchto blokov zodpovedá tomu, že nastavované bloky zodpovedajú atribútom monitorovaného bloku (Obrázok č. 62).



Obrázok č. 62 Polyném realizovaný v agent-space

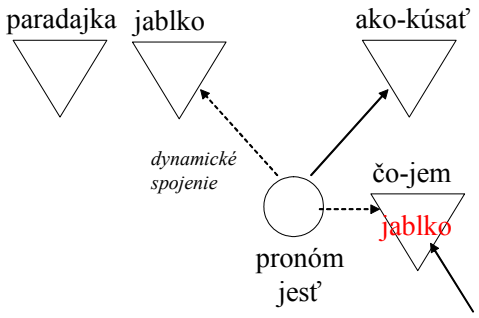
Melaném má naopak na základe atribútu aktivovať predmety, ktoré ho majú. Syntakticky bude v našom vyjadrení zhodný s polynémom, iba sémantika sa zmení (Obrázok č. 63).



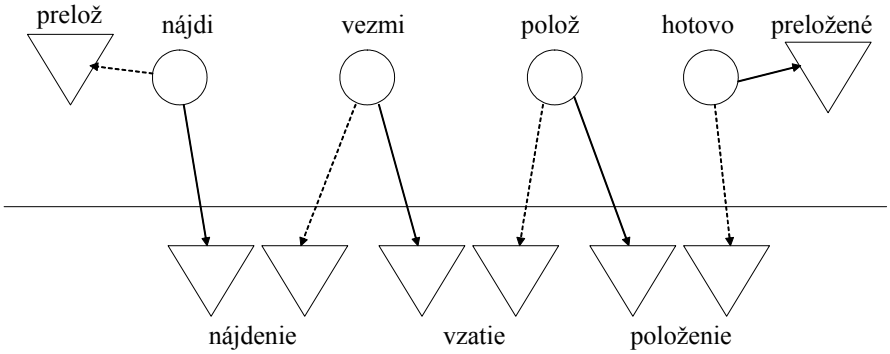
Obrázok č. 63 Melaném realizovaný v agent-space

Pronómy ukazujú na objekty, sú to „premenné“, poskytujú referenciu. Ide tu presne o to, čo subsumpčná architektúra principiálne neposkytuje, ale agent-space áno. Základom pronómu v agent-space je agent, ktorý číta v určitom bloku názov iného bloku s ktorým potom pracuje. Tak môžu iné agenty zápisom názvu bloku do určitého bloku s pevným pomenovaním priviesť určitý agent k zmene jeho činnosti (presnejšie povedané bude vykonávať rovnakú činnosť nad inými blokmi). (Obrázok č. 64).

Scripty vykonávajú postupnosť krokov a sú najzaujímavejšou z týchto štruktúr. Pri prepise do agent-space totiž získavajú vlastnosti, ktoré nie sú Minským predpokladané, inak ich ale prepísať nejde. Typický script v agent-space znázorňuje Obrázok č. 65. Pozostáva zo série agentov, v ktorej postupne jeden aktivuje druhý. Vzniká tak distribuovaná štruktúra, ktorá realizuje praobyčajnú sekvenčnú (prípadne aj cyklickú) procedúru. Tak je to však len v prípade, keď bloky, ktoré aktivujú nasledujúci agent, menia svoju hodnotu iba na základe ukončenia pôsobnosti predchádzajúceho agenta. Vzťah medzi aktivačnými blokmi nasledujúceho agenta a blokmi, ktoré zapisuje predchádzajúci agent, však nemusí byť taký zošnurovaný. Aktivačné bloky môžu byť dokonca úplne izolované (Obrázok č. 65) a môžu sa spoliehať na to, že určitý vnem z prostredia sa môže vyskytnúť iba ako následok dokončenia určitej fázy scriptu. Kým tá prvá stratégia vedie k stopercentne správnemu poradiu fáz, tá druhá vedie k odolnosti voči výnimkám, ktoré sa pri realizácii môžu vyskytnúť: pri nejakom nezdare sa systém automaticky vráti k niektorej predchádzajúcej fáze, ktorej východzie podmienky ostali naplnené.



Obrázok č. 64 Pronóm realizovaný v agent-space



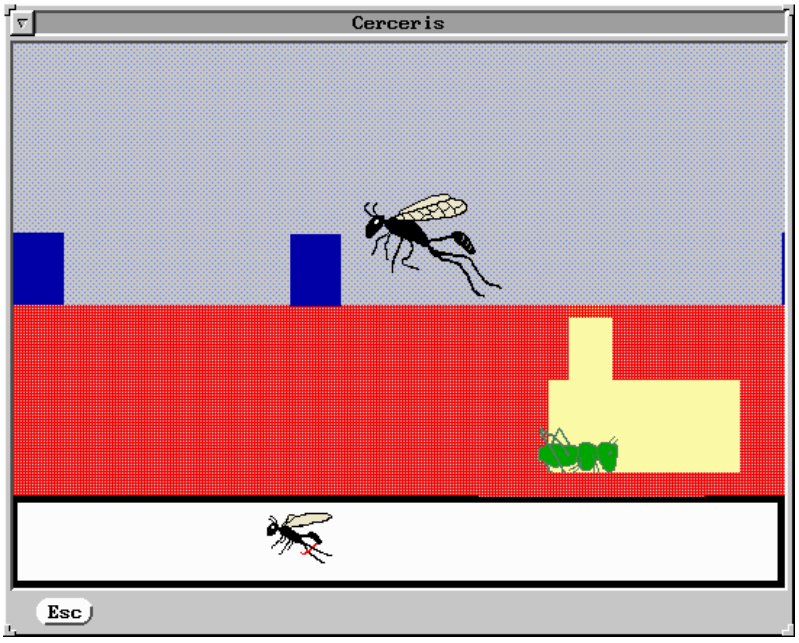
Obrázok č. 65 Script realizovaný v agent-space

Stratégia s aktiváciou cez prostredie sa vyznačuje zaujímavou vlastnosťou: po návrate do určitej predošlej fázy dokáže systém preskočiť fázy, ktoré zopakovať netreba, pokiaľ je možné z prostredia bezprostredne zistiť, že to naozaj netreba. Naopak, nemilosrdne sa zopakujú fázy, ktorých ukončovacia podmienka sa z prostredia nedá vyčítať.

Príklad č. 18 [Lúčny 2001a]

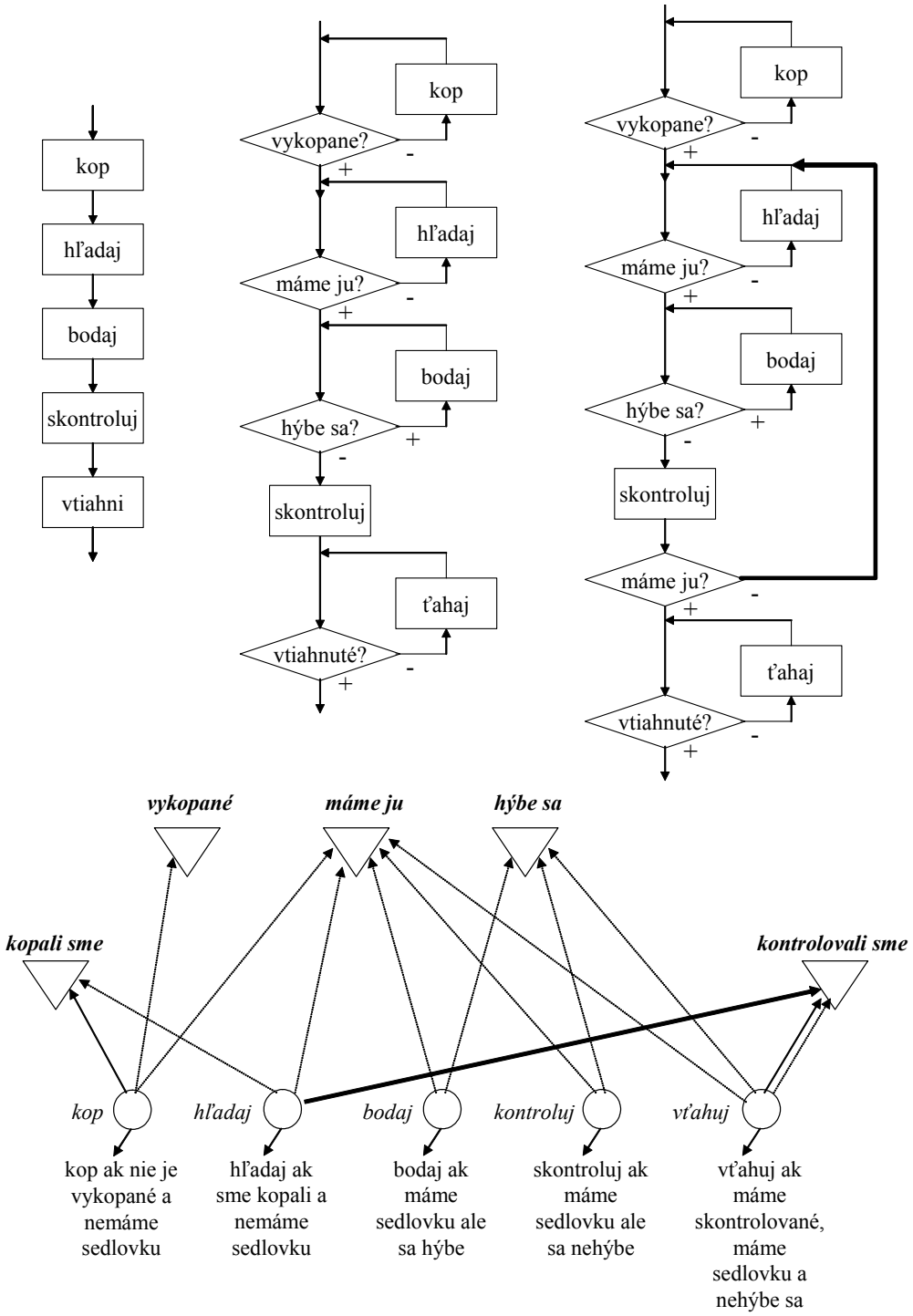
Kutavka rodu Cerceris je samotárska osa známa najmä pre hrôzostrašný spôsob akým zabezpečuje pre svoje larvy dostatok čerstvej potravy: žihadlom pichne svoju korisť (každý druh loví špecifickú obeť z radov chrobákov, napríklad sedlovku) tak, že ju ochrnie a odtiahne ju do podzemnej komôrky, kde na ňu nakladie vajíčko. Z toho sa vyliahne kutavčia larva, ktorá potom paralyzovanú obeť za živa zje, pričom jej to trvá niekoľko mesiacov. Kutavka pritom realizuje vždy rovnaký rituál: najprv vyhrabe komôrku, potom uloví obeť, položí si ju pred komôrku, vlezie do komôrky skontrolovať ju, potom do nej vtiahne obeť, znesie na ňu vajíčko, vylezie z komôrky a zacelí jej vchod. Pokiaľ by ste kutavke vo fáze keď vlezie na kontrolu do komôrky sedlovku odtiahli, ale nie príliš ďaleko, aby ju kutavka našla, opäť by ju priniesla ku komôrke. Ale opäť by ju položila pred vchod a šla vykonať kontrolu (ktorú už práve pred chvíľou vykonala), čiže vliezla by do komôrky. Takto má experimentátor možnosť opakovať odtiahnutie sedlovky od vchodu a udržiavať kutavku v bezvýchodiskovom cykle[26] [Hass 1970]. Kutavka ľubovoľný počet krát zakaždým sedlovku pritiahne ku vchodu komôrky, ale nepoučí sa a nevtiahne ju hneď dnu, naopak, ide vykonávať kontrolu, ktorú už vykonala mnoho krát a s pričinením experimentátora opäť o sedlovku príde. Na druhej strane v momente, keď sedlovku nájde, nejde ju druhý raz pichať žihadlom, ale túto fázu preskočí.

Egon Gál položil v [Gál 2000] provokujúcu otázku: čo chýba kutavke v porovnaní so správaním človeka, ktorý by sa takto nedal vodiť za nos? S predstavou, že kutavka je z hľadiska zložitosti svojho riadiaceho mechanizmu len jednooký medzi slepými (teda relatívne jednoduchý tvor vykonávajúci pomerne zložité správanie) sme sa pokúsili túto situáciu modelovať umelým riadiacim systémom v agent-space, ktorý bol prepojený s jednoduchým grafickým simulátorom (Obrázok č. 66).



Obrázok č. 66 Simulátor správania sa kutavky rodu cerceris pri zakladaní potomstva

Zakladanie potomstva sme simulovali práve štruktúrou typu script. Dospeli sme k záveru, že jedným z možných vysvetlení by bolo, že kutavka sa v mechanizme postupnej aktivácie agentov, ktoré sa do realizácie scriptu zapájajú, spolieha viac na podnety z prostredia, než na vlastnú pamäť.



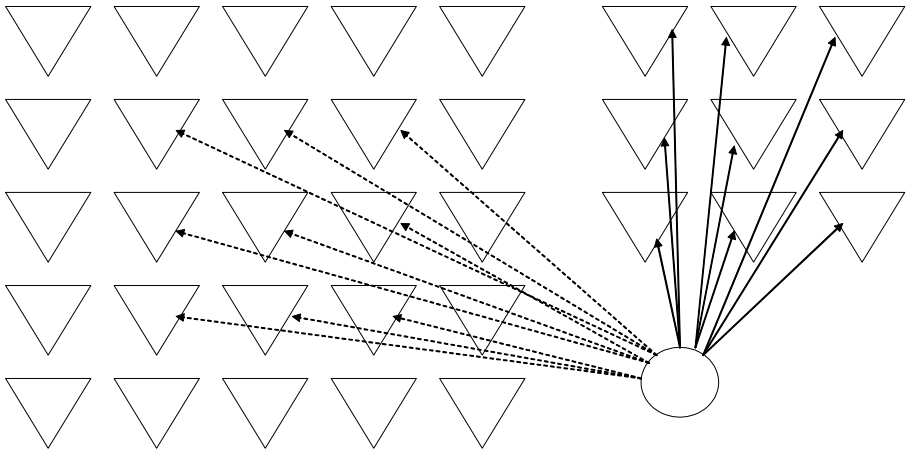
Obrázok č. 67 Porovnanie klasickej štruktúry scriptov a ich obdoby v agent-space

Hore vľavo: script realizovaný tradičnou procedúrou, ktorý nedokáže žiadnu fázu ani zopakovať, ani vynechať. Hore v strede: script realizovaný tradičnou procedúrou, ktorý dokáže zbytočné fázy vynechať okrem kontroly, ktorú nevie otestovať na základe stavu prostredia. Hore vpravo: script realizovaný tradične, ktorý navyše dokáže určité fázy opakovať, ale za cenu toho, že sa to musí explicitne vyjadriť (hrubá šípka). Dolu: script v agent-space, ktorý dokáže fázy vynechať i opakovať, pričom je to vyjadrené implicitne. Musí však využívať pamäť (bloky kopali sme, kontrolovali sme) a starať sa o zabúdanie jej obsahu (hrubá šípka). (Z úsporných dôvodov tu neuvažujeme párenie, hľadanie komôrky po ulovení sedlovky, kladenie vajíčka a zapečatenie komôrky)

Spoliehanie sa na prostredie umožňuje modelu kutavky vyjadreného v agent-space ošetrovať výnimky bez toho, že by to bolo explicitne nakódované (čo pre scripty predpokladá Minsky). Umožňuje to taktiež preskočiť fázy, ktoré netreba opakovať (čo by však išlo aj pri pôvodnej Minského predstave) (Obrázok č. 67). Táto stratégia zlyháva len v tom prípade, keď sa zo stavu prostredia nedá určiť či sú už splnené kritériá pre ukončenie fázy (a teda ju rovno preskočíme) alebo nie. Pri opakovaní bodania žihadlom sa táto fáza preskočí, lebo sedlovka už nekladie odpor, ale komôrka sa musí skontrolovať, lebo to sa nedá vidieť. Rozdiel medzi kutavkou a nami by bol potom v tom, že my sa oveľa viac spoliehame na to, čo si o celej veci pamätáme.

Nedostatkom tohto vysvetlenia je, že kutavka by si to pokojne mohla pamätať, veď napríklad to, že má komôrku už vykopanú si musí pamätať – inak by po vykopaní kúsok odletela a kopala zas. Dokonca si musí pamätať i samotný fakt, že ju skontrolovala, inak by ju po skontrolovaní kontrolovala zas. Pamätanie je problematické v tom, že treba k nemu implementovať aj zabúdanie – aby kutavka nepokladala za skontrolovanú komôrku vykopanú na inom mieste – čo by zodpovedalo v modeli tomu, že agent riadiaci kopanie komôrky vymazáva blok, v ktorom je táto informácia uložená. Keby to bolo spravené takto, kutavka bez problémov vynechá zbytočnú kontrolu. Aby sa dosiahla zhoda s pozorovaním musí toto mazanie urobiť agent, ktorý riadi nájdenie sedlovky. To má logiku v tom, že keby kutavke počas kontroly sedlovku niekto nadobro uchmatol, musela by ísť loviť novú a pritom by sa dostatočne vzdialila od komôrky na to, aby ju bolo potrebné opäť kontrolovať. Zbytočnou kontrolou v nastrojených podmienkach teda platí za efektívne správanie v prirodzených podmienkach, čo iste rada zaplatí.

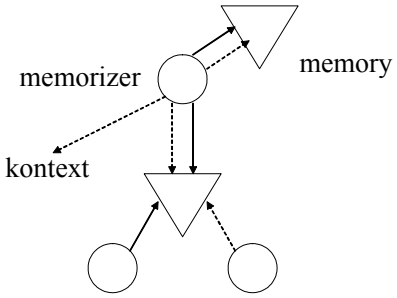
Z hľadiska porovnania agent-space a tradičných procedúr na modelovanie scriptov je teda agent space lepšia v tom, že dokáže ošetrovanie výnimiek a následné opakovanie fáz implementovať implicitne. Za to však nemôže využívať informáciu o tom na ktorej inštrukcii scriptu sa nachádza (tzv. PC – program counter) a všetko si musí buď zistiť v prostredí alebo zapamätať. Zapamätané však treba niekedy vymazať. Zaujímavým riešením je zabúdať pomocou časovej platnosti blokov. Takto to však kutavka zrejme nerobí, lebo to by sa prejavilo práve tým, že počas výcviku by ju to prestalo baviť a šla by si vykopať inú komôrku a uloviť inú sedlovku. Skontrolovanie komôrky musí zabúdať práve pri hľadaní sedlovky, inak zhoda s pozorovaním nebude. Nevysvetlili sme teda prečo sa kutavka správa tak ako sa správa, ale dokázali sme prísť na to akú vnútornú štruktúru by musel jej model mať, aby sa správal rovnako.



Obrázok č. 68 Realizácia frame v agent-space

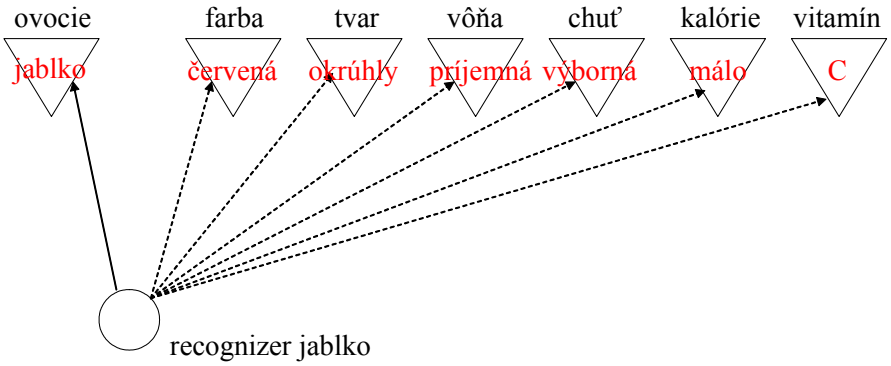
Rámce sú ďalšou Minského štruktúrou, ktorú sme modelovali. Obsahujú informáciu usporiadanú podľa príbuznosti, umožňujú meniť aktuálne spracúvaný objekt a zachovať pritom kontext. V agent-space to zodpovedá kopírovaniu časti skupiny blokov (Obrázok č. 68). Dobrým príkladom je čítanie textu, kde z viacerých písmen, ktoré dokážeme zachytiť na sietnici, vyberáme písmeno na ktoré práve sústreďujeme pozornosť a ešte zopár písmen okolo neho. Vďaka tomuto mapovaniu môžu potom iné agenty pracovať s pojmami ako „susedné písmeno“ a pritom používať fixné názvy blokov.

Memorizery sa učia kontext na základe určitého kľúča. Keď sa potom tento kľúč objaví, obnovia kontext. Sú to tzv. myšlienkové skratky. Tieto hrajú nezastúpiteľnú úlohu pri realizácii anticipácie. V agent-space je to prirodzené realizovať čisto reaktívnym agentom, ktorý si určitú informáciu odkladá do určitého „súkromného“ bloku. Pekné je, že v agent-space sa vždy môže ľahko zo „súkromného“ stať „verejné“. Tak je možné memorizery použiť aj na preškolenie systému.



Obrázok č. 69 Memorizer vyjadrený v agent-space

Recognizery rozpoznávajú na základe atribútov predmet. Sú duálne k polynémom a preto budú mať veľmi podobnú štruktúru, len miesto čítania bude zápis a naopak (Obrázok č. 70). Recognizery sú určené pre spracovanie vnímaných dát a preto sa často vyskytujú i v najjednoduchších modeloch.



Obrázok č. 70 Realizácia recognizeru v agent-space

Vcelku teda možno sumarizovať, že medzi Minského štruktúrami a agent-space je istý vzťah. Spočíva v tom, že pre našu architektúru dokážu Minského štruktúry poslúžiť ako zmysluplné návrhové vzory. Pri vytváraní modelov je preto vhodné ich mať na pamäti asi tak, ako je pri

objektovom programovaní vhodné pamätať na návrhové vzory objektov. Podobne ako medzi objektmi existujú objekty pre objekty, existujú aj agenty pre agenty a tieto možno štandardizovať.

Záver kapitoly

Úlohou tejto práce nie je riešiť konkrétne úlohy UI, ale poskytnúť architektúru na ich riešenie. Preto sme v tejto kapitole uviedli viacero príkladov a sústredili sme sa na tie detaily, ktoré svedčia o prínose našej architektúry. Tomu zodpovedá aj určitá neúplnosť výkladu jednotlivých príkladov. Väčšina z týchto projektov bola samostatne publikovaná, takže čitateľa so záujmom o všetky detaily určitého konkrétneho projektu odkazujem na literatúru. Väčšina týchto projektov vznikla z iniciatívy autora práce a preto je dostupná na jeho domovskej stránke www.microstep-mis.com/~andy. Ďalším zdrojom – keďže dosť príkladov spadá do oblasti mobilnej robotiky – je na www.robotika.sk. Kontaktovať možno aj autora na adrese andy@microstep-mis.com.

---

[16] Deutschove klamy udávajú zoznam predpokladov, ktoré neplatia pre počítačovú sieť. Pozri stranu 26

[17] „Inteligencia je schopnosť adaptovať sa na zmeny v prostredí„

[18] A už vôbec nechcem povedať, že by za každý chlievik bol zodpovedný jeden gén. Na to je génov príliš málo. Sada všetkých chlievikov je však podľa môjho názoru v genóme skomprimovaná. Dekomprimácia sa pritom opiera o fyzikálne a chemické zákony, čo predstavuje ohromné množstvo informácie, ktorá nemusí byť ku kódu priložená, lebo je všadeprítomná. Preto je možné s tak malým počtom povedať tak veľa. Podrobnejšie k danej problematike viď [Kováč 2004].

[19] Za túto jednoduchú ale kľúčovú myšlienku vďačím môjmu školiteľovi. Dovolil by som si preto tento princíp nazvať Kelemenovým princípom.

[20] Dobrý prehľad možno nájsť napr. v [Csontó 2002]

[21] Ide tu o reálny projekt, na ktorom už pracujeme, ale jeho ukončenie je zatiaľ v nedohľadne

[22] Kde je každá informácia zakódovaná do váh, čo sú reálne čísla – tým pádom každú informáciu musíme nejako do tejto reprezentácie previesť, inak s ňou nemôžeme pracovať

[23] http://www.robotika.sk

[24] http://www.grassyknoll.homestead.com/5307WorldFinishedFiles.html

[25] Pozor, tých 0.5 to nie je kvôli tomu, že by boli fázy časovačov posunuté o polovicu periódy, ale preto, že sa fázy sa zhodujú a teda je reálne poradie impulzov je náhodné a to v pomere 1:1

[26] http://www.alanturing.net/turing_archive/pages/Reference%20Articles/What%20is%20AI.html a http://www.hans-hass.de/Englisch/Human_Animal/1_02_Innate_Behavior_Patterns.html

VII. Záver

Cieľom tejto práce bolo uviesť netradičnú softwarovú architektúru, ktorá je vhodná na implementáciu systémov, ktoré napodobňujú alebo modelujú správanie živých či inteligentných systémov. Jej vhodnosť pre tieto účely sa dosahuje za cenu opustenia tradičných názorov na tvorbu software i na riešenie úloh umelej inteligencie. V krátkosti ich zhrnieme:

1. Cieľom výpočtov prebiehajúcich v počítači nie je výsledok, ale správanie – vzťah medzi nekonečnou postupnosťou vstupov a výstupov zo systému.
2. Systému je dovolené robiť chyby, nevyžaduje sa od neho 100% presnosť. (Takýto systém môže byť vyslovene nevhodný na riešenie mnohých klasických úloh, ktoré zverujeme výpočtovej technike – napríklad na počítanie peňazí.) Vítame u neho schopnosť vykoľajiť sa, pokiaľ je schopný vrátiť sa do starých koľají alebo si nájsť iné.
3. Systém nemusí pracovať optimálne, nemusí šetriť prostriedky ktoré má k dispozícii. Radi ich obetujeme, ak tým uľahčíme tvorcovi systému prácu.
4. Systém nemusí mať jeho tvorca plne pod kontrolou, ani počas jeho behu, ani počas jeho vývoja. Musí mať samozrejme možnosť overiť si, že vyvinul správny systém, avšak ako sa v ňom táto správnosť dosahuje, môže mu ostať skryté, presnejšie povedané môže to ostať v neskomprimovanej podobe: „je to tak, lebo tých 20000 parametrov je naladených práve tak ako je“.
5. Inteligencia systému nie je sústredená v nejakom jeho podsystéme, ktorý sa na túto funkciu špecializuje, ale vyžaduje všadeprítomný princíp, podľa ktorého sa tvoria tie najzložitejšie aj tie najjednoduchšie modely.
6. Inteligencia systému nespočíva vo všeobecnom algoritme, ktorý je aplikovateľný na riešenie veľkého množstva prípadov, ale v dosiahnutí súladu medzi obrovským počtom špecifických algoritmov aplikovateľných pre jednotlivé prípady. Takto je všeobecný algoritmus na riešenie problémov nahradený všeobecným princípom ako zosúladiť obrovské množstvo špecifických parciálnych riešení.

Je doslova bolestné prijať túto predstavu, ale nie je to až v takej kontradikcii so všetkým čo poznáme, než by sa na prvý pohľad zdalo. Ide skôr o logické vyústenie toho, čo už roky používame. Z hľadiska tradičnej tvorby software je logické, že ak pri extrémne zložitých úlohách tvorcovia zlyhávajú, keď strácajú nad vytváraným systémom kontrolu, mali by sa hľadať také metódy tvorby, ktoré túto kontrolu nevyhnutne nevyžadujú. Z hľadiska umelej inteligencie, tradičná honba za správnou podobou reprezentácie poznatkov a inferenčného mechanizmu, ktorý s nimi manipuluje, sa mení na honbu ako použiť veľa rôznych reprezentácii a veľa rôznych inferenčných mechanizmov odrazu.

Riešenie, ktoré je v tejto práci predložené, zakladá sa na predstave, že každý systém pozostáva z veľkého počtu spolupracujúcich programov, tzv. reaktívnych agentov, ktoré sa neustále točia v nekonečnom cykle, v ktorom sledujú ostatné agenty, prípadne užívateľa či fyzikálne prostredie a na základe toho si neustále volia čo podniknú. Navyše, spôsob tejto voľby je jednoduchý (neopiera sa o žiadnu špeciálnu metódu tradičnej umelej inteligencie, ako je to u tzv. deliberatívnych agentov). Podstata riešenia bude potom spočívať jednak v spôsobe akým medzi sebou tieto agenty interagujú, jednak v metóde ako z nich budovať systém, v ktorom sa z tohto množstva špecifických trikov vytvára želané globálne správanie.

Táto práca vychádza z viacerých prameňov, ktorých pôvod sa dá vysledovať až do polovice 80-tych rokov. Staršie zdroje jej poskytujú predovšetkým filozofické postoje, zatiaľ čo technologicky sledujeme modernejšie trendy. Do istej miery sa dá povedať, že používame

nový jazyk na staré myšlienky. Veľa vlastností uvádzanej architektúry je priamo zdedených po subsumpčnej architektúre. Sú to dobré vlastnosti a preto ich radi zdedíme. Originalita tu spočíva v tom, že nám nie je známe, že by niekto idey obsiahnuté v Brooksovej subsumpčnej architektúre či v Minského prácach dával vôbec do súvisu s agentovo orientovaným programovaním. Filozofické myšlienky, s ktorými pracujú mnohí, premieňame na programátorské prostriedky. Znášame tu čosi z neba na zem.

Túto prácu možno považovať za reformuláciu. Prináša však nielen novú formu, ale aj rozširuje pôvodný obsah. Jej originalita a prínos sa dá zhrnúť do nasledovných bodov:

1. Prezentujeme skutočné a všeobecne použiteľné agentovo-orientované programovanie, ktoré nie je zviazané so žiadnym konkrétnym simulátorom.
2. V porovnaní s objektovo-orientovaným programovaním ponúkame v rámci nášho druhu agentovo-orientovaného programovania nový model spracovania udalostí, ktorý odstraňuje jednu z najnepríjemnejších vlastností objektovo-orientovaného prístupu, či „event-driven“ programovania vôbec, spočívajúcu v bujnení „handlerov“ a ich rôznych rozhraní s narastajúcim počtom typov spracovávaných udalostí.
3. Ponúkame architektúru pre interaktívne systémy, ktorá má niekoľko jedinečných vlastností najmä dátové toky many:many a implicitné vzorkovanie.
4. Vývojové metódy považované za úzko späté z oblasťou mobilnej robotiky generalizujeme a umožňujeme ich použitie pre akýkoľvek interaktívny systém
5. Na rozdiel od prác z ktorých vychádzame, používame pri modelovaní živých organizmov konkurenciu ako jeden z možných štrukturálnych princípov
6. Práca obsahuje veľké množstvo podnetných príkladov, ktoré sa dajú použiť ako vzor pre ďalšie aplikácie prezentovanej architektúry alebo tvorbu alternatívnych architektúr.

Možno samozrejme očakávať aj kritiku a to nasledovnú:

1. Navrhovaná architektúra chce slúžiť umelej inteligencii, ale pritom odsúva nabok jej tradičné metódy: expertné systémy, neurónové siete, fuzzy systémy a pod. Nuž v tejto práci sme sa zameriavali na čistotu jej používania, ale hybridný systém nie je problém urobiť tým spôsobom, že do časti agenta, ktorá volá akcie (fáza select), vložíme takýto tradičný mechanizmus, či celý systém takýchto mechanizmov. Agenty s jednoduchou voľbou akcie potom obstarajú reaktívne globálne správanie a zvyšok bude mať deliberatívnu podobu.
2. Práca nepoužíva moderné trendy z oblasti softwarového inžinierstva a to aj tých, ktoré sa týkajú multiagentových systémov (napríklad AgentUML). Nuž naozaj, na rozdiel od tradičných postupov, kde sa budujú čoraz zložitejšie vývojové nástroje, ktoré majú umožniť vývojárovi zvládnuť narastajúcu komplexnosť, ponúkame ako alternatívne riešenie použiť jednoduché vývojové nástroje a vzdať sa stopercentnej kontroly nad vyvíjaným systémom. Tu si treba uvedomiť, že zaškolenie vývojárov na súčasné prostriedky vývoja komplexných systémov trvá zhruba 10 až 12 mesiacov. Naša ponuka je teda ekonomicky lákavá. Strata kontroly tu nevedie k zhoršeniu, ale k zlepšeniu kvality riešenia a preto je to aj ponuka akceptovateľná. Nemusíte sa báť, že vývojár nepokryje nejakú situáciu o ktorej vie a padne kvôli tomu lietadlo. Naopak, môžete dúfať, že pokrytá bude aj situácia o ktorej nikto netuší, že by mohla nastať a lietadlo vtedy nepadne.
3. Obmedziť komunikáciu medzi agentami na nepriamu je príliš obmedzujúce. Tu sme opäť strážili čistotu, nič menej aj v rámci uvádzaných implementácií netreba nič pridať, aby sa dala priama komunikácia použiť. Napríklad pri implementácii nad OOP priamej komunikácii zodpovedá zavolanie metódy patriacej k aplikačnému kódu jedného agenta iným agentom. Volajúci agent sa musí akurát odniekiaľ dozvedieť referenciu na agent volaný. Nič ľahšie, volaný agent sa sám zapíše do určitého bloku (presnejšie zapíše tam smerník na seba) a volajúci si ho z neho prečíta.

4. Chýba použitie štruktúr v agent-space pre genetické programovanie. Dúfame, že táto problematika bude obsahom budúcich prác. Povedať, že sa toto do práce nedostalo pre krátkosť času, by vzhľadom na to, že práca je výsledkom desaťročného úsilia, nebolo na mieste.

Každá slušná práca prináša množstvo podnetov pre jej rozširovanie a pokračovanie. Z toho zároveň vyplýva, že žiadna práca nemôže byť vyčerpávajúca a v istom bode musí končiť. Do tohto dokumentu sa už nedostali napríklad moje posledné rozšírenia implementácie o oneskorovanie triggrov a výskum ich významu, ktorý na systéme reálneho času spočíva v možnosti integrácie údajov bez rizika fúzie vzájomne nekonzistentných dát.

Z hľadiska pokračovania tejto práce je veľmi naliehavou potrebou vyskúšať evolúciu riadiaceho systému vyjadreného v našej architektúre. Úspechom v tejto oblasti by sme dali opodstatnenie našej téze, že hľadáme štruktúry, ktoré sú hodné evolvovania. Zavedenie evolúcie nad agent-space vyžaduje vyriešiť niekoľko problémov. V prvom rade je potreba obmedziť sa na jeden typ dát a zariadiť, aby sa týmto typom dala vyjadriť aj časová platnosť dát. Nemyslím, že by tu bolo dobré ísť na reálne čísla, pohrávam sa skôr s myšlienkou použiť stringy a definovať nad nimi nejakú rafinovanú operáciu, ktorá stringu pevne priradí jeho platnosť. Túto by bolo potrebné založiť na rozklade stringu. Tak bude možné vyjadriť názvy blokov, ich obsahy, aj ich časovú platnosť. Aplikačný kód agentov by sa dal urobiť podobným spôsobom ako je bežné pre genetické programovanie Johna Kozu, len miesto stromovej štruktúry by to bola štruktúra podobná zväzu, do ktorej by odspodu šiel vstup z blokov a navrchu vychádzal výstup do blokov a v uzloch by boli operácie nad usporiadanými dvojicami stringov. Obmedzili by sme sa na budenie časovačom a frekvenciu by sme odvodili zo štruktúry toho zväzu. Jedinec by pozostával z niekoľkých agentov, takže kríženie by sa nerobilo nad zväzmi, ale len nad množinami zväzov a mutácia jedného zväzu by nepredstavovala problém. Ťažko povedať, čo by to dalo, k realizácii je ešte cesta ďaleká. Zaujímavým špeciálnym prípadom by bolo keby sme stringy zúžili na logické formuly.

Ďalším pokračovaním by mali byť taktiež experimenty v modelovaní živých systémov. Na tomto poli sa postupne prepracúvame k stále dokonalejším simulátorom, lebo je to kardinálne aj z hľadiska skúmania štruktúr riadiacich systémov, ktoré nimi hýbu. Rozhranie realizujúce interakciu s prostredím a dynamika samotného prostredia sú rovnako dôležité ako riadiaci systém. To dáva len dve možnosti: dokonalý simulátor alebo stelesnený systém vybavený solídnymi perifériami. Obe možnosti majú výhody a nevýhody a obe sú dosť obtiažne. Snažíme sa sledovať obe možnosti a to vývojom 3D simulátorov s fyzikálnym enginom a mobilnými robotmi vybavenými kamerou a mikrofónom.

Nevyhnutnou zložkou budúcnosti architektúry agent-space sú taktiež aplikácie tejto architektúry pre vývoj reálnych produktov. Táto architektúra je síce už dávno použitá pre niekoľko komerčných monitorovacích systémov, ale treba stále dvíhať latku obtiažnosti a snažiť sa riešiť náročné úlohy umelej inteligencie. V súčasnosti beží v MicroStep-MIS zaujímavý projekt, kde sa na báze tejto architektúry (samozrejme nielen na nej) vyvíja senzor na meranie pokrytia oblačnosti. Nádejne tiež vyzerá spolupráca s DENSO Corrporation, v rámci ktorej sa má naša architektúra použiť na modelovanie scény snímanej z idúceho auta (na ktorej sa dá založiť varovný systém pre šoféra).

Pravdu povediac, osobne som s touto prácou celkom spokojný. Konečné rozhodnutie o jej kvalite však ponechávam na každom čitateľovi.

Literatúra

[1] Arkin, R.: *Behavior-based Robotics (Intelligent robots and autonomous agents).* MIT Press, Cambridge, Mass., 1998

[2] Beňušková, Ľ. – Kvasnička, V. – Pospíchal, J.: *Hľadanie spoločného jazyka v kognitívnych vedách.* IRIS, Bratislava, 2000

[3] Brooks, R.: *Achieving Artificial Intelligence Through Building Robots.* A.I. Memo 899, AiLab, MIT, Cambridge, Mass., 1986a

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

[5] Brooks, R: *A Robots that Walks: Emergent Behaviors from a Carefully Evolved Network.* Neural Computation 1:2, Summer, 1989

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

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

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

[9] Brooks, R. – Stein, L.: *Building Brains for Bodies.* A.I. Memo 1439, AiLab, MIT, Cambridge, Mass., 1993

[10] Bryson, J.: *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents.* AITR 2002-003, AI Lab, MIT, Cambridge, Mass., 2002

[11] Cruse, H. – Bartling, C. – Cymbalyuk, G. – Dean, J. – Dreifert, M.: *A modular artificial neural net for controlling a six-legged walking system.* Biological Cybernetics 72 (1995), Springer-Verlag, pp. 421-430

[12] Csontó, J. – Palko, M.: *Umelý život.* Elfa, Košice, 2002

[13] Csuhaj-Varjú, E. – Kelemen, J. – Kelemenová, A. – Paun, Gh.: *Eco-grammar systems – A Gramatical Framework for Studying Life-like Interactions.* Artificial Life 3 (1997), pp. 1-28

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

[15] Doran, J.: *Distributed AI and its Applications.* In: Advanced Topics in Artificial Intelligence (Mařík, V. – Štepánková, O. – Trappl, R., eds.) Springer-Verlag, Berlin, 1992, pp. 368-372

[16] Eckel, B.: *Myslíme v jazyku Java.* Grada, Praha, 2000.

[17] Ferber, J.: *Multi-Agent Systems - An Introduction to Distributed Artificial Intelligence.* Addison-Wesley, Harlow, 1999

[18] Ferko, A. – Kalaš, I. – Kelemen, J.: *Počítač Hamlet.* Mladé letá, Bratislava, 1990

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

[20] Forest, S.: *Emergent computation: self-organizing, collective, and cooperative phenomena in natural and artificial computing networks.* Introduction to the Proceedings of the Ninth Annual CNLS Conference. In: Emergent Computation (Forest, S., ed.), The MIT Press, Cambridge, Mass., 1991, pp. 1-11

[21] Gál, E.: *Intuitívna psychológia a kognitívne vedy.* In: Kognitívne vedy III (Kvasnička V., Pospíchal J. eds.), CHTF, STU, Bratislava, 2000

[22] Gál, E. – Kelemen, J.: *Myseľ, telo, stroj.* Bradlo, Bratislava, 1992

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

[24] Carriero, N. – Gelernter, D.: *Linda in context.* Communications of the ACM 32(4), 1989, pp. 444-458

[25] Gamma, E. – Helm, R. – Johnson, R. – Vlissides, J.: *Návrh programů pomocí vzorů.* Grada, Praha, 2003

[26] Goodwin, R.: *Formalizing Properties of Agents.* Report CMU-CS-93-159, Computer Science Department, Carnegie Mellon University, Pittsburg, Pensylvania, 1993

[27] Hamer, D., Copeland P.: *Geny a osobnost.* Portál, Praha, 2003

[28] Hass, H.: *The Human Animal: The Mystery of Man's Behavior.* G. P. Putnam's Sons., 1970, pp. 27

[29] Holland, J. H.: *Emergence – from Chaos to Order.* Addison-Wesley, Reading, Mass., 1998

[30] Jennings, N.: *On agent-based software engineering.* Artificial Intelligence 117, 2000, pp. 277-296
[31] Jones, L. J. – Flynn, A. M.: *Mobile robots: Inspiration to Implementation.* AK Peters. Ltd., Wellesley, Mass., 1993
[32] Kautz, H. – Selman, B. – Coen M.: *Bottom-up Design of Software Agents.* Communications of the ACM 37(7), 1994, pp. 143-147.
[33] Kelemen, J.: *Myslenie, počítač ...* Spektrum, Bratislava, 1989
[34] Kelemen, J. – Ftáčnik, M. – Kalaš, I. – Mikulecký, P.: *Základy umelej inteligencie.* Alfa, Bratislava, 1992
[35] Kelemen, J.: *Multiagent symbol systems and behavior-based robots.* Applied Artificial Intelligence 7, 1993, pp. 419-432
[36] Kelemen, J.: *Strojovia a agenty.* Archa, Bratislava, 1994
[37] Kelemen, J.: *From statistics to emergence - exercises in systems modularity.* In: Multi-Agent Systems and Applications, Luck, M. – Mařík, V. – Štepánková, O. – Trappl, R., (eds.), Springer, Berlin, 2001a, pp. 281-300
[38] Kelemen, J.: *Kybergolem.* Votobia, Olomouc, 2001b
[39] Kelemen, J.: *The Agent Paradigm.* Computing and Informatics, Vol.22. (2003), pp. 513-519
[40] Kelemen, J. – Kelemenová, A.: *A grammar theoretic treatment of multi-agent systems.* Cybernetics and Systems 26, 1992, pp. 621-633
[41] Kelemenová, A. (ed.): *Proceedings on the MFCS98 satellite workshop on Grammar Systems*, Silezian University, Opava, 1998
[42] Knight, K.: *Are Many Reactive Agents Better Than a Few Deliberative Ones?* In: Proc. IJCAI '93, Chambery, 1993, pp. 132-137.
[43] Kováč, L.: *Komentovaný úvod do kognitívnej biológie.* In: Kognice a umělý život IV (Kelemen, J. – Kvasnička, V., eds.), FPF SU, Opava, 2004, pp. 233-258
[44] Kubík, A.: *Agenty a multiagentové systémy.* Slezská univerzita, Opava, 2000
[45] Kubík, A.: *Agent-Based Computational Economies: A Language-Theoretic Approach.* PhD thesis, University of Economics, Bratislava, 2002
[46] Kvasz, L.: *O revolúciách vo vede a ruptúrach v jazyku vedy.* UK, Bratislava, 1998
[47] Lea, D.: *Concurrent Programming in Java: Design Principles and Patterns,* Addison-Wesley, 1999
[48] Van Leeuwen, J. – Wiedermann, J.: *Beyond Turing Limit: Evolving Interactive Systems.* In. SOFSEM 2001: Theory and Practice of Informatics (Pacholski, L. – Ružička, P., eds.), Springer-Verlag, 2001, pp. 90-109
[49] Luck, M. – Mařík, V. – Štepánková, O. – Trappl, R., (eds.): *Multi-Agent Systems and Applications*, EASSS, Prague, 2001
[50] Lúčny, A.: *Emergentné správanie v kolóniách agentov.* Diplomová práca, Katedra umelej inteligencie, Matematicko-fyzikálna fakulta, Univerzita Komenského, Bratislava, 1994
[51] Lúčny, A.: *Architektúra inteligentných programových systémov.* Návrh dizertačnej práce, Ústav informatiky, Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, Bratislava, 1997.
[52] Lúčny, A.: *Reaktívny model inteligentného systému.* In: Kognitívne vedy II (Kvasnička, V. – Pospíchal, J., eds.), CHTF STU Bratislava, 1999a, pp. 75-84
[53] Lúčny, A.: *Architektúra inteligentných programových systémov.* In. Gramatické, multiagentové a znalostní systémy (Kelemen, J., ed.), FPF SU, Opava, 1999b, pp.75-98
[54] Lúčny, A.: *Hľadanie kvalitatívneho rozdielu.* In: Kognice a umělý život (Kelemen, J. – Kvasnička, V. – Pospíchal, J., eds.), FPF SU, Smolenice, 2001a, pp. 167-176
[55] Lúčny, A.: *Modelovanie správania živých systémov pomocou subsumpčnej architektúry.* Rigorózna práca, Katedra informatiky, Matematicko-fyzikálna fakulta, Univerzita Komenského, Bratislava, 2001b
[56] Lúčny, A.: *Spaces and Reactive Agents under QNX4.* QNX Tools and Technology, Bratislava, 2001c
[57] Lúčny A.: *Modelovanie kognitívneho zlyhania multiagentovým systémom so stigmergickou komunikáciou.* In: Kognice a umělý život II (Kelemen, J. – Kvasnička, V., eds.), FPF SU, Milovy, 2002, pp. 119-132

[58] Lúčny, A.: *Baldwinov efekt: Plyn a brzda evolúcie.* In: Kognice a umělý život III (Kelemen, J. ed.), FPF SU, Opava, 2003, pp. 151-164
[59] Lúčny, A.: *Refinement and Emergency versus Design and Predetermination.* In: Proceedings of IWES 2004 (Ueda, K. – Monostori, L. – Márkus, A., eds.), Budapest, 2004a, pp. 13-18
[60] Lúčny, A.: *Modelovanie umelého pohybu na báze VRML.* In: Kognice a umělý život IV (Kelemen, J. – Kvasnička, V., eds.), FPF SU, Opava, 2004b, pp. 353-362
[61] Lúčny, A.: *Building Complex Systems with Agent-Space Architecture.* Computing and Informatics, Vol. 23 (2004c), pp. 1001-1036
[62] Lúčny, A.: *Building Control Systems of Mobile Robots with Agent-Space architecture.* 1st CLAWAR/EURON Workshop, 2004d
[63] Maes, P.: *How To Do the Right Thing.* A. I. Memo No. 1180, MIT AI Lab, Cambridge, Mass., 1989
[64] Markoš, A. – Kelemen, J.: *Berušky, andělé a stroje.* Dokořán, Praha, 2004
[65] Mařík, V. – Štepánková, O. – Lažanský, J.: *Umělá Inteligence (1).* Academia, Praha, 1993
[66] Mařík, V. – Štepánková, O. – Lažanský, J.: *Umělá Inteligence (2).* Academia, Praha, 1997
[67] Mařík, V. – Štepánková, O. – Lažanský, J.: *Umělá Inteligence (3).* Academia, Praha, 2001
[68] Mařík, V. – Štepánková, O. – Lažanský, J.: *Umělá Inteligence (4).* Academia, Praha, 2003
[69] Mařík, V. – Štepánková, O. – Trappl, R.. (eds.): *Advanced Topics in Artificial Intelligence.* Springer-Verlag, 1992
[70] Mataric, M. J.: *Interaction and Intelligent Behavior.* Submitted to the Department of Electrical Engineering in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy, MIT AI Lab, Cambridge, Mass., 1994
[71] McFarland, D. – Bosser, T.: *Intelligent Behavior in Animals and Robots.* The MIT Press, Cambridge, Mass., 1993
[72] Migita, M.: *Modeling Phenotypic Ambiguity as a Source of Emergence.* In: Proceedings of IWES 2004 (Ueda, K. – Monostori, L. – Márkus, A., eds.), Budapest, 2004, pp. 1-12
[73] Minsky, M.: *The Society of Mind.* Simon&Schuster, New York, 1986
[74] Minsky, M.: *Konštrukcia mysle.* Archa, Bratislava, 1996
[75] Minsky, M.: *Emotion machine.* In Proc. EMCSR'2000 (Trappl, R. ed.), Vienna, 2000
[76] Minsky, M.: *Emotion machine.* http://web.media.mit.edu/~minsky/, 2004
[77] Mlichová, R.: *Niektoré experimenty so subsumpčnou architektúrou v nedeliberatívnej robotike.* Diplomová práca, Katedra umelej inteligencie, Matematicko-fyzikálna fakulta, Univerzita Komenského, Bratislava, 1993
[78] Naik, D.: *Internet – standardy a protokoly.* Computer Press, Brno, 1999
[79] Návrat, P. a kol.: *Umelá inteligencia.* STU, Bratislava, 2002, pp. 303-328
[80] Nilsson, N.: *Artificial Intelligence: A New Synthesis.* Morgan Kaufmann Publishers, San Francisco, California, 1997
[81] Nwana, H.: *Software agents: An overview.* Knowledge Engineering Review, vol. 11, 1996, pp. 1-40
[82] Parker, L. E.: *Adaptive Action Selection for Cooperative Agent Teams.* In: From Animals to Animates, Proc. 2nd International Conference on Simulation of Adaptive Behavior (Meyer, J. A. – Roirblar, H. – Wilson S., eds.), MIT Press, Cambridge, Mass., 1992a, pp. 442-450
[83] Parker, L. E.: *Local versus Global Control Laws for Cooperative Agent Teams.* A. I. Memo No. 1357, MIT AI Lab, Cambridge, Mass., 1992b
[84] Parker, L. E.: *Heterogeneous Multi-Robot Cooperation.* Submitted to the Department of Electrical Engineering in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy, MIT AI Lab, Cambridge, Mass., 1994
[85] Resnick, M.: *Turtles, Termites, and Traffic jams.* The MIT Press, Cambridge, Mass., 1994
[86] Ridley, M.: Genom. Portál, Praha, 2001
[87] Ronald, E. M. A. – Sipper, M. – Caprarrére, M. S.: *Design, observation, surprise! A Test of emergence.* Artificial Life 5 (1999), pp. 225-239
[88] Rosenblatt, J. K. – Payton, D. W.: *A Fine-Grained Alternative to the Subsumption Architecture for Mobile Robot Control.* In: Proceedings of the IEEE/INNS International Joint Conference on Neural Networks, Washington DC, vol. 2, 1989, pp. 317-324

[89] Rybár, J. a kol.: *Filozofia a kognitívne vedy.* IRIS, Bratislava, 2002
[90] Rybár, J. – Beňušková, Ľ. – Kvasnička, V. (eds.): *Kognitívne vedy.* Kalligram, Bratislava 2002
[91] Singh, P.: *Examining the Society of Mind*, Vol.22. (2003), pp. 521-543
[92] Snow, C.: *Concurrent Programming (Cambridge Computer Science Texts).* Cambridge University Press, Cambridge, 1992
[93] Stein, L.: *Imagination and Situated Cognition.* A. I. Memo 1277, AiLab, MIT, Cambridge, Mass., 1991
[94] Šešera, Ľ. – Mičovský, A.: *Objektovo-orientovaná tvorba systémov a jazyk C++.* Perfekt, Bratislava, 1994
[95] Thagard, P.: *Úvod do kognitivní vědy.* Portál, Praha, 2001
[96] Valckenaers, P. – Van Brussel, H. – Kollingbaum, M. – Bochmann O.: *Multi-agent Coordination and Control Using Stigmery Applied.* In: Multi-Agent Systems and Applications (Luck, M. – Mařík, V. – Štepánková, O. – Trappl, R., eds.), EASSS, Prague, 2001, pp. 317-334.
[97] Waldo, J: *Mobile Code, Coordination and Changing Networks.* CONCOORD, Lipari, 2001
[98] Wätjen, D.: *Function-dependent teams in eco-grammar systems.* Teoretical Computer Science 306 (2003), pp. 39-53
[99] Wooldridge, M.: *Introduction to MultiAgent Systems.* John Wiley & Sons, London, 2002.
[100] Wooldridge, M. – Jennings, N.: *Pitfalls of Agent-Oriented Development.* In proceedings of the 2nd International Conference on Autonomous Agents (Agents'98), Minneapolis / St. Paul, 1998
[101] Zrzavý, J.: *VRML Tvorba dokonalých www stránek.* Grada Publishing, Praha, 1999
[102] Žára, J.: *VRML 97 Laskavý průvodce virtuálními světy.* Computer Press, Brno, 1999