UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA METAMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Andrej Lúčny

Modelovanie správania živých systémov pomocou subsumpčnej architektúry.

Rigorózna práca

Bratislava 2001

Prehlásenie

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

Banka, 15.6.2001 *Andrej Lúčny*

Obsah

1. Úvod ... 3

2. Subsumpčná architektúra a jej pretavenie ... 5
- 2.1 Na prahu tvorby systému ... 5
- 2.2 Inkrementálny vývoj zdola nahor ... 6
- 2.3 Dekompozícia aktivitou ... 7
- 2.4 Vtieranie sa do činnosti ... 8
- 2.5 Nepriama komunikácia ... 8
- 2.6 Čisto reaktívne agenty ... 9
- 2.7 Transducery ... 10
- 2.8 Zhrnutie ... 10

3. Simulácia zakladania potomstva u kutavky cerceris bupresticida ... 13
- 3.1 Výber správania na aproximáciu umelým systémom ... 13
- 3.2 Analýza aproximovaného správania ... 14
- 3.3 Prostriedky na implementáciu ... 15
- 3.4 Simulátor vonkajšieho prostredia ... 17
- 3.5 Transducery ... 18
- 3.6 Význam náhody ... 19
- 3.7 Implementácia vrstvy lezenia ... 19
- 3.8 Implementácia vrstvy lietania ... 20
- 3.9 Implementácia vrstvy vyhýbania sa prekážkam ... 21
- 3.10 Implementácia vrstvy zakladania potomstva ... 22
- 3.11 Vyhodnotenie experimentov ... 30
- 3.12 Zobrazenie aktivity systému ... 35
- 3.13 Zhrnutie ... 35

4. Záver ... 37

References ... 39

1. Úvod

V polovici osemdesiatych rokov sa v oblasti umelej inteligencie objavil prevratne nový názor na to, ako konštruovať umelé systémy o ktorých by bolo možné v zmysle Turingovho testu povedať, že sú inteligentné. V porovnaní s tradičným názorom, že taký systém by mal pozostávať zo všeobecného algoritmu operujúceho nad špecifickými dátami vyjadrenými v jednotnej forme, nový smer ponúkol možnosť zložitej sústavy špecifických algoritmov, ktorej celková funkčnosť vzniká (emerguje) z ich vzájomnej interakcie a v ktorých sú dáta vyjadrené rôznymi spôsobmi [Brooks 1986a].

Ako príklad klasického systému si môžeme predstaviť expertný systém, v ktorom všeobecný inferenčný mechanizmus operuje nad faktami na základe problémovo špecifických pravidiel. Napriek veľkej vyjadrovacej schopnosti pravidiel, ukazuje sa potrebné v praxi dopĺňať pravidlá o akčné časti, fakty vyjadrené rámcovou reprezentáciou o rôzne démony a podobne, čo do istej miery poukazuje na potrebu používať špecifické algoritmy viazané na problémovú oblasť.

Slovami M. Minského nepotrebujeme na konštrukciu inteligentného systému nájsť univerzálny spôsob ako vyjadriť určité dáta a algoritmy, ale nájsť spôsob, ktorý nám umožní integrovať množstvo takýchto spôsobov v jednom systéme [Minsky 1986].

Nestačí samozrejme povoliť uzdu fantázii ohľadne mechanizmov a taktík ktoré použijeme pri tvorbe systému. Dostávame sa do role, v ktorej si nevieme vybrať kedy čo použiť, spomínaná integrácia nie je rozhodne triviálna. Vyžaduje to určitú architektúru, ktorá nám predpisuje určité pravidlá a zabezpečuje, že systém ktorý tvoríme, bude v konečnom dôsledku zodpovedať tomu, čo sme chceli vytvoriť. Jednou z možných takýchto architektúr je subsumpčná architektúra, ktorú navrhol R. Brooks v rokoch 1986-1991 [Brooks 1991]. Sústreďuje v sebe viaceré dôležité myšlienky, z ktorých najdôležitejšou je inkrementálny spôsob implementácie systému, ktorý umožňuje systém tvoriť bez detailného plánu jeho štruktúry a ktorý zaručuje, že implementácia napriek tomu končí želaným výsledkom.

S prácami R. Brooksa som sa prvý krát stretol v roku 1993 prostredníctvom J. Kelemena a jeho prác [Kelemen 1994]. Od roku 1996 pracujem na aplikácii týchto myšlienok v softwarovom inžinierstve a to hlavne v oblasti systémov reálneho času. Pri tomto prístupe používam radikálne iný pojmový slovník než Brooks, jednak preto, že za 15 rokov čo uplynuli od navrhnutia subsumpčnej architektúry sa vyskytli nové pojmy umožňujúce ju lepšie vyjadriť, jednak preto, že táto architektúra bola pevne viazaná na oblasť mobilnej robotiky. Samotné myšlienky si pritom takmer nežiadali úpravu ani doplnenie, len som niektorým prikladal väčší či menší význam, než bolo pôvodne zamýšľané. Výsledkom mojich snáh v tejto oblasti je programový balík `MsAgent`, ktorým je možné implementovať zložité softwarové systémy vyznačujúce sa (i keď nie nevyhnutne) subsumpčnou architektúrou.

`MsAgent` možno použiť na rôzne účely. V tejto práci sa ho budeme snažiť použiť na implementovanie simulátora správania sa kutavky cerceris bupresticida počas zakladania jej potomstva. Na rozdiel od Brooksa, ktorý budoval mobilné roboty, zameriame sa na simuláciu konkrétneho živého systému. Pokúsime sa predstaviť si, že tento hmyz má štruktúru ktorú predpisuje subsumpčná architektúra a túto štruktúru sa pokúsime odhaliť jej metódami. Pritom budeme dbať, aby správanie nášho systému bolo zhodné so správaním, ktoré jeho živý náprotivok vykazuje počas etologických pozorovaní a experimentov.

V práci sa nebudeme zameriavať na softwarovú kvalitu simulátora, kvalitu grafiky a podobne, pôjde nám len o použitú architektúru systému a hlavne o jeho správanie, teda o to, či sa nám podarí aproximovať správanie jeho živého vzoru. Správanie kutaviek počas zakladania potomstva je jedno z najzložitejších, najzaujímavejších a dokonca i najhrôzostrašnejších správaní, ktoré možno v ríši hmyzu pozorovať, takže táto práca by mala dostatočne preveriť biologickú relevantnosť subsumpčnej

architektúry [Lúčny 2001a]. S biologickými motiváciami pracoval (a pracuje) i sám Brooks, ale nie je mi známe, že by sa to niekedy pokúsil preveriť podobným spôsobom, snáď s výnimkou opisu správania sa morského slimáka "Littorina" [Mlichová 1993]. Tento prípad bol však skôr ukážkou, ako sa živé systémy spoliehajú vo svojom fungovaní na dynamické oscilácie prostredia a samotné správanie bolo triviálne.

Moja práca teda pozostáva z dvoch častí. V prvej uvádzam použitú architektúru. Do tejto časti vkladám okrem Brooksových myšlienok vlastnú verziu ich výkladu. Nasleduje časť venovaná pokusu o implementáciu systému simulujúcemu vybraný živý systém a vyhodnotenie tohto pokusu. V závere sa zaoberáme možnosťami ďalších podobných projektov.

2. Subsumpčná architektúra a jej pretavenie

V tejto kapitole sa zameriame na opis prostriedkov a metód použitých pri implementácii vybraného umelého systému. Uvedieme všetky myšlienky, ktoré stoja za subsumpčnou architektúrou [Brooks 1991]. Pritom sa nebudeme zaťažovať uvádzaním jej pôvodného opisu, nakoľko je pevne viazaný na oblasť mobilnej robotiky a (predsa len) aj 15 rokov starý. Vložíme doň vlastnú invenciu a pojmový slovník z oblasti multiagentových systémov. Veríme, že myšlienke to neublíži, ba práve naopak.

V našom nadchádzajúcom výklade sa budeme taktiež musieť zmieniť o veciach ktoré sa netýkajú len umelej inteligencie ale tvorby softwaru vôbec. Budeme sa snažiť obmedziť toto vybočenie z našej témy na minimum. Avšak podľa nášho názoru majú limity na ktoré naráža umelá inteligencia a softwarové inžinierstvo rovnakú povahu (spočívajúcu vo vysokej miere komplexnosti budovaných systémov), preto tieto veci nemožno vynechať.

2.1 Na prahu tvorby systému

Našou úlohou je vytvoriť systém s určitým správaním. Teda taký systém, ktorý je v neustálej interakcii s prostredím či užívateľom a ktorého úlohou je produkovať pre určitý neustávajúci sled vstupov, daných dynamikou prostredia neustávajúcu adekvátnu postupnosť výstupov toto prostredie ovplyvňujúcich.

Aj keď to znie triviálne, treba si povedať, že výsledkom našej práce bude kód, určitá postupnosť inštrukcií, ktorú možno interpretovať na počítači a získať tak ono správanie. Aby sme sa mohli pustiť do práce musíme veriť, že medzi všetkými možnými postupnosťami inštrukcií existuje aspoň jedna taká. Náš problém je nájsť ju. Pritom sa od štruktúry tejto postupnosti dostávame k samotnej povahe procesu jej kódovania. Pri kódovaní okrem technických problémov narážame na problémy epistemologické, dané našimi ľudskými ohraničenými schopnosťami. Jedným takýmto podstatným ohraničením, je schopnosť dokonale pochopiť riešený problém a detailne navrhnúť systém pred tým, než začne prebiehať jeho tvorba (čo je odporúčaný postup v oblasti softwarového inžinierstva). Preto spravidla pri tvorbe väčšieho systému narazíme na to, že sa musíme vrátiť spať k jeho návrhu a modifikovať ho. A tu sa vynára ďalší problém: naša schopnosť udržať komplexnú predstavu o systéme. Spravidla dosť často chybíme pri zohľadnení kde všade v našom kóde musíme zohľadniť príslušnú modifikáciu. Vytváraný kód väčšieho systému sa vplyvom množstva modifikácií programátorom doslova rozpadáva pod rukami, často ešte predtým, než ho prvý krát spustia.

Keby sme mali neohraničenú schopnosť ponímať nami vytváraný systém ako celok v ktorom nám neunikne žiadny detail, a modifikovať ho ľubovoľný počet krát, boli by sme schopní urobiť akýkoľvek systém nasledovným drevorubačským spôsobom: Povedzme, že chceme program, s ktorým nezáväzne konverzujeme prirodzeným jazykom. Vezmeme teda program "hello world!" a začneme ho modifikovať tak, aby sme vždy pridali určitú konverzačnú frázu, samozrejme tak, aby sa nám nepokazili už zaimplementované frázy. Vždy by sme sa chvíľu s aktuálnou verziou systému porozprávali a pokiaľ by sme narazili na nejakú nešikovnú odpoveď, urobili sme ďalšiu modifikáciu. Pokiaľ by sa pridávané frázy významovo prekrývali tak ako je to tomu v bežnom jazyku, celkom zákonite by takouto sériou bezchybných modifikácií v systéme vznikli štruktúry zodpovedajúce pochopeniu ich významu. Stačila by na to trpezlivosť, čo je tiež naše ďalšie ohraničenie.

Keby sme spôsob komunikácie v uvedenom vzorovom príklade obmedzili na písanie cez terminál, mohli by sme taký systém začať tvoriť uvedeným spôsobom hneď teraz, bez trápenia sa so súčasnými

multimediálnymi prostriedkami. Ako by tento náš pokus skončil? Pokiaľ by nám nedošla prvá trpezlivosť, dostali by sme sa časom do situácie, kedy by sa nám nepodarilo zrealizovať určitú modifikáciu bez straty funkčnosti predchádzajúcich vimplementovaných vlastností. Pritom by sme na to prišli zrejme až po niekoľkých ďalších modifikáciách. A celý výsledok by sme mohli zahodiť. Táto metóda naozaj nie je dobrá.

Môžeme však (a v umelej inteligencii obzvlášť) upadnúť do situácie, kde nejasnosť alebo rozsah problému spôsobí, že zlyhajú všetky štandardné metódy. Pre tieto prípady má zmysel hľadať v uvedenej drevorubačskej metóde nejaké zdravé jadro, a nachádzať špecifické podmienky za ktorých by systém prežil bez ujmy veľmi dlhú postupnosť modifikácií i pri našich rozumových ohraničeniach. Ako uvidíme, subsumpčnú architektúru možno považovať za jednu z takýchto špecifikácií. Prejdime preto k úvahám vedúcim k jej jednotlivým detailom.

2.2 Inkrementálny vývoj zdola nahor

Každý kto sa zaoberá tvorbou softwaru vie, že jediná správna metóda implementácie systému je metóda zhora nadol. A je to nielen všeobecne známe, ale je to aj pravda. Kód implementovaného systému totiž pozostáva z určitých častí, ktoré sú buď základné, alebo využívajú iné časti. Pri implementácii metódou zhora nadol, implementujeme najprv časti, ktoré využívajú a až potom tie ktoré sú vyžívané. Tak sa vyhneme riziku, že vyvinieme nejaké časti, ktoré nebudeme vedieť zintegrovať, nakoľko rozhranie pre integráciu určitej časti je navrhnuté skôr než časť samotná. Avšak pokiaľ budujeme veľmi komplexný systém, nie je možné, a to opäť kvôli našim rozumovým ohraničeniam, vytvoriť ho jediným neprerušeným kódovaním. Vždy keď nakódujeme dostatočne veľkú časť, je nevyhnutné podrobiť ju ladeniu, kým pokročíme v kódovaní ďalej. Vývoj teda prebieha inkrementálne, fáza kódovania sa strieda s fázou ladenia. Pokiaľ však postupujeme zhora na nadol, musíme tie časti ktoré sú už využívané, ale ešte nie sú implementované, nahradiť nejakými triviálnymi náhražkami, aby sme mohli k ladeniu pristúpiť. V bežných aplikáciách to spravidla nie je problém. Napríklad v nejakom bankovom systéme môžeme smelo abstrahovať od časti ktorá realizuje zadanie výšky výberu z účtu zákazníkom a nahradiť ju triviálnou časťou ktorá vždy žiada vybrať 150 Sk. Takáto zámena nezníži kvalitu testu funkčnosti nášho systému. Avšak tam, kde takáto časť sprostredkúva zvyšku systému dynamicky sa meniaci priebeh, ako napríklad vizuálny systém v mobilnom robotovi, testovanie s náhražkou takmer nič nevypovedá o funkčnosti doteraz implementovaných častí systému. Nevieme teda napriek inkrementálnej metóde povedať, či postupujeme správnym smerom.

Bolo by možné namietať, že systém možno dekomponovať takým spôsobom, aby sa v ňom časti produkujúce dynamický výstup nevyskytovali. Na poli umelej inteligencie by to však predstavovalo výrazné ochudobnenie. Je jedným z hesiel "Brooksovej školy", že z dynamiky prostredia možno čerpať inteligenciu. Na základe našich skúseností [Lúčny 1994] sa s týmto heslom plne stotožňujeme. Dynamika prostredia dokáže "oživiť" umelý systém a rovnako to platí o výmene údajov medzi jeho jednotlivými časťami.

Ako alternatíva k metóde zhora nadol, ponúka sa nám jej opačná alternatíva zdola nahor. V nej implementujeme najprv využívané a až potom využívajúce časti. Avšak pri tejto metóde nám hrozia ďaleko horšie nebezpečenstvá, než že nám klesne kvalita testovania. Môže sa nám ľahko stať, že vyvinieme zbytočný kód, že navrhneme také rozhranie pre využitie určitej časti, že ju nebudeme vedieť použiť. Ďalej nám hrozí, že si kód rozložíme do častí neadekvátne a nebudeme ho vedieť efektívne integrovať.

V subsumpčnej architektúre sú použité dva triky na obídenie týchto úskalí metódy zdola nahor. Prvý bráni neadekvátnemu rozloženiu kódu do častí a spočíva v netradičnej dekompozícii systému na časti, na tzv. dekompozícii aktivitou. Druhý bráni zavedeniu neadekvátnych rozhraní a je z hľadiska celej architektúry kľúčový. Miesto zvyčajných volaní, kde jedna časť systému využíva druhú vložením určitých argumentov do presne definovaného rozhrania využívanej časti, používa sa odpočúvanie a ovplyvňovanie využívanej časti. Využívajúca časť nevolá využívanú ako jej užívateľ, ale vtiera sa do jej činnosti ako votrelec.

2.3 Dekompozícia aktivitou

Dekompozícia systému aktivitou spočíva v tom, že jeho kód je štruktúrovaný do vrstiev podľa štruktúry ním produkovaného správania a nie na základe štruktúry prostriedkov (funkcií, metód, programátorských techník), ktoré používame pri jeho implementácii. Rozloženie kódu na časti je teda zrealizované tak, že blízko seba ležia kódy podieľajúce sa jednej zložke správania sa systému, hoci by funkčne boli veľmi rôzne. Napríklad v nejakom lingvistickom systéme by boli v jednej časti kódy pracujúce so slovami s koreňom jablk a zatiaľ čo kódy podieľajúce na parsovaní jazyka by boli rozmetané po celom systéme.

Takéto rozloženie síce vedie k neúspornému vyjadrovaniu sa, na druhej strane umožňuje zachytiť v systéme sémantické vzťahy a tým pádom dramaticky znížiť výpočtovú náročnosť reťazenia dát v systéme. V klasickom systéme, kde je jedna nakódovaná procedúra použitá pre všetky prípady jej použitia, je totiž nevyhnutné vyjadriť v každom takomto prípade dáta v rovnakom formáte. Teda pracujeme s nejakým reprezentačným jazykom. To vedie k tomu, že význam dát sa rozplynie pri ich preklade do reprezentačného jazyka. Z dát, ktoré niečo znamenajú sa stanú postupnosti symbolov s ktorými sa manipuluje jednotným spôsobom. Väčšinu času potom systém strávi snahami reťaziť k sebe dáta, ktoré nemajú žiadny sémantický vzťah. Taký systém (ako príklad tu môže poslúžiť napr. STRIPS) sa väčšinou zaoberá úvahami či "let muchy nemá vplyv na výbuch bomby" (ak si vypožičiame modelový príklad problému rámca z [Kelemen a kol. 1992]).

Na druhej strane pokiaľ sa uchýlime k tvore špecifického kódu, môžeme vyjadrovať dáta vždy špecificky vhodným spôsobom a môžeme na ne uplatniť špecificky vhodné metódy. V štruktúre kódu dokážeme zachytiť sémantické vzťahy. Platíme za to zvyšovaním zložitosti kódu. Pri bežných postupoch by nám takýto kód veľmi rýchlo prerástol cez hlavu. Pri dodržaní určitej technológie je však možné toto prekonať, či aspoň výrazne posunúť ohraničenia týmto dané.

Pre návrhára systému ďalej predstavuje dekompozícia aktivitou možnosť pustiť sa do tvorby systému bez predstavy aké algoritmy bude v kóde používať, teda bez predstavy o funkčnej stránke kódu. Miesto toho sa musí vážne zamyslieť na štruktúrou správania sa systému. Musí v nej rozpoznať jednotlivé aktivity systému a ich hierarchiu, teda poradie v akom sa budú implementovať. Každá aktivita musí byť dostatočne jednoduchá, aby ju bolo možné implementovať počas jedného cyklu inkrementálneho vývoja, teda jedným kódovaním a ladením. Každej aktivite tu prislúcha jedna vrstva kódu zložená z viacerých spolupracujúcich častí, ktoré sú vyvinuté naraz. Výhoda tohto prístupu spočíva v tom, že vývojár nemusí mať na začiatku implementácie jasnú predstavu ako realizovať jednotlivé aktivity. V každom sa cykle sa snaží modifikovať kód zodpovedajúci určitej aktivite dovtedy, kým testovanie nepreukáže, že dosiahol želané riešenie. Pritom sa riadi hlavne negatívnymi výsledkami testovania, takže nie je nevyhnutne nutné, aby disponoval dobrou formálnou definíciou príslušnej aktivity.

Vzaté do dôsledkov, je možné takto implementovať i také vágne definované aktivity ako je "inteligentný pohyb". Urobíme nejaký pohyb, a potom sa naň dívame a kladieme si otázku, či je inteligentný. Naša intuícia nám povie, dajme tomu, že nie je a to preto, lebo sme si všimli, že, napríklad, chodí stále na mieste. Modifikujeme teda kód tak, aby sme túto vlastnosť odstránili. Takto postupne odhaľujeme formálnu podstatu intuitívnych pojmov. Postupne zistíme, že inteligentný pohyb zodpovedá určitému súboru podmienok. Od negatívnych výsledkov pri testovaní dostávame sa k pozitívnemu vyjadreniu štruktúry nášho systému. Touto metódou vlastne budujeme prototyp, ktorý nám odhalí formálne zadanie, na základe ktorého by sme boli potom schopní zostrojiť analogický systém ľubovoľnou klasickou metódou.

Samozrejme kľúčovou podmienkou na to, aby bol takýto spôsob vývoja systému možný, je, že modifikáciami, ktorými dosiahneme elimináciu negatívnej, alebo prítomnosť pozitívnej vlastnosti, nesmieme zmeniť význam pôvodných štruktúr. Teda ak niečo pridáme alebo odoberieme v správaní systému, nesmieme pritom odobrať niečo, čo sme tam pridali predtým, ani pridať niečo, čo sme predtým odobrali. A toto je, ako sme už naznačili, vážny problém, ktorý volá po netriviálnom riešení.

2.4 Vtieranie sa do činnosti

Dekompozícia aktivitou umožní pri vývoji zdola nahor dosiahnuť správne rozloženie kódu na časti, avšak potrebujeme ďalší mechanizmus, aby sme prekonali problémy s použitím rozhraní už vyvinutých častí častami, ktoré ešte len vyvíjame. Pri bežnom spôsobe, keď využívajúca časť volá využívanú sa môže stať, že organizácia jej rozhrania značne sťažuje alebo aj znemožňuje jej využitie. Tento problém, typický pre vývoj zdola nahor, je v subsumpčnej architektúre prekonaný tým, že vyvinuté vrstvy neposkytujú vôbec žiadne rozhrania, a využívajúca vrstva využíva využívanú vtieraním sa do jej činnosti. Toto vtieranie môže mať trojakú podobu. Môže ísť o

- odpočúvanie využívanej vrstvy
- potlačenie vplyvu určitej jej časti na iné časti v tejto vrstve (inhibícia)
- nahradenie vplyvu iných častí na určitú jej časť (supresia)

Ambíciou vtierania je zabezpečiť aktivovanie pravých častí v pravý okamih, celkom v zmysle požiadavky M. Minského [Minsky 1986]. Pomocou vtierania je možné vysvetliť u živých (alebo realizovať u umelých) systémov správania, ktoré sa tradične považujú div nie za dôkaz ich schopnosti vedome konať.

2.5 Nepriama komunikácia

Vtieranie musí byť samozrejme umožnené štruktúrou kódu a hlavne spôsobom výmeny údajov medzi jednotlivými časťami kódu v rámci vrstvy. V pôvodnom návrhu subsumpčnej architektúry [Brooks 1991] bolo vtieranie realizované infiltráciou komunikačných vedení, konkrétne odpočúvanie rozdvojením vedenia, inhibícia vložením inhibítora do vedenia a supresia vložením supresora. Pre naše účely sme museli tieto prostriedky zovšeobecniť. Z hľadiska tohto zovšeobecnenia je na celom modeli podstatné to, že kód je rozdelený do modulov, ktoré vzájomne komunikujú posielaním správ a moduly vyššej vrstvy majú prostriedky na to, aby mohli správy prechádzajúce v nižšej vrstve z modulu do modulu odpočuť, zrušiť alebo nahradiť.

Vhodným konceptom pre takéto vzťahy je nepriama komunikácia, kde si každé dve komunikujúce entity posielajú správy prostredníctvom tretej. Túto entitu nazývame prostredie. Komunikovaná správa pritom neobsahuje žiadny identifikátor (adresu) jej príjemcu, iba identifikátor (meno) miesta v prostredí kde je uložená. Prostredie teda slúži ako čierna tabuľa, kde komunikujúce kódové entity môžu zapisovať, čítať a mazať správy na dohodnutej pozícii. Každá takáto pozícia je označená jedinečným menom, ktoré slúži ako jediný identifikátor, ktorý používajú kódové entity na jej referencovanie. Takúto pomenovanú pozíciu nazývame blok (tento názov je vymyslený, ide o výraz ktorý sa samovoľne ujal v kolektíve programátorov, ktorí túto techniku používali v mojom zamestnaní, a napriek pokusom ho už nikto nezmenil).

Na platformách s posielaním správ a multitaskingom môžeme považovať kódové entity i prostredie za samostatné komunikujúce procesy. V takomto modeli je prostredie špeciálny server obsahujúci niekoľko málo služieb, ktoré sprostredkúvajú výmenu správ a kódové entity sú jeho klienti. Z hľadiska softwarových architektúr je zaujímavé, že vzťah klient-server je tu zúžený do špecifickej podoby a normalizovaný, pričom toto zúženie umožňuje definovať veľmi jednoduchú normu. To je unikátne, pokiaľ to porovnáme z bežnými normami vzťahu klient-server ako je napr. CORBA.

V rámci tohto konceptu môžeme vtieranie sa do činnosti zabezpečiť nasledovným spôsobom. Uvažujme dve entity v nižšej vrstve. Prvý z nich zapisuje správu do prostredia a druhý ju číta. Každá entita z vyššej vrstvy môže túto správu čítať tiež, čím je zabezpečené odpočúvanie. Podobne môže správu zmazať a realizovať tak inhibíciu. A môže ju aj prepísať inou správou, čím realizuje supresiu. Týmito operáciami môže vyššia vrstva ovplyvňovať správanie produkované nižšou vrstvou tak, aby boli zložky správania produkované obomi vrstvami koordinované. Akonáhle pominie príčina vtierania, obe správania sa stanú opäť nezávislými.

Hoci týmto spôsobom môžeme zrealizovať zamýšľané vtieranie sa, platíme za to zložitejším realizovaním dátových tokov v systéme. Hlavným problémom je tu zabezpečiť, aby sa príjemca správy dozvedel, že má prečítať príslušný, v prostredí zapísaný, blok. Hoci by mu prostredie mohlo poskytnúť

na to impulz na základe zápisu bloku (tento mechanizmus sa v systémoch klient-server označuje ako trigger), v našej koncepcii sa snažíme zaobísť sa bez toho. Príslušný efekt dosahujeme špeciálnou štruktúrou komunikujúcich entít, ktorá umožňuje pravidelné čítanie bloku bez ohľadu na to, či bol jeho obsah aktualizovaný.

Iným problémom je zabezpečiť, aby príjemca stihol správu prečítať prv než ju zapisovateľ opäť zaktualizuje. Toto sa opäť dá riešiť napr. použitím FIFO štruktúr obsahujúcich všetky neprečítané správy v prostredí. Pre naše potreby sa však tomuto opäť vyhýbame a systém organizujeme radšej tak, aby jeho funkčnosť nebola závislá na jednej stratenej správe. Požadujeme teda, aby takéto straty mali iba prechodný vplyv. Na druhej strane tým výrazne posilňujeme schopnosť systému pracovať v reálnom čase. Jednoducho, čo sa nestihne v systéme spracovať je okamžite zahodené a je to zahodené automaticky bez akéhokoľvek špeciálneho mechanizmu, ktorý by to zabezpečoval. Náš systém má teda veľkú šancu pôsobiť v reálnom dynamickom prostredí.

Ďalším aspektom tohto konceptu je povaha obsahu správy. Dáta uložené v bloku môžu mať povahu:

- buffrov, teda kto ich bude chcieť čítať musí vedieť ako sú uložené a čo znamenajú
- hodnôt s definovaným typom, teda ktokoľvek vie ako sú uložené, ale čitateľ musí vedieť čo znamenajú
- hodnôt vyjadrených v reprezentačnom jazyku (ako sú KIF alebo XML), v takom prípade každý vie ako sú uložené i čo znamenajú

Na rozdiel od podobných modelov [Cianciarini 1999], my preferujeme reprezentovať dáta ako buffre. V našich experimentoch s modelovaním správania hmyzu sme však zhodou okolností vystačili z celými číslami, takže sa dajú považovať i za hodnoty s definovaným typom, nie je to však zámer, ale zhoda okolností vyplývajúca z faktu, že tento spôsob je špeciálnejší prípadov buffrov. Na úplné minimum sme stlačili starosti spojené so zabalením údajov do správy na strane posielateľa a rozbalenie údajov zo správy na strane príjemcu (tzv. marshalling).

V rámci nášho komunikačného konceptu prinášame i jednu novinku voči subsumpčnej architektúre. Tou je už spomínaná možnosť neprebratia správy z bloku ako i porušenie synchrónnosti viacerých zápisov do toho istého bloku. V tomto jedinom sme si dovolili Brooksov model, v ktorom sa žiadna správa nemôže stratiť ani duplikovať a pri viacerých odosielateľoch správy je pevne definované poradie určujúce čia správa bude doručená, zovšeobecniť. Týmto zovšeobecnením sa na jednej strane vnášajú do systému prvky nestability, avšak naše skúsenosti hovoria, že majú prechodný charakter. Na druhej strane majú výrazný vplyv na správanie systému, vnášajú do neho paralelizmus, spočívajúci v súťažení rôznych stratégii. Môžu sa výskytnúť odchýlky od normálu, ktoré môžu vyvrcholiť určitým objavom. Do systému je takto vnášaná živosť.

2.6 Čisto reaktívne agenty

Sústreďme sa teraz na povahu komunikujúcich kódových entít. Už sme spomínali, že pri našom modeli nepriamej komunikácie, príjemca správy sa nemá ako dozvedieť, že obsah ním čítaného bloku sa zmenil (čiže bola do neho zapísaná nová správa). Pokiaľ má teda príjemca pracovať s poslednou verziou správy, neostáva mu iné, než pravidelne (opakovane) príslušný blok čítať (polling). Preto je v našom modeli každý kód organizovaný ako reaktívny agent, teda ako proces neustále konajúci cyklus vnímanie/voľba akcie/akcia (sense/select/act) sledujúc určitý cieľ. V tomto cykle reaktívny agent najprv vníma podnety (teda číta obsahy určitých blokov), na základe nich volí akcie, ktoré má vykonať, aby sledoval stanovený cieľ a následne tieto akcie vykonáva (teda zapisuje zvolené obsahy do zvolených blokov).

Navyše na voľbu akcie sa uvaľujú ďalšie obmedzenia vzhľadom na požiadavku reaktivity. Voľba musí prebiehať na základe jednoduchého algoritmu, ktorý realizuje púhu reakciu. Nie je dovolené modelovať vnútorné náprotivky vnímaných skutočností vyjadrených v nejakom reprezentačnom jazyku, teda agent musí pracovať s podnetom, nie s modelom toho čo za ním stojí. Cieľ musí byť vyjadrený iba implicitne v kóde realizujúcom reakciu. Keďže podnety ani cieľ nesmú byť vyjadrené explicitne v reprezentačnom jazyku, v rámci voľby nie je možné odhadnúť následok zvolených akcií. Tým pádom tu nie je možné plánovanie. Agent sa nerozhoduje pre najlepšiu možnosť z viacerých možností, ale

reaguje [Kelemen 1994]. Nesnaží sa o konanie na základe rozumu, ale módy, robí niečo, lebo sa to v danej chvíli robieva [Minsky 1986].

Ďalšie veľmi dôležité obmedzenie, ktorým mierne sprísňujeme Brooksov model (mierne v tom zmysle, že napriek tomu, že explicitne v ňom táto požiadavka nebola postulovaná, jej porušenie nebolo napr. v robotovi ALLEN nikde využité [Brooks 1991]), je požiadavka čistej reaktivity. Čisto reaktívny agent si nesmie pamätať žiadne vnútorné informácie z predchádzajúceho cyklu. Hovoríme, že nemá vnútorný stav. Táto požiadavka núti programátora kódovať agentov tak, aby boli odkrytí pre zasahovanie z vyšších vrstiev. Pokiaľ by skrývali v sebe enkapsulované informácie, veľmi by to obmedzovalo vtieranie sa do činnosti z vyšších vrstiev. Takto je ale programátor nútený nahradiť vnútornú pamäť zriadením blokov v prostredí, v rámci implementovanej vrstvy výlučne privátnych, ale potenciálne umožňujúcich infiltráciu agentmi z vyššej vrstvy. Nakódovaný agent je takto automaticky predpripravený k tomu, aby sa neskôr vyvinuté agenty vtierali do jeho činnosti. Túto požiadavku je teda možné chápať ako normu na kód agenta, teda syntaktickú záležitosť. Cieľom je automaticky predpripraviť agenta natoľko, aby nemusel byť modifikovaný, keď nastane potreba vtierať sa do jeho činnosti. Keby sme požiadavku na čistú reaktivitu nepostulovali, vtieranie by bolo obmedzené na dátové toky v priestore (z agenta do agenta v určitom čase). S čistou reaktivitou vtieraním môžeme zasiahnuť i dátové toky v čase (v rámci jedného agenta z predchádzajúceho okamihu do nasledujúceho). U Brooksa by tomu zodpovedalo vyvedenie vedenia z výstupných registrov do časti vstupných registrov v rámci modulu von z modulu tak, aby sa do vedenia dali vkladať supresory a inhibítory a aby sa dalo odpočuť. V princípe však túto možnosť v rámci svojich prác veľmi nepotreboval, preto niet divu, že ju nevyužil. Pri porovnaní oboch modelov, možno takmer so sto percentnou účinnosťou ztotožniť našich čisto reaktívnych agentov s Boooksovými modulmi (i keď výnimky existujú, napr. modul Status v ALLENovi je len akýmsi kontajnerom signálov, teda zodpovedá časti nášho prostredia, zaujímavé však je, že práve tento modul je v našich očiach podozrivý, že vznikol modifikáciou odladenej vrstvy, teda proti princípom subsumpčnej architektúry).

2.7 Transducery

Bloky v našom modeli neslúžia len na sprostredkovanie prenosu informácie medzi dvomi agentmi, alebo v čase. Potenciálne slúžia taktiež na prepojenie na periférne zariadenia (senzory a aktuátory). Obsah určitých blokov sa sám mení podľa signálov prichádzajúcich zo senzorov. Naopak zmena obsahu určitých blokov je interpretovaná vysielaním signálov na aktuátory. Takéto špecifické bloky nazývame transducery.

V rámci tejto práce nepracujeme s fyzickými senzormi a aktuátormi, ale simulujeme ich, prostredníctvom softwarového simulátora. Napriek tomu, že v blokoch, ktoré sprostredkúvajú styk so simulátorom, neprebieha žiadny prevod signálu, budeme tieto tiež nazývať transducery.

2.8 Zhrnutie

V prechádzajúcich kapitolách sme predviedli logickú úvahu, ktorá nás priviedla k nášmu prerozprávaniu Brooksovho modelu subsumpčnej architektúry, čo nám umožnilo aplikovať Brooksove myšlienky z oblasti mobilnej robotiky na vývoj inteligentného software a software vôbec. Naša verzia subsumpčnej architektúry teda pozostáva z nasledovných prvkov:

1. Vyvíjaný systém je organizovaný ako multiagentový systém pozostávajúci z čisto reaktívnych agentov. Títo neustále vykonávajú sense/select/act cyklus sledujúc implicitne zadaný cieľ.
2. Tieto agenty komunikujú nepriamo, skrz prostredie. Toto prostredie poskytuje agentom služby pre čítanie, zápis a vymazanie správ. Identifikačným kľúčom správ je pritom meno tzv. bloku, v ktorom sú uložené. Prepojenie na fyzické senzory a aktuátory je realizované pomocou špeciálnych blokov, tzv. transducerov.
3. Takýto systém je vyvíjaný inkrementálne zdola nahor, na základe úvodnej dekompozície aktivitou, pri ktorej analyzujeme správanie, ktoré by mal systém produkovať a dekomponujeme ho na hierarchicky usporiadané vrstvy. V každom implementačnom kroku je do systému pridaných niekoľko

agentov tvoriacich jednu vrstvu a realizujúcich jednu aktivitu. Táto je ladená a modifikovaná až kým testovanie nepreukáže, že želané správanie bolo dosiahnuté, pričom prv implementované aktivity neboli narušené.

4. Hierarchicky vyššie vrstvy kooperujú svoju činnosť s hierarchicky nižšími prostredníctvom vtierania sa do ich činnosti. Toto vtieranie je realizované na základe toho, že agenty vo vyššej vrstve manipulujú s blokmi, ktoré boli zavedené pôvodne pre agentov v nižšej vrstve.

Rôzne modifikácie tejto architektúry môžu spočívať

- v rozšírení služieb prostredia
  - urýchlenie dátových tokov prostredníctvom triggerov
  - synchronizácia zápisov pomocou priorít
  - zabránenie možnosti straty správy pre určité bloky prostredníctvom FIFO štruktúr
- vo vzdaní sa požiadavky na čistú reaktivitu

So všetkými týmito variantami máme praktické skúsenosti na základe ktorých sme ich pre potreby tejto práce odmietli. Nevylučujeme však ich použitie v iných aplikáciách.

Výhody architektúry spočívajú

- v schopnosti systému podstúpiť dlhú sériu modifikácií
- v odolnosti systému voči prechodovým javom, chybám, strate reálneho času (tzv. robustnosť)
- v možnosti vyvíjať systém, pre ktorý nevieme definovať formálne precízne požiadavky
- vo vhodnosti pre simulovanie jednoduchých živých systémov
- v normalizovaní komunikačných rozhraní v systéme
- v rozbití systému do veľkého množstva relatívne jednoduchých procesov
- vo vysokej miere konfigurovateľnosti systému, v možnosti meniť za behu jeho parametre bez použitia propagačných mechanizmov
- v rovnomernom rozložení nárokov na výpočtovú kapacitu
- v podpore emergencie, t.j. výskytu stavov, keď systém nadobudne (želanú) vlastnosť, ktorú doň vývojár explicitne nevložil (hoci sa objavila celkom zákonite)
- v podpore lokálneho testovania jednotlivých agentov na základe simulovania zápisov a monitorovania blokov s ktorými pracuje

Nevýhody architektúry spočívajú

- v drastickom obmedzení prostriedkov ktoré má programátor k dispozícii
- v atypickom prístupe k vývoju software
- v strate enkapsulácie a znovuvyužitia na základe podobnosti
- v náraste množstva kódu
- v náraste nárokov na výpočtovú a komunikačnú kapacitu (v prípade analogickej realizácie klasickým spôsobom, ak by bola možná)
- v absencií podpory učenia (v porovnaní napr. s neurónovými sieťami)

Na záver zhrnutia uvedieme ešte heslá, ktoré sa snaží uvedená architektúra naplniť:

- Minsky: "Inteligencia systému spočíva v aktivovaní správnych agentov v správny čas"
- Brooks: "Systém by mat dokázať čerpať inteligenciu z dynamiky prostredia"
- Brooks: "Najlepšou reprezentáciou sveta je svet sám"

3. Simulácia zakladania potomstva u kutavky cerceris bupresticida

Na rozdiel od Brooksa, ktorý sa po skonštruovaní prvých mobilných robotov začal od roku 1993 orientovať na konštruovanie humanoidných robotov, t.j. snažil sa ísť cestou konštrukcie ďaleko zložitejších umelých systémov, my sa pokúsime o zameranie sa na tvorbu jednoduchších systémov, ktoré čo najpresnejšie aproximujú určité reálne živé systémy. Na našu architektúru sa teda nedívame len ako na technický prostriedok na vývoj nejakých zaujímavých umelých systémov, ale aj ako na prostriedok na opis fungovania systémov živých.

Tento pohľad nás oprávňuje zrieť sa niektorých prostriedkov, ktoré pre programátora predstavujú nepochybnú výhodu, na základe toho, že ich považujeme za málo biologicky relevantné. Ide tu najmä o triggery, teda o možnosť zobudiť agenta pre vykonanie ďalšej otočky svojho sense/select/act cyklu na základe zmeny obsahu určitého bloku v prostredí. Keby sme programovali nejakú bežnú úlohu (napr. nejaký monitorovací alebo riadiaci systém), boli by triggre veľmi často používané. Pre potreby tejto práce ich však nepoužijeme, hoci to naše implementačné prostriedky umožňujú.

Je potrebné poznamenať, že i Brooks uvažuje biologickú relevanciu subsumpčnej architektúry. V jeho prípade však ide o paralelu s biologickou evolúciou, ktorú prirovnáva k inkrementálnemu vývoju zdola nahor. Programátor tu hrá rolu faktorov spôsobujúcich prírodný výber vo fáze testovania a rolu faktorov spôsobujúcich vznik odchýlok vo fáze kódovania [Brooks 1997]. Vtieranie sa do činnosti zodpovedá interakciám na chemickej úrovni organizmu (mimochodom naša verzia vtierania bez presne stanovených priorít a s istou pravdepodobnosťou je zjavne biologicky relevantnejšia než Brooksova).

My však prechádzame od paralely k aproximácii. To nám umožňuje definovať kritérium úspešnosti nášho snaženia, náš umelý výtvor môžeme porovnávať so živým systémom. Približujeme sa teda k ideálu Turingovho testu [Ferko - Kalaš - Kelemen 1990], chýbajú už len prostriedky na zabránenie možnosti rozlíšiť umelý a živý systém na základe vonkajších znakov. Keby sme mali takú prepracovanú grafiku, že by sme vedeli správanie generované systémom v simulátore nahrať na video a toto by nebolo z hľadiska kvality obrazu rozlíšiteľné od nahrávky kamerou v prírode, mohli by sme entomologickým expertom poskytnúť toto video a sledovať, či odhalia, že nevzniklo nafilmovaním živého systému.

3.1 Výber správania na aproximáciu umelým systémom

Na aproximovanie umelým systémom sme si v tejto práci vybrali zakladanie potomstva u kutavky cerceris bupresticida. Inšpiroval nás k tomu [Gál 2000]. Ťažko by sme našli v ríši hmyzu iné tak uzavreté a pritom tak zložité správanie. Náš výber má tú výhodu, že je veľmi obľúbeným medzi kognitívnymi vedcami. Ako prvý ho opísal jeden z prvých etológov J. Fabre, ešte v 19. storočí [Chalifman 1983]. Medzi kognitívnymi vedcami ho propagoval hlavne R. Dawkins [Dawkins 1996]. Nevýhodou nášho výberu na druhej strane je, že prechodom z oblasti entomológie do oblasti kognitívnych vied sa na jeho výklad nalepilo niekoľko interpretácií, ktoré by samotní entomológovia nikdy neprijali. Medzi kutavkami panuje značná diverzita, každú druh loví inú korisť, má trochu iný postup pri zakladaní potomstva a rozdiely možno pozorovať dokonca v rámci rôznych rodín toho istého druhu. Budeme sa teda snažiť vyhnúť akýmkoľvek zovšeobecneniam. Preto v nasledujúcom texte pod kutavkou budeme rozumieť druh cerceris bupresticida a tie jeho rodiny, ktoré pri pozorovaní uvedené správanie vykazujú. Takže pozrime sa na to správanie, ktoré by sme chceli aproximovať:

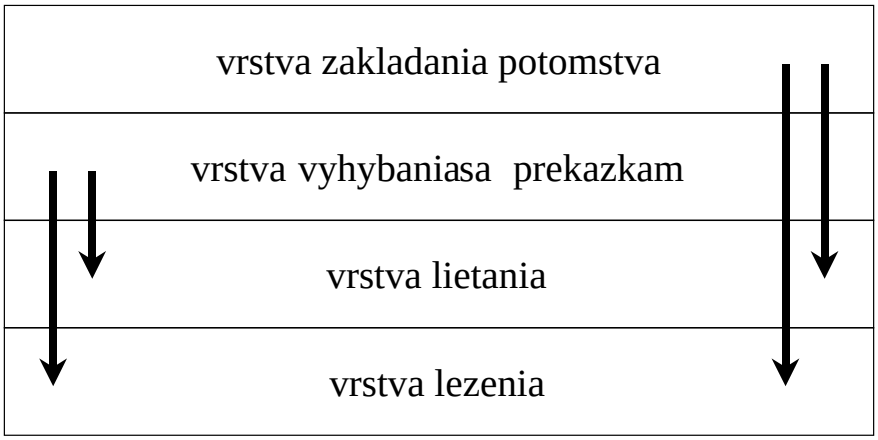
Kutavka je samotárska osa známa najmä pre hrôzostrašný spôsob akým zabezpečuje pre svoje larvy dostatok čerstvej potravy: žihadlom pichne svoju korisť (napr. samičku sedlovky) tak, že ochrnie a odtiahne ju do podzemnej komôrky, kde na ňu nakladie vajíčko. Z toho sa vyliahne kutavčia larva,

ktorá potom paralyzovanú obeť zaživa zje, pričom jej to trvá niekoľko mesiacov. Kutavka najprv vyhrabe komôrku, potom uloví obeť, položí si ju pred komôrku, vlezie do komôrky skontrolovať ju, potom do nej vtiahne obeť, nakladie 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. 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.

Nedá sa povedať, že by kutavka konala neúčelne, nakoľko kontrola komôrky je pre ňu dôležitá. Keby totiž do komôrky vliezol medzitým niektorý z hmyzích dravcov, mohol by jej pokaziť celú operáciu zakladania potomstva, ba dokonca ohroziť samotnú kutavku, ktorá lezie do komôrky zadočkom napred. Nápadná je však absencia učenia, nás ľudí by minimálne omrzelo po čase vykonávať rovnakú činnosť bez zaznamenania pokroku, hoci by sme si boli vedomí, že kontrolu pre istotu treba vykonať každý jeden raz. Neprítomnosť učenia v tomto procese je na druhej strane vítaným faktorom a to ako z hľadiska schopností našej architektúry, tak z hľadiska vyhodnotenia, pri ktorom je opakovaný pokus nezávislý od predchádzajúcich.

3.2 Analýza aproximovaného správania

Podľa pravidiel subsumpčnej architektúry musíme na začiatku stanoviť jednotlivé aktivity, ktoré definujú hierarchiu a poradie implementácie vrstiev systému. Pri tejto analýze sme sa sústredili na to, aby sme minimalizovali aktivity na tie, ktoré hrajú rolu pri zakladaní potomstva. Pri zakladaní potomstva je v postate jediným dôležitým, vzhľadom na potomstvo nezávislým, faktorom pohyb. Ten možno štruktúrovať na lezenie, lietanie a vyhýbanie sa prekážkam, ktoré ich využíva a riadi. Ďalším kandidátom na vrstvu by bolo získanie potravy, avšak kutavka pre zakladanie potomstva získava potravu diametrálne odlišným spôsobom, než pre vlastnú potrebu. Pre potomstvo loví určitý špecifický druh chrobákov, múch alebo ich lariev, zatiaľ čo sama sa živí predovšetkým nektárom z kvetov a plodmi. Pritom sama je niekedy obeťou niektorých chrobákov a v prípade kutaviek špecializujúcich sa nosáčikov, je obeťou dokonca tých istých chrobákov, ktorých nekompromisne porazí v čase zakladania potomstva, ako preukázal jeden z najvynaliezavejších etológov S. I. Malyšev na začiatku 20. storočia [Chalifman 1983]. Z toho usudzujeme, že by sme nemali používať väzbu medzi zakladaním potomstva a získavaním potravy pre samotnú kutavku, a toto rozhodnutie nám navyše umožní úplne vypustiť výživu kutavky z implementácie.



Obrázok 3.1: Štruktúra aproximovaného správania - dekompozícia aktivitou

V systéme budeme mať teda štyri vrstvy ako ukazuje obrázok 3.1. Vrstvy lezenia a lietania sú nezávislé, vrstvy zakladania potomstva a vyhýbania sa prekážkam sa vtierajú do ich činnosti.

3.3 Prostriedky na implementáciu

Vzhľadom na naše dobré skúsenosti s operačným systémom QNX4, zvolili sme tento OS za našu implementačnú platformu. QNX4 nám okrem toho, že ho dôverne poznáme prináša oproti iným platformám nasledovné výhody:

- prácu v reálnom čase, kvalitné časovanie procesov
- uniformné posielanie správ (message passing)
- dokonalú synchronizáciu procesov (t.j. zabezpečenie, aby priebeh cez dva zvolené body v kóde dvoch procesov nastal v rovnaký okamih)

Pod touto platformou použijeme proprietárny balík `MsAgent` od MicroStep-MIS, ktorý bol vyvinutý pod mojím vedením v rokoch 1996-1998, a aktualizovaný v roku 2001. Tento balík pozostáva z runtime, ktorého hlavnou časťou je proces `MsEnv`, realizujúci prostredie a toolkitu, ktorého hlavnou časťou je klientská knižnica pre agentov `MsAgent.lib`. Prostredie i knižnica sú realizované vo Watcom C 10.6. Toto bude náš implementačný jazyk i pre simulátor a jednotlivých agentov.

Prostredie `MsEnv` je špecifický server, ktorý realizuje nasledovné služby:

- zápis buffra do bloku podľa mena
- čítanie buffra z bloku podľa mena
- vymazanie obsahu bloku podľa mena
- zavedenie absolútneho timera
- zavedenie triggera na skupinu blokov podľa mena (v tejto práci nepoužité)
- masové čítanie blokov podľa triggera (nepoužité)
- masové čítanie blokov podľa masky na meno (nepoužité)

Na strane agenta sprostredkúva marshalling klientská knižnica `MsAgent.lib`. Obsahuje nasledovné funkcie:

```
// MsAgent.h

//*************************************************************************
// planovane citanie z prostredia
void MsAgentRead (
    char *blocks,    // definicia blokov
    void *data,      // buffer na prijem dat
    int size,        // velkost buffra
    int *success     // vrati uspesnost 1/0, moze byt NULL
);

//*************************************************************************
// planovane zapisy do prostredia
void MsAgentWrite (
    char *blocks,    // definicia blokov
    void *data,      // buffer s datami
    int size,        // velkost buffra
    int *success     // vrati uspesnost 1/0, moze byt NULL
);

//*************************************************************************
// planovane zrusenie bloku z prostredia
void MsAgentDel (
    char *blocks     // definicia blokov
```

```
);

//***************************************************************************
// planovanie nastavenia triggeru v prostredi na urcite bloky
// - od okamihu nastavenia vzdy pri zmene niektoreho z definovanych
//   blokov sa agentovi triggeruje proxy
// - pokial sa jednym zapisom ineho agenta meni viacero z tychto
//   blokov, generuje sa len jedno proxy
// - pokial je psec, pnsec, sec a nsec nulove, proxy sa triggeruje v
//   okamihu zapisu (v pripade postupneho zapisania blokov sa teda proxy
//   triggeruje zakazdym)
// - pokial je nastavena perioda psec a pnsec, proxy sa triggeruje az
//   ked sa dosiahne tato perioda, i ked sa pocas tejto doby uskutocni
//   viacej zapisov urobi sa iba jeden trigger
// - pokial je nastavene oneskorenie sec, nsec triggeruje sa az po uplynuti
//   danej doby, i ked sa pocas tejto doby uskutocni viacej zapisov
//   urobi sa iba jeden trigger
// - pokial su nastavene perioda i oneskorenie, proxy sa triggeruje
//   az po vyprsani oneskorenia po prvom dosiahnuti periody
// - pokial sa ako blocks uvedie NULL, agent si zavadza specialny timer,
//   ktory sa vsak trigerruje proxy az po tom, co sa v danom okamihu
//   dostane ku slovu prostredie. Pritom casovanie je absolutne,
//   nezalezi od okamihu kedy sa on poziada ale, iba od periody,
//   pripadne oneskorenia. Proxy opakovane prichadzaju, ked vyprsi
//   oneskorenie ratane od okamihu kedy je cas delitelny periodou.
//   pokial je perioda nulova, tak pride iba jeden impulz po vyprsani
//   oneskorenia.
// - po prijati triggernuteho proxy sa odporuca vycucat toto proxy
//   cez while (Creceive(proxy,0,0)!=-1);
void MsAgentInsTrigger (
    char *blocks,    // definicia blokov alebo NULL
    pid_t proxy,     // proxy
    int psec,        // perioda
    int pnsec,
    int sec,         // oneskorenie
    int nsec
);

//***************************************************************************
// planovanie zrusenia triggeru v prostredi na urcite bloky
// - pokial je definicia blokov NULL, zrusi sa kazdy trigger na dane proxy
void MsAgentDelTrigger (
    char *blocks,    // definicia blokov alebo NULL
    pid_t proxy      // proxy
);

//***************************************************************************
// vykonanie naplanovaneho
// vrati 0 ak vsetko prebehlo v poriadku, inak -1
int MsAgent (
    char *envname,   // meno prostredia (pre locate)
    time_t *utc      // cas, moze byt NULL
);
```

Treba upozorniť na mätúcosť názvu funkcie `MsAgentInsTrigger()`, ktorá slúži i na zavedenie timera, keď sa zadá ako názov bloku `NULL`. Čiže i napriek tomu, že používame túto funkciu, nepoužívame triggery, ale iba timery.

Typický agent má teda nasledovný tvar:

```
void main (void)
{
    // zavedenie timera s urcitou frekvenciou
    pid_t proxy = qnx_proxy_attach(0,0,0,-1);
    MsAgentInsTrigger(NULL,proxy,...);
    MsAgent(...,NULL);
    for (;;) {
        Receive(proxy,0,0);
        // vnimanie (sense)
        MsAgentRead(...);
        MsAgentRead(...);
        ...
        MsAgentRead(...);
        MsAgent(...,NULL);
        // volba akcie (select)
        ...
        // vykonavanie akcii (act)
        MsAgentWrite(...);
        MsAgentWrite(...);
        ...
        MsAgentWrite(...);
        MsAgent(...,NULL);
    }
}
```

Náš prístup nám teda na rozdiel na od ostatných umožňuje definovať agenta na základe tvaru jeho kódu:

Agent je proces, ktorý neustále (opakovane) vníma svoje prostredie, na základe toho volí akcie, ktoré má v tomto prostredí vykonať, aby dosiahol stanovený cieľ a následne tieto akcie vykonáva. [Lúčny 1997] [Doran 1992]

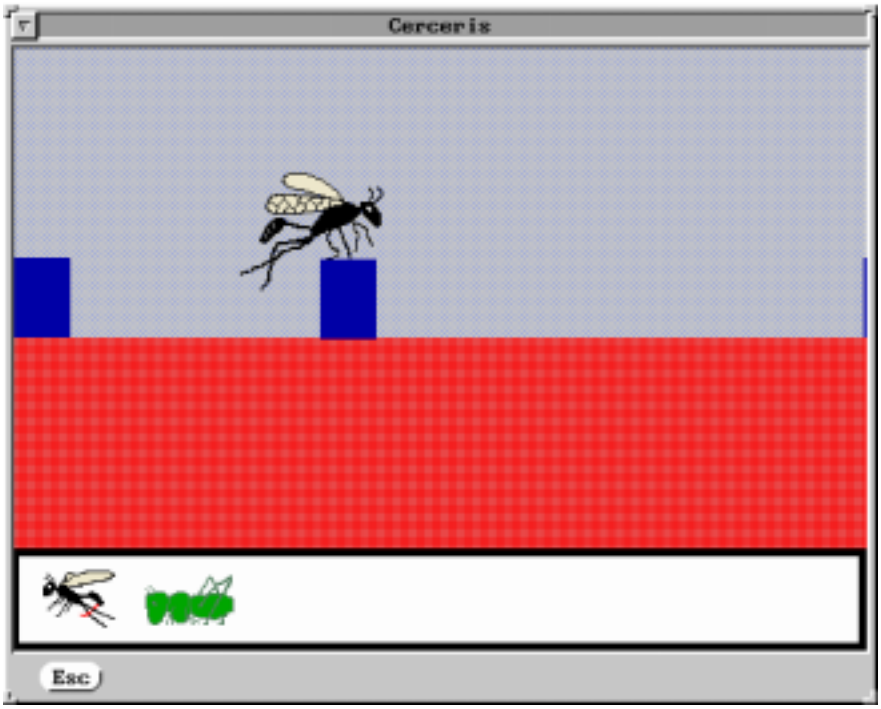
Od povahy voľby akcie potom záleží, či ide o reaktívneho agenta.

3.4 Simulátor vonkajšieho prostredia

Simulátor sme založili na grafickom užívateľskom rozhraní QNX Windows. Realizovali sme ho vo veľmi jednoduchej podobe, nakoľko sme chceli väčšinu energie venovať jednotlivým agentom. Simulujeme uzavretý svet, v ktorom máme vzdušný priestor a priestor pod zemou. Na zemi ležia okrajové prekážky a jedna prekonateľná prekážka. Je tam ďalej jedno miesto "vhodné" (tak ho vníma aproximovaná kutavka) pre kopanie komôrky. Okrem toho sú tam už len dve drag–and–drop ikony s ktorými môže pohybovať užívateľ a simulovať tak objekty, ktoré sú svojou povahou aktívne, ale neaproximujeme ich agentmi. Sú to sedlovka (potrava pre kutavčiu larvu) a malý kutavčí samček. Základný pohľad na okno simulátora poskytuje obrázok 3.2.

Vzdialenosti v simulovanom svete sa merajú v tipsoch, čo sú jednotky miery z GUI QNX Windows. Jeden pixel na obrazovke reprezentuje útvar o veľkosti 11x11 tipsov. Celý svet je 7040 tipsov dlhý, 2310 tipsov vysoký a 1848 tipsov hlboký (v rámci jednej zvislej dimenzie dlhej 4158 tipsov). Dĺžka prekážky a dĺžka miesta na kopanie komôrky je 209 tipsov. Svet je dvojrozmerný, teda orientácia kutavky je dvojhodnotová: doprava alebo doľava.

Čas sa meria v sekundách, môže ísť pritom aj o necelé hodnoty, nakoľko základné časovanie prostredia je 250 ms.



Obrázok 3.2: Simulátor senzorov, aktuátorov a vonkajšieho prostredia

Jediný nečistý prvok, ktorý sme pri simulácii použili je tzv. "pohárový efekt". Simulovaný priestor je uzavretý, čo nám umožňuje nezohľadňovať navigáciu kutavky priestore. To je v rozpore so skutočnosťou, lebo kutavky sa dokázateľne riadia pri navigácii podľa slnka a objektov na zemi. (Ich navigáciu skúmal prvý opäť S. I. Malyšev, ktorý presádzal v ich revíri borovice a pozoroval aký výrazný vplyv to malo na ich, inak dosť stereotypný, let [Chalifman 1983].

3.5 Transducery

Simulátor poskytuje v pravidelných časových intervaloch (1s) kutavke nasledovné údaje prostredníctvom týchto (senzorom zodpovedajúcich) transducerov (všetko sú to celé čísla):

- `unava` 0–40, čím vyššia hodnota tým väčšia únava. Hodnota sa zvyšuje vykonávaním každej činnosti, pri ktorej sa koná práca. Zmenšuje sa na základe oddychu, teda zablokovaním konania akejkoľvek práce na istý čas.
- `naraz` 0/1, detekuje náraz na prekážku
- `sedlovka` 0/1–650, vzdialenosť sedlovky, ak je v dohľade
- `komorka` 0/1–2000, vzdialenosť od miesta na kopanie komôrky, ak je v dohľade
- `hlbka` 0–24, detekuje hĺbku zarytia pod zem
- `samcek` 0/1, detekuje blízkosť samčeka

Vnímanie je podstatne sťažené malými toleranciami. Kutavka vidí sedlovku na 650 tipsov, cíti prítomnosť samčeka na 650 tipsov (páriť sa s ním dokáže na vzdialenosť 200 tipsov) a miesto vhodné pre komôrku na 2000 tipsov.

Simulátor umožňuje ďalej kutavke vykonávať akcie prostedníctvom nasledovných, aktuátorom zodpovedajúcim transducerov (opäť sú to všetko celé čísla):

- `hybsa` 1/0, príkaz na vykonanie jedného kutavčieho kroku, vrátane posunu počas letu
- `otocsa` 1/0, príkaz na otočenie sa
- `oddych` 0-t, blokovanie všetkých činností na určitý čas v sekundách za účelom zníženia únavy

- `lietaj` 0-t, lietaj určitý čas v sekundách
- `vyska` 0–5, určuje výšku letu nad zemou
- `zihadlo` 1/0, bodni žihadlom
- `kop` 1/0, kop do zeme, spôsobuje zvýšenie hĺbky výkopu
- `spustisa` 1/0, spúšťanie sa do výkopu
- `vylez` 1/0, vyliezanie z výkopu
- `uchop` 1/0, chyť (spravidla sedlovku), ak je čo
- `pusti` 1/0, pusti (spravidla sedlovku), ak niečo držíš. Ak je tento povel aplikovaný počas letu, simulátor zariadi, že pustený objekt spadne na zem.
- `suloz` 1/0, pár sa, ak je samček na tebe
- `zapecat` 1/0, zapečať komôrku vrstvou hliny

Naša voľba transducerov nám samozrejme výrazne zjednodušuje situáciu. V reálnom svete by sme museli napríklad na kopanie produkovať povely pre zložité aktuátory s mnohými stupňami voľnosti. Hoci práve na ovládanie takýchto zložitých aktuátorov bola subsumpčná architektúra vymyslená, nezvládli by sme ich odsimulovať. Pri stelesnenom riešení by to bolo opačné: nezvládli by sme zrealizovať napríklad povel kop nejakým samostatným algoritmom, ktorý by ho premieňal na motorické akcie kutavčích hryzadiel a nôh. Takže v žiadnom prípade nemožno hovoriť, že by sa naše riešenie v simulátore dalo ztelesniť v robotickom systéme. Možno však smelo povedať, že sa robotický systém dá vybudovať rovnakou metódou.

K týmto transducerom sme v skutočnosti dospeli počas vývoja jednotlivých vrstiev, takže niektoré z nich sú "vývojovo staršie" a iné "vývojovo mladšie".

3.6 Význam náhody

Pre voľbu akcie budeme často využívať generátor náhody, dávajúci nahodné celé čísla v rozsahu 0-100. Tento generátor predstavuje jednak širokospektrálny vplyv vonkajšieho prostredia na živý systém, ktorý simulátorom nevieme realizovať, jednak stochastickú povahu biologických a chemických procesov v živom systéme. Z hľadiska programátora tým ďalej šetríme pamäťové miesta, ktorými by sme náhodnosť generovali a tým podporujeme čistú reaktivitu. (Napríklad: aby kutavka nekonala dlhšiu priamu cestu než 100 krokov, môžeme tieto kroky odpočítať a po dosiahnutí limitu dať príkaz na otočenie. My sa ale radšej v každom kroku otočíme s pravdepodobnosťou $\frac{1}{100}$. Výhody tejto taktiky sme potvrdili už v [Lúčny 1994].)

3.7 Implementácia vrstvy lezenia

Lezenie spočíva v tom, že sa dáva opakovaný príkaz `hybsa`, ktorý na základe narastajúcej únavy, môže byť vystriedaný príkazom `oddych` na oddych. Ďalej, aby kutavka neliezla do nekonečna jedným smerom, zvolí si niekedy náhodne otočenie a to tým skôr, čím je viac unavená. Toto nám zhodou okolností zrealizuje i vyhýbanie sa prekážkam v tej miere, že keď kutavka dostatočne dlho búši do prekážky, takmer s istotou sa otočí potom, čo sa búšením dostatočne unaví. (Pri istej dávke fantázie tu môžeme hovoriť o emergencii, samozrejme niet na nej nič mimoriadneho, avšak po dostatočne dlhom zamyslení nemôže byť nič mimoriadneho na žiadnom prípade emergencie.) Ostatné je vecou ladenia konštánt pri opakovanom testovaní. (Tieto konštanty, žiaľ, silne závisia od konkrétnych priestorových a časových parametrov simulátora, ale to je aspoň z hľadiska biologickej relevantnosti v poriadku.)

Vrstvu realizujeme jediným agentom `lezenie`.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
MsAgentInsTrigger(NULL,proxy,2,0,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int unava = 0;
    int oddych = 0;
```

```
    int neotacajsa = 0;
    int otocsa = 0;
    int hybsa = 1;
    Receive(proxy,0,0);
    MsAgentRead("unava",&unava,sizeof(int),NULL);
    MsAgentRead("oddych",&oddych,sizeof(int),NULL);
    MsAgentRead("neotacajsa",&hybsa,sizeof(int),NULL);
    MsAgent("KUT",NULL);
    if (unava > 30) oddych = 1;
    if (unava <= 10) oddych = 0;
    if (oddych) {
        hybsa = 0;
        unava -= 7;
    }
    else {
        hybsa = 1;
        unava++;
    }
    if (neotacajsa) neotacajsa--;
    if (hybsa && !neotacajsa) {
        int trpezlivost = (40-unava) / 4;
        if (trpezlivost <= 0) trpezlivost = 1;
        if (rnd() < 80 / trpezlivost) {
            otocsa = 1;
            neotacajsa = 7;
        }
    }
    if (otocsa) MsAgentWrite("otocsa",&otocsa,sizeof(int),NULL);
    MsAgentWrite("hybsa",&hybsa,sizeof(int),NULL);
    MsAgentWrite("unava",&unava,sizeof(int),NULL);
    MsAgentWrite("oddych",&oddych,sizeof(int),NULL);
    MsAgentWrite("neotacajsa",&hybsa,sizeof(int),NULL);
    MsAgent("KUT",NULL);
}
```

3.8 Implementácia vrstvy lietania

Z času na čas kutavka si kutavka zvolí lietanie na určitý čas závislý od miery jej únavy, čím je čerstvejšia, tým dlhšie sa rozhodne zalietať si. Výška letu sa stanovuje na základe pravdepodobnosti vzhľadom na čas letu. Vo začiatočnej a koncovej fáze je výška spravidla malá, v strednej fáze môže dosahovať i maximálnu hodnotu.

Vrstvu realizujeme jediným agentom `lietanie`. Pritom lietanie zdieľa prostredníctvom simulovaných aktuátorov i stratégiu lezenia.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
MsAgentInsTrigger(NULL,proxy,1,0,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int oddych = 0;
    int lietaj = 0;
    int vyska = 0;
    Receive(proxy,0,0);
    MsAgentRead("lietaj",&lietaj,sizeof(int),NULL);
    MsAgentRead("vyska",&vyska,sizeof(int),NULL);
```

```
    MsAgentRead("oddych",&oddych,sizeof(int),NULL);
    MsAgent("KUT",NULL);
    if (!oddych) {
        if (lietaj == 0) {
            if (rnd() <=  5) {
                lietaj = 10 + rnd() / 6;
                vyska = 1;
            }
        }
        else {
            if (vyska == 0) vyska = 1;
            lietaj--;
            if (lietaj == 0) vyska = 0;
            else if (lietaj / vyska >= 4) vyska++;
            else if (lietaj / vyska <= 2) vyska--;
            if (lietaj) {
                if (vyska < 1) vyska = 1;
                if (vyska > 5) vyska = 5;
            }
        }
    }
    MsAgentWrite("lietaj",&lietaj,sizeof(int),NULL);
    MsAgentWrite("vyska",&vyska,sizeof(int),NULL);
    MsAgent("KUT",NULL);
}
```

3.9 Implementácia vrstvy vyhýbania sa prekážkam

S predchádzajúcimi dvomi vrstvami máme kutavku, ktorá lezie, sem-tam lieta a sem-tam sa otočí. Pokiaľ však narazí na prekážku, naráža do nej, pokiaľ sa jej nepodarí náhodne sa otočiť, čo nie je príliš často, aj keď pravdepodobnosť otočenia vzrastá s dĺžkou trvania narážania prostredníctvom zvyšujúcej sa únavy. Kutavka preto na prekážke nepôsobí veľmi múdro a tomu by mu mala zabrániť vrstva vyhýbania sa prekážkam.

Pri realizácii tejto vrstvy použijeme vtieranie sa do činnosti ako lezenia, tak lietania. Na základe detekcie nárazu zo simulátora, zapíšeme príkaz na otočenie alebo lietanie, tak, ako by si to zaželala príslušná vrstva lezenia alebo lietania. Tie to robia len občas a vrstva vyhýbania sa prekážkam simuluje, že taký moment práve nastal.

Ostatné je vecou ladenia konštánt. Vrstvu realizujeme jediným agentom `prekazky`.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
MsAgentInsTrigger(NULL,proxy,0,500000000,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int naraz = 0;
    int nula = 0;
    int otocsa = 0;
    int lietaj = 0;
    Receive(proxy,0,0);
    MsAgentRead("naraz",&naraz,sizeof(int),NULL);
    MsAgentWrite("naraz",&nula,sizeof(int),NULL);
    MsAgent("KUT",NULL);
    if (naraz) {
        if (rnd() < 20) lietaj = 5 + rnd()/8;
        else if (rnd() < 70) otocsa = 1;
```

```
        }
        if (otocsa || lietaj) {
            if (lietaj) MsAgentWrite("lietaj",&lietaj,sizeof(int),NULL);
            if (otocsa) MsAgentWrite("otocsa",&otocsa,sizeof(int),NULL);
            MsAgent("KUT",NULL);
        }
    }
```

3.10 Implementácia vrstvy zakladania potomstva

Dostávame sa k implementácii veľmi zložitej vrstvy. Túto vrstvu samotnú sme tiež implementovali inkrementálne postupným vyvinutím viacerých agentov. Nebolo by však správne deliť ju formálne na rôzne vrstvy, lebo určité veci sme museli rozhodnúť a naplánovať pre všetkých týchto agentov vopred. Predovšetkým tu ide o zrealizovanie sekvenčnej činnosti. Multiagentovým technológiám svedčí skôr paralelná väzba činností, sekvenciu musíme realizovať na základe vzájomnej aktivácie a inhibície agentov realizujúcich jej jednotlivé fázy [Minsky 1986].

V našom prípade ide o sekvenciu:

1 párenie
2 vykopanie komôrky
3 nájdenie a ochromenie sedlovky
4 odnesenie sedlovky ku komôrke
5 skontrolovanie komôrky
6 vtiahnutie sedlovky do komôrky a vylezenie z komôrky
7 zapečatenie komôrky

Vzájomnú aktiváciu v sekvencii budú zabezpečovať bloky:

1–2 `treba_kopat`
2–3 `treba_potravu`
3–4 `treba_odniest`
4–5 `treba_skontrolovat`
5–6 `treba_vtiahnut`
6–7 `treba_zapecatit`

Jednotlivé články sekvencie zrealizujeme agentmi:

1 `parenie`, `potomstvo`
2 `komorka`
3 `sedlovka`
4 `prinesenie`
5 `kontrola`
6 `vtiahnutie`
7 `pecat`

Párenie je realizované dvomi agentmi, nakoľko sme pri implementácii rozhodli o blokoch realizujúcich sekvenciu až po vyvinutí agenta `parenie`. Preto sme na detekciu spárenia a naštartovanie zakladania potomstva zaviedli samostatného agenta `potomstvo`, hoci by sme neporušili pravidlá ani modifikovaním agenta `parenie`, keďže ide o tú istú vrstvu. Chceli sme si však dokázať, že sa spätným modifikáciám vieme vyhýbať. Samotné párenie spočíva v tom, že v momente ako kutavka zdetekuje samčeka, zosadne na zem a potlačí vrstvy lezenia a lietania, aby sa nehýbala a samček ju mohol oplodniť. Až na fázu zosadnutia na zem, kde musíme predpokladať, že samička môže byť v určitej výške a postupne musí klesať, to nie je veľmi zložité, lebo sa spoliehame na to, že samček (teda užívateľ, ktorý ním hýbe pomocou myši) sa bude snažiť samičku oplodniť. Ide o typický prípad čerpania inteligencie z dynamiky prostredia.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
MsAgentInsTrigger(NULL,proxy,0,500000000,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int samcek = 0;
    int oplodnena = 0;
    int nula = 0;
    int unava = 40;
    int vyska = 0;
    int lietaj = 0;
    int suloz;
    Receive(proxy,0,0);
    MsAgentRead("samcek",&samcek,sizeof(int),NULL);
    MsAgentRead("oplodnena",&oplodnena,sizeof(int),NULL);
    MsAgentRead("lietaj",&lietaj,sizeof(int),NULL);
    MsAgentRead("vyska",&vyska,sizeof(int),NULL);
    MsAgentRead("suloz",&suloz,sizeof(int),NULL);
    MsAgent("KUT",NULL);
    if (!oplodnena) {
        if (samcek) {
            if (lietaj > 0) {
                if (lietaj < vyska) vyska = lietaj;
                lietaj = vyska;
                MsAgentWrite("lietaj",&lietaj,sizeof(int),NULL);
            }
            else {
                MsAgentWrite("hybsa",&nula,sizeof(int),NULL);
                MsAgentWrite("unava",&unava,sizeof(int),NULL);
            }
            MsAgent("KUT",NULL);
        }
    }
}
```

Agent `potomstvo` je typický príklad diferenčného agenta, ktorý sleduje diferenciu prechádzajúceho a súčasného stavu [Minsky 1986]. Zdetekuje, že došlo k oplodneniu a na základe toho dá signál pre starostlivosť o potomstvo (aktivuje blok `treba_kopat`). Po jednorázovom vyslaní signálu však už tento signál ďalej negeneruje. Tým sa líši sekvenčná činnosť od paralelnej, pre ktorú je typické opakovane určitý signál potvrdzovať.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
MsAgentInsTrigger(NULL,proxy,0,500000000,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int oplodnena = 0;
    int bolaoplodnena = 0;
    int jedna = 1;
    Receive(proxy,0,0);
    MsAgentRead("oplodnena",&oplodnena,sizeof(int),NULL);
    MsAgentRead("bolaoplodnena",&bolaoplodnena,sizeof(int),NULL);
    MsAgent("KUT",NULL);
    if (!bolaoplodnena && oplodnena)
        MsAgentWrite("treba_kopat",&jedna,sizeof(int),NULL);
    MsAgentWrite("bolaoplodnena",&oplodnena,sizeof(int),NULL);
    MsAgent("KUT",NULL);
```

```
}
```

Starostlivosť o potomstvo, spočívajúca v nakladení vajíčka na ochrnutú potravu vo vykopanej komôrke, začína vykopaním komôrky. Na to treba nájsť vhodné miesto, čo je ponechané na vrstvy lezenia a lietania. V momente kedy je vhodné miesto zamerané, agent `komorka` sa votrie do vrstvy lietania, spôsobí klesanie k zemi (pokiaľ sa letí) a vrstvy lezenia, kde vhodným dodatočným otáčaním na základe diferencie vzdialenosti od vyhliadnutého miesta doslova zápasí s vrstvou lezenia, aby postupoval správnym smerom. Keď vyhliadnuté miesto dosiahne, inhibuje pohyb a inicializuje kopanie. Na základe pozitívnej väzby od kopania, spúšťa sa hlbšie a kope, až usúdi podľa hĺbky, že komôrka je vykopaná. Vtedy zase riadi vylezenie a po dosiahnutí povrchu, ukončí inhibíciu pohybu a aktivuje blok `treba_potravu`. Inhibícia pohybu nie je (vzhľadom na našu verziu vtierania sa do činnosti) stopercentná, ale nepozorovali sme, že by sa to vôbec prejavilo.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
MsAgentInsTrigger(NULL,proxy,0,500000000,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int treba = 0;
    int lietaj = 0;
    int vyska = 0;
    int komorka = 0;
    int bolakomorka = 0;
    int jedna = 1;
    int nula = 0;
    int hlbka = 0;
    int vykopane = 0;
    Receive(proxy,0,0);
    MsAgentRead("treba_kopat",&treba,sizeof(int),NULL);
    MsAgentRead("komorka",&komorka,sizeof(int),NULL);
    MsAgentRead("bolakomorka",&bolakomorka,sizeof(int),NULL);
    MsAgentRead("lietaj",&lietaj,sizeof(int),NULL);
    MsAgentRead("vyska",&vyska,sizeof(int),NULL);
    MsAgentRead("hlbka",&hlbka,sizeof(int),NULL);
    MsAgentRead("vykopane",&vykopane,sizeof(int),NULL);
    MsAgent("KUT",NULL);
    if (treba) {
        int hotovo = 0;
        if (vykopane) {
            if (vyska < 0) {
                MsAgentWrite("hybsa",&nula,sizeof(int),NULL);
                MsAgentWrite("vylez",&jedna,sizeof(int),NULL);
                MsAgentWrite("oddych",&jedna,sizeof(int),NULL);
            }
            else hotovo = 1;
        }
        else if (komorka) {
            if (lietaj > 0) {
                if (lietaj < vyska) vyska = lietaj;
                MsAgentWrite("lietaj",&vyska,sizeof(int),NULL);
            }
            if (komorka < 150) {
                MsAgentWrite("hybsa",&nula,sizeof(int),NULL);
                MsAgentWrite("kop",&jedna,sizeof(int),NULL);
                MsAgentWrite("oddych",&jedna,sizeof(int),NULL);
                if (hlbka >= 24)
```

```
                MsAgentWrite("vykopane",&jedna,sizeof(int),NULL);
            }
            else if (bolakomorka && bolakomorka < komorka) {
                MsAgentWrite("otocsa",&jedna,sizeof(int),NULL);
            }
        }
        if (hotovo) {
            MsAgentWrite("treba_kopat",&nula,sizeof(int),NULL);
            MsAgentWrite("treba_potravu",&jedna,sizeof(int),NULL);
        }
        MsAgentWrite("bolakomorka",&komorka,sizeof(int),NULL);
        MsAgent("KUT",NULL);
    }
}
```

Hľadanie potravy realizuje agent `sedlovka`. V úvodnej fáze používa rovnakú taktiku ako hľadanie miesta pre komôrku. Po fáze priblíženie nasleduje ochromenie sedlovky niekoľkými vpichmi žihadla. Následne sa aktivuje blok `treba_odniest`.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
MsAgentInsTrigger(NULL,proxy,0,500000000,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int treba = 0;
    int lietaj = 0;
    int vyska = 0;
    int sedlovka = 0;
    int bolasedlovka = 0;
    int jedna = 1;
    int nula = 0;
    int ochrnuta = 0;
    int drzime = 0;
    Receive(proxy,0,0);
    MsAgentRead("treba_potravu",&treba,sizeof(int),NULL);
    MsAgentRead("sedlovka",&sedlovka,sizeof(int),NULL);
    MsAgentRead("bolasedlovka",&bolasedlovka,sizeof(int),NULL);
    MsAgentRead("lietaj",&lietaj,sizeof(int),NULL);
    MsAgentRead("vyska",&vyska,sizeof(int),NULL);
    MsAgentRead("ochrnuta",&ochrnuta,sizeof(int),NULL);
    MsAgentRead("drzime",&drzime,sizeof(int),NULL);
    MsAgent("KUT",NULL);
    if (treba) {
        if (ochrnuta) {
            MsAgentWrite("zihadlo",&nula,sizeof(int),NULL);
            if (!drzime) {
                MsAgentWrite("uchop",&jedna,sizeof(int),NULL);
            }
            else {
                MsAgentWrite("treba_potravu",&nula,sizeof(int),NULL);
                MsAgentWrite("treba_odniest",&jedna,sizeof(int),NULL);
            }
        }
        else if (sedlovka) {
            if (lietaj > 0) {
                if (lietaj < vyska) vyska = lietaj;
```

```
                MsAgentWrite("lietaj",&vyska,sizeof(int),NULL);
            }
            if (bolasedlovka && bolasedlovka < sedlovka) {
                MsAgentWrite("otocsa",&jedna,sizeof(int),NULL);
            }
            else if (lietaj == 0 && sedlovka < 150) {
                MsAgentWrite("hybsa",&nula,sizeof(int),NULL);
                MsAgentWrite("zihadlo",&jedna,sizeof(int),NULL);
                MsAgentWrite("oddych",&jedna,sizeof(int),NULL);
            }
        }
        MsAgentWrite("bolasedlovka",&sedlovka,sizeof(int),NULL);
        MsAgent("KUT",NULL);
    }
}
```

Prinesenie ochrnutej potravy realizuje agent `prinesenie`. Ide o podobnú taktiku, ako keď sa hľadá komôrka, akurát na nej konci nasleduje pustenie sedlovky. (Opäť pritom neriešime problémy s navigáciou, ale "zneužívame" uzavretosť sveta.) Pustenie je možné aplikovať i z výšky, kolmý pád sedlovky zabezpečí simulátor. Následne sa kutavka snaží zosadnúť v mieste komôrky a aktivuje sa blok `treba_skontrolovať`.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
MsAgentInsTrigger(NULL,proxy,0,500000000,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int treba = 0;
    int lietaj = 0;
    int vyska = 0;
    int komorka = 0;
    int bolakomorka = 0;
    int jedna = 1;
    int nula = 0;
    int drzime = 0;
    Receive(proxy,0,0);
    MsAgentRead("treba_odniest",&treba,sizeof(int),NULL);
    MsAgentRead("komorka",&komorka,sizeof(int),NULL);
    MsAgentRead("bolakomorka",&bolakomorka,sizeof(int),NULL);
    MsAgentRead("lietaj",&lietaj,sizeof(int),NULL);
    MsAgentRead("vyska",&vyska,sizeof(int),NULL);
    MsAgentRead("drzime",&drzime,sizeof(int),NULL);
    MsAgent("KUT",NULL);
    if (treba && drzime) {
        if (komorka) {
            if (lietaj > 0) {
                if (lietaj < vyska) vyska = lietaj;
                MsAgentWrite("lietaj",&vyska,sizeof(int),NULL);
            }
            if (komorka < 150) {
                MsAgentWrite("treba_odniest",&nula,sizeof(int),NULL);
                MsAgentWrite("pusti",&jedna,sizeof(int),NULL);
                MsAgentWrite("treba_skontrolovat",&jedna,sizeof(int),NULL);
            }
            else if (bolakomorka && bolakomorka < komorka) {
                MsAgentWrite("otocsa",&jedna,sizeof(int),NULL);
```

```
            }
        }
        MsAgentWrite("bolakomorka",&komorka,sizeof(int),NULL);
        MsAgent("KUT",NULL);
    }
    else if (treba && !drzime) {
        MsAgentWrite("treba_odniest",&nula,sizeof(int),NULL);
        MsAgentWrite("treba_potravu",&jedna,sizeof(int),NULL);
        MsAgent("KUT",NULL);
    }
}
```

Kontrola komory prebehne podobným spôsobom ako vykopanie komôrky, pozostáva z fáz nájdenia vstupu do komôrky, spustenia sa a vylezenia. Realizuje ju agent `kontrola`. Následne sa aktivuje blok `treba_vtiahnut`.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
MsAgentInsTrigger(NULL,proxy,1,0,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int treba = 0;
    int lietaj = 0;
    int vyska = 0;
    int komorka = 0;
    int bolakomorka = 0;
    int jedna = 1;
    int nula = 0;
    int hlbka = 0;
    int overene = 0;
    Receive(proxy,0,0);
    MsAgentRead("treba_skontrolovat",&treba,sizeof(int),NULL);
    MsAgentRead("komorka",&komorka,sizeof(int),NULL);
    MsAgentRead("bolakomorka",&bolakomorka,sizeof(int),NULL);
    MsAgentRead("lietaj",&lietaj,sizeof(int),NULL);
    MsAgentRead("vyska",&vyska,sizeof(int),NULL);
    MsAgentRead("hlbka",&hlbka,sizeof(int),NULL);
    MsAgentRead("overene",&overene,sizeof(int),NULL);
    MsAgent("KUT",NULL);
    if (treba) {
        int hotovo = 0;
        if (overene) {
            if (vyska < 0) {
                MsAgentWrite("hybsa",&nula,sizeof(int),NULL);
                MsAgentWrite("vylez",&jedna,sizeof(int),NULL);
                MsAgentWrite("oddych",&jedna,sizeof(int),NULL);
            }
            else hotovo = 1;
        }
        else if (komorka) {
            if (lietaj > 0) {
                if (lietaj < vyska) vyska = lietaj;
                MsAgentWrite("lietaj",&vyska,sizeof(int),NULL);
            }
            else if (bolakomorka && bolakomorka < komorka) {
                MsAgentWrite("otocsa",&jedna,sizeof(int),NULL);
```

```
            }
            else if (komorka < 150) {
                MsAgentWrite("hybsa",&nula,sizeof(int),NULL);
                MsAgentWrite("spustisa",&jedna,sizeof(int),NULL);
                MsAgentWrite("oddych",&jedna,sizeof(int),NULL);
                if (hlbka >= 24)
                    MsAgentWrite("overene",&jedna,sizeof(int),NULL);
            }
        }
        if (hotovo) {
            MsAgentWrite("treba_skontrolovat",&nula,sizeof(int),NULL);
            MsAgentWrite("treba_vtiahnut",&jedna,sizeof(int),NULL);
            MsAgentWrite("overene",&nula,sizeof(int),NULL);
        }
        MsAgentWrite("bolakomorka",&komorka,sizeof(int),NULL);
        MsAgent("KUT",NULL);
    }
}
```

Vtiahnutie potravy riadi agent `vtiahnutie`. Jeho spôsob činnosti možno rozdeliť do fáz nájdenia a uchopenia sedlovky, spustenia sa, pustenia sedlovky a vylezenia. Dôležité si je všimnúť, že vo fáze nájdenia nesmieme použiť príliš silný mechanizmus, ktorý by sedlovku nachádzal v celom simulovanom svete. Vtedy by sme totiž nedostali požadované zlyhanie kutavčej inteligencie v experimente s odsúvaním sedlovky. Na druhej strane, s istou pravdepodobnosťou pri našom riešení nastáva prípad, keď kutavka pri vyliezaní vyletí z komôrky, nestihne si sedlovku uchopiť a chvíľu si musí zakrúžiť, kým sa opäť dostane na správnu stopu. Túto možnosť odchýlky od správania sme zatiaľ neeliminovali, nakoľko nám slúži ako dobrý príklad robustnosti systému, kde síce dôjde k vydaniu sa po zlej ceste, ale systém opäť správnu cestu nájde.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
MsAgentInsTrigger(NULL,proxy,1,0,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int treba = 0;
    int lietaj = 0;
    int vyska = 0;
    int komorka = 0;
    int bolakomorka = 0;
    int jedna = 1;
    int nula = 0;
    int hlbka = 0;
    int vtiahnute = 0;
    int drzime = 0;
    int kdeje = 0;
    Receive(proxy,0,0);
    MsAgentRead("treba_vtiahnut",&treba,sizeof(int),NULL);
    MsAgentRead("komorka",&komorka,sizeof(int),NULL);
    MsAgentRead("bolakomorka",&bolakomorka,sizeof(int),NULL);
    MsAgentRead("lietaj",&lietaj,sizeof(int),NULL);
    MsAgentRead("vyska",&vyska,sizeof(int),NULL);
    MsAgentRead("hlbka",&hlbka,sizeof(int),NULL);
    MsAgentRead("vtiahnute",&vtiahnute,sizeof(int),NULL);
    MsAgentRead("drzime",&drzime,sizeof(int),NULL);
    MsAgentRead("kdeje",&kdeje,sizeof(int),NULL);
    MsAgent("KUT",NULL);
```

```
    if (drzime) kdeje = 0;
    if (treba) {
        if (vtiahnute) {
            if (vyska < 0) {
                MsAgentWrite("hybsa",&nula,sizeof(int),NULL);
                MsAgentWrite("vylez",&jedna,sizeof(int),NULL);
                MsAgentWrite("oddych",&jedna,sizeof(int),NULL);
            }
            else {
                MsAgentWrite("treba_zapecatit",&jedna,sizeof(int),NULL);
                MsAgentWrite("treba_vtiahnut",&nula,sizeof(int),NULL);
            }
        }
        else if (drzime) {
            if (komorka) {
                if (lietaj > 0) {
                    if (lietaj < vyska) vyska = lietaj;
                    MsAgentWrite("lietaj",&vyska,sizeof(int),NULL);
                }
                if (komorka < 150) {
                    MsAgentWrite("hybsa",&nula,sizeof(int),NULL);
                    if (hlbka >= 24) {
                        MsAgentWrite("vtiahnute",&jedna,sizeof(int),NULL);
                        MsAgentWrite("pusti",&jedna,sizeof(int),NULL);
                    }
                    else {
                        MsAgentWrite("spustisa",&jedna,sizeof(int),NULL);
                    }
                    MsAgentWrite("oddych",&jedna,sizeof(int),NULL);
                }
                else if (bolakomorka && bolakomorka < komorka) {
                    MsAgentWrite("otocsa",&jedna,sizeof(int),NULL);
                }
            }
        }
        else {
            if (kdeje >= 3) {
                MsAgentWrite("treba_vtiahnut",&nula,sizeof(int),NULL);
                MsAgentWrite("treba_potravu",&jedna,sizeof(int),NULL);
            }
            else {
                kdeje++;
                MsAgentWrite("uchop",&jedna,sizeof(int),NULL);
            }
        }
        MsAgentWrite("bolakomorka",&komorka,sizeof(int),NULL);
    }
    MsAgentWrite("kdeje",&kdeje,sizeof(int),NULL);
    MsAgent("KUT",NULL);
}
```

Po vylezení v komôrky, v ktorej sa už nachádza potrava s nakladeným vajíčkom, sa komôrka zapečatí, čo realizuje celkom (možno až príliš) jednoduchý agent `pecat`.

```
pid_t proxy = qnx_proxy_attach(0,0,0,-1);
```

```
MsAgentInsTrigger(NULL,proxy,2,0,0,0);
MsAgent("KUT",NULL);
for (;;) {
    int treba = 0;
    int jedna = 1;
    int nula = 0;
    Receive(proxy,0,0);
    MsAgentRead("treba_zapecatit",&treba,sizeof(int),NULL);
    MsAgent("KUT",NULL);
    if (treba) {
        MsAgentWrite("zapecat",&jedna,sizeof(int),NULL);
        MsAgentWrite("treba_zapecatit",&nula,sizeof(int),NULL);
    }
    MsAgent("KUT",NULL);
}
```

To sú všetky agenty tvoriace vrstvu zakladania potomstva.

3.11 Vyhodnotenie experimentov

Počas svojich behov, náš systém napriek nízkej kvalite použitej grafiky vytvára dobrý subjektívny dojem. Správanie kutavky nie je stereotypné, ale zato napĺňa želanú šablónu. Kutavka sa naoko dosť trápi, ale dosahuje stanovený cieľ. Chvíľami pôsobí dojmom, že je slepá, ale to je dané rozmermi simulovaného sveta a prísnymi limitmi na viditeľnosť objektov v ňom. Kutavkine simulované oči naozaj nie sú výkonné, aby to nemala až také ľahké.

Čo sa týka produkovania "neinteligentného správania" v prípade odsúvania sedlovky, kutavka ho poslušne generuje, ide teda o dobrú aproximáciu. Na druhej strane, rovnako dobre by sme vedeli kutavku naprogramovať tak, aby komôrku už druhý krát nekontrolovala. Vidí sa nám však, že v prípade, že by kutavka mohla vykopať viac komôrok, pričom by nemala zaručené, že sa jej podarí sedlovku vtiahnuť dnu (kutavka najprv vykope komôrku a až potom nájde sedlovku, ktorá môže viac alebo menej vypasená), by už bolo výrazne jednoduchšie použiť kontrolu zakaždým.

Z hľadiska emergencie sme zaznamenali skôr neželaný efekt pri vťahovaní sedlovky do komôrky. Kutavke sa niekedy nepodarí sedlovku uchopiť, čo je spôsobené tým, že sa v kritický okamih nepodarí prehlušiť vrstvu lietania a táto si práve zaželá lietať. Samozrejme bolo by to možné eliminovať. Na druhej strane však niekedy pritom kutavka len trošku nadletí sedlovku a vykonáva presne taký pohyb ako keď chce osa preletieť sklom. Tento pohyb naopak vyvoláva veľmi dobrý subjektívny dojem.

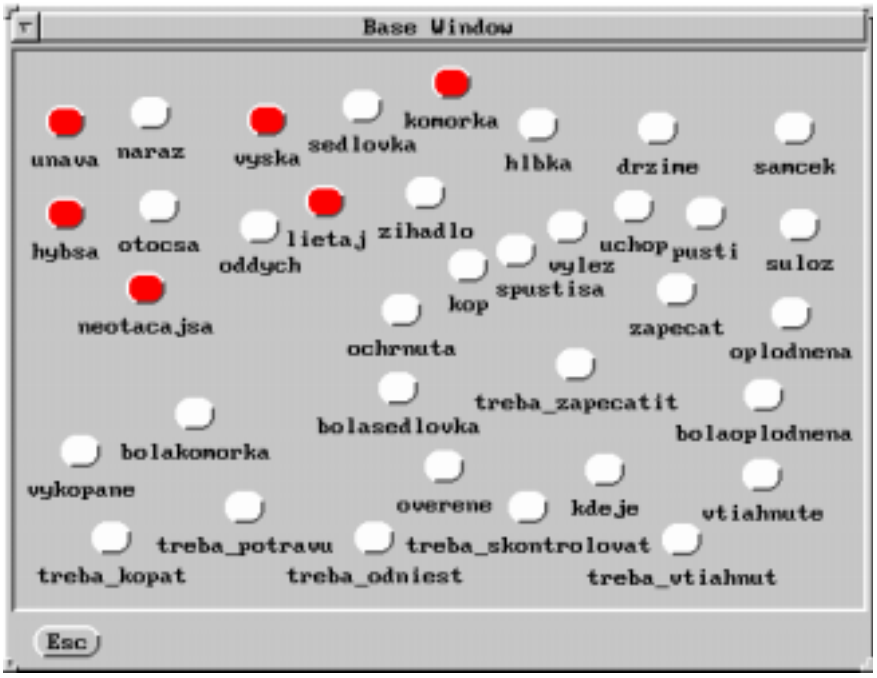
Pri realizovaní sekvencie môžeme na základe skúseností (a i z uvedených kódov to vyplýva) konštatovať, že rozdelenie jednotlivých činností do agentov by sa dalo urobiť aj lepšie. Mohli by sme mať viac a jednoduchších agentov.

Mohli by sme taktiež vziať do úvahy ďalšie vrstvy sedlovky, ktoré by síce nemuseli byť relevantné v konečnom riešení, ale zodpovedali by evolučným medzičlánkom v biologickej evolúcii kutaviek. Takémuto zložitému zakladaniu potomstva museli evolučne predchádzať oveľa jednoduchšie varianty ležiace v nižších hierarchických vrstvách, ktoré sa už ale dnes nedostanú ku slovu.

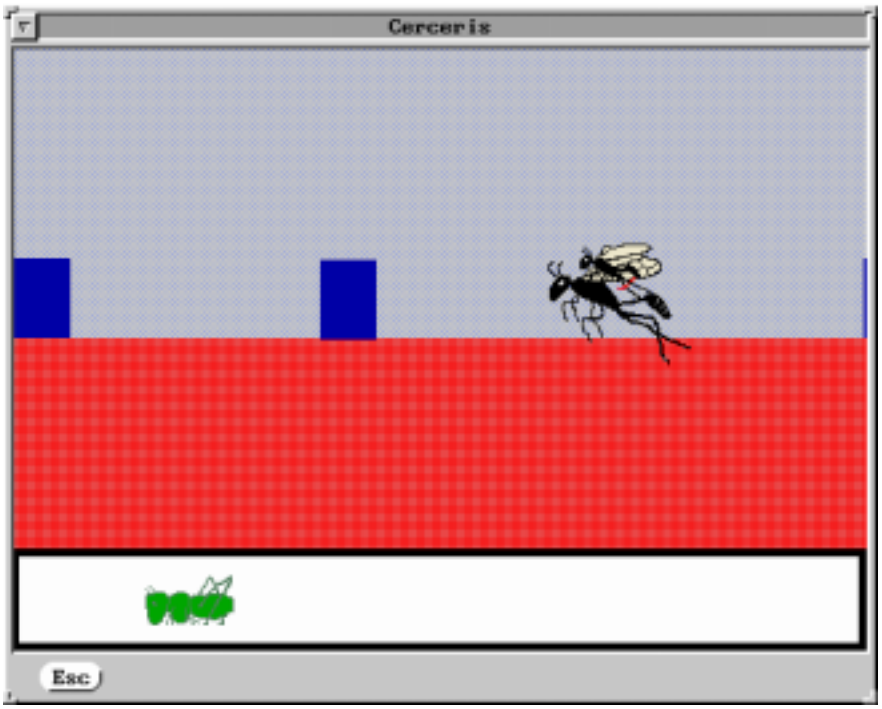
Z analýzy vyvinutých kódov možno konštatovať, že existujú dva okruhy podobných stratégii ktoré používa viac agentov. V podstate v nich tvoria duplikovaný kód. Agenty komôrka, sedlovka a prinesenie zdieľajú kód realizujúci nájdenie niečoho, v našom prípade komôrky alebo sedlovky. Podobne agenty komôrka, kontrola a vtiahnutie zdieľajú kód realizujúci spustenie sa a vylezenie. Tieto dva okruhy činností sú kandidátmi na tzv. agent-pronome [Minsky 1986], ktorý by bol pri našich implementačných prostriedkoch realizovaný tak, že by sa do jeho určitých blokov zapísali mená iných blokov a agent by teda operoval nad rôznymi skupinami blokov, podľa aktuálneho nastavenia.

Získali sme taktiež určitú predstavu, ako by bolo možné zaviesť v systéme učenie, ktoré by dokázalo z vyššej úrovne korigovať zbytočnú mnohonásobnú kontrolu komôrky. Bol by na to potrebný agent, ktorý odpamätáva určitý kontext – v našom prípade obsah určitých blokov – a za istých okolností ho vyvolá na pôvodné miesto. Ide o princíp tzv. agenta-memorizera [Minsky 1986].

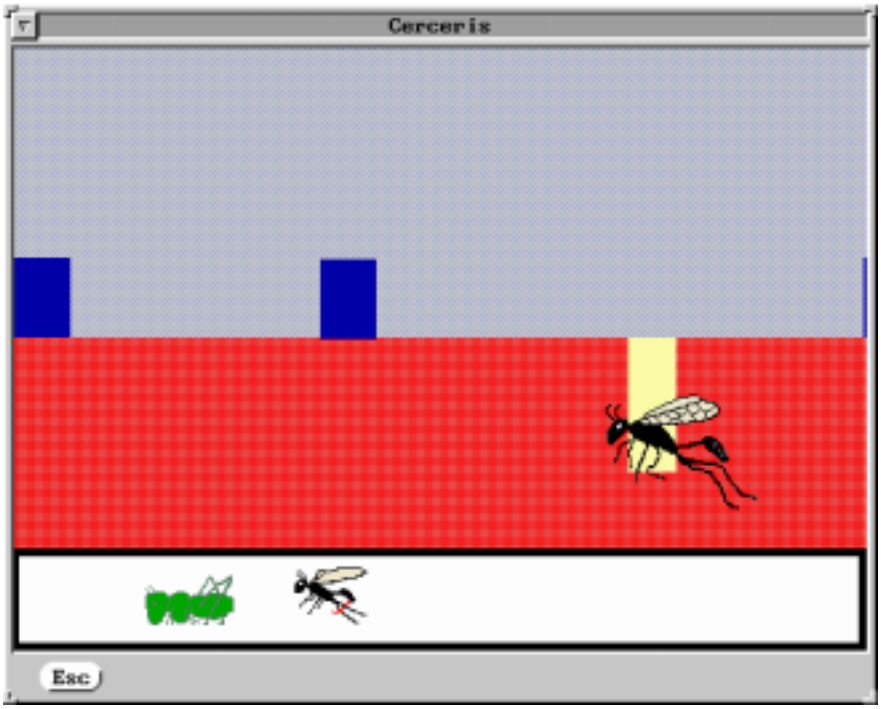
Štandardný priebeh experimentu vidno na obrázkoch 3.4 až 3.11.



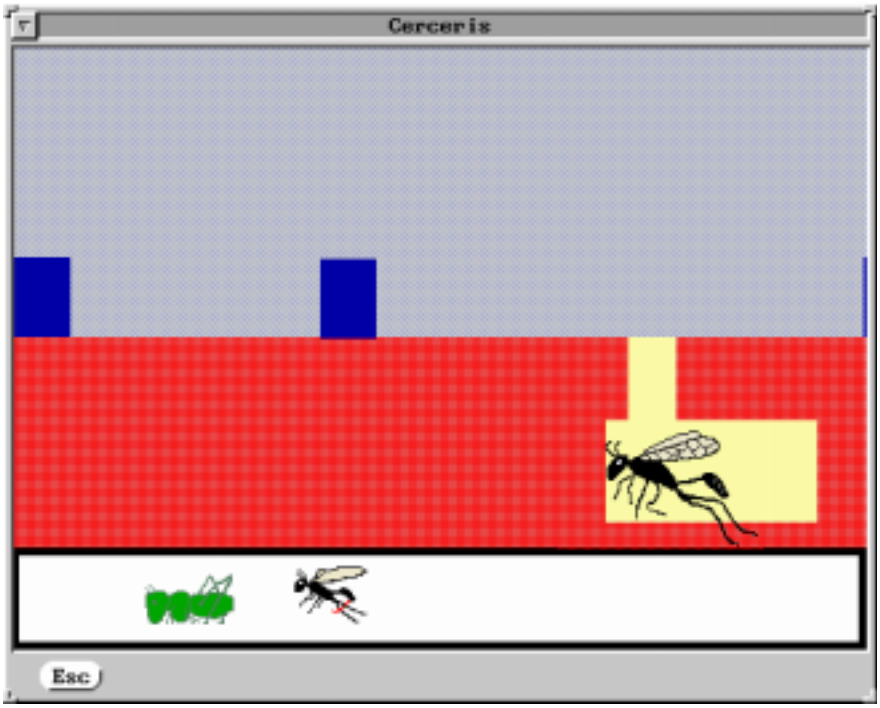
Obrázok 3.3: Znázornenie aktivity systému (kutavčí mozog)



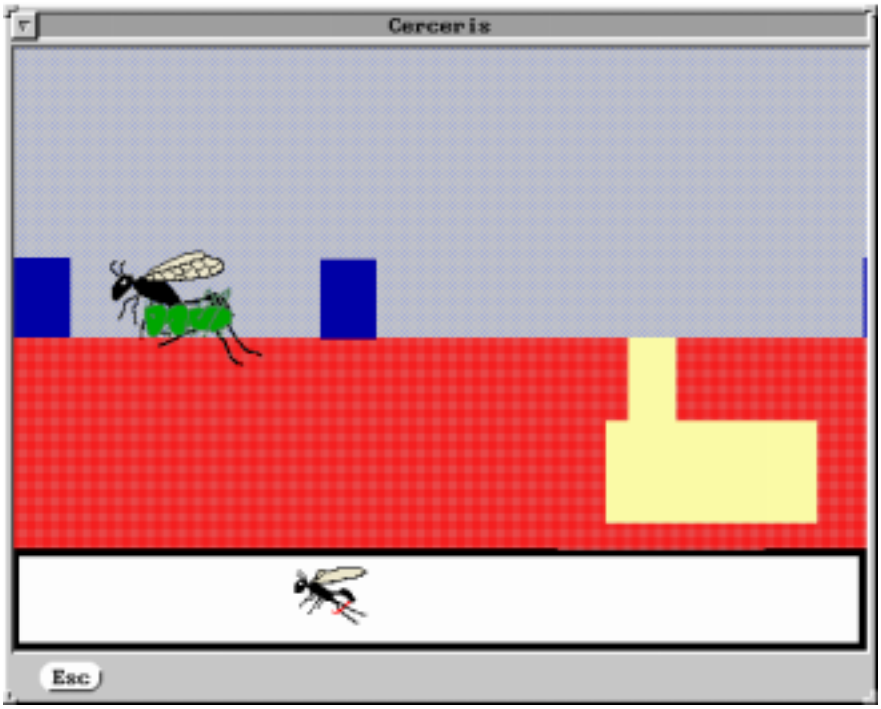
Obrázok 3.4: Párenie



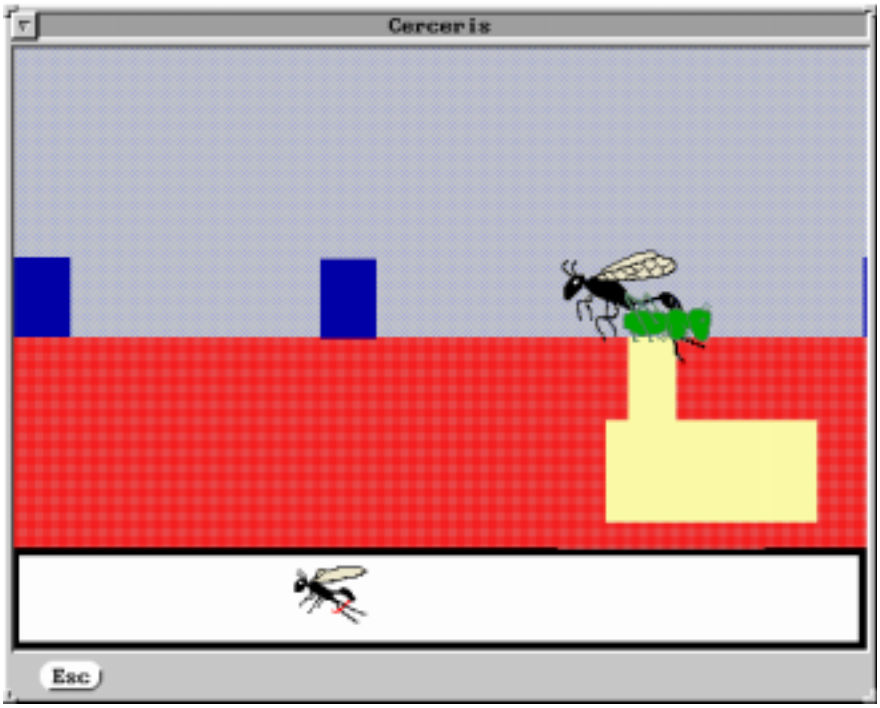
Obrázok 3.5: Kopanie komôrky



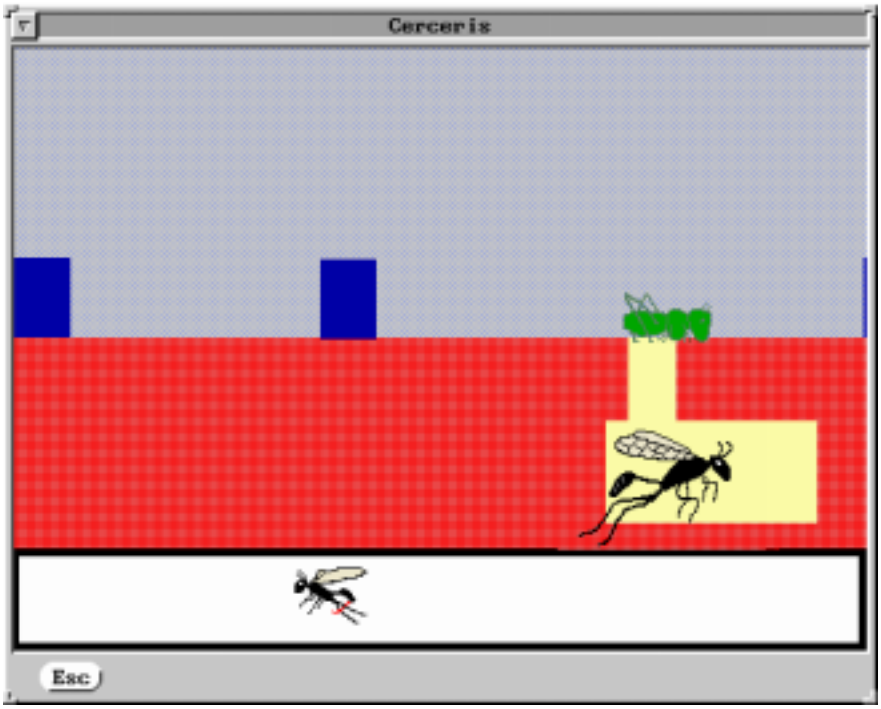
Obrázok 3.6: Komôrka vykopaná



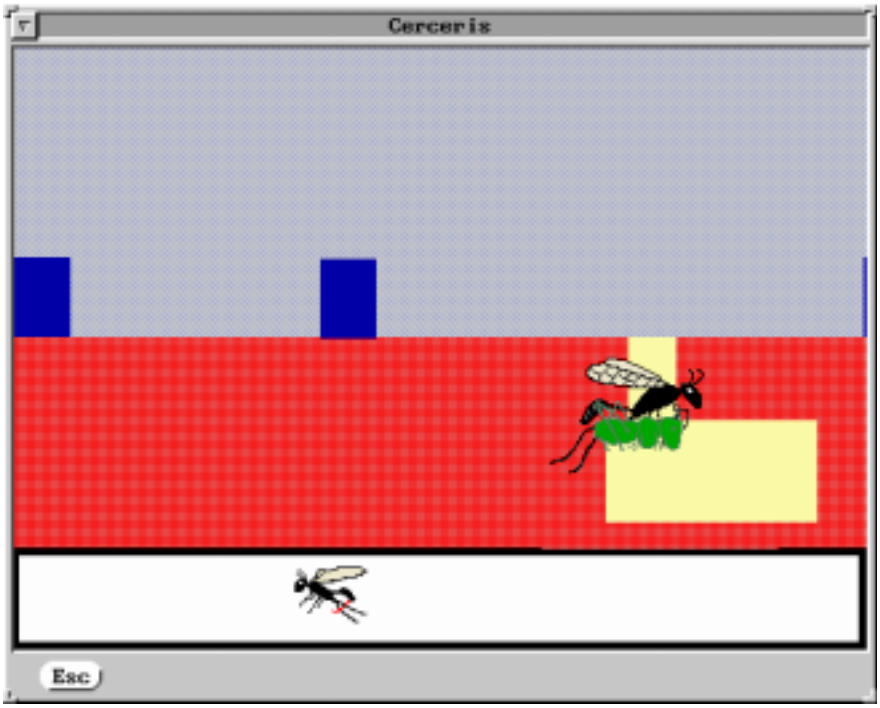
Obrázok 3.7: Ochromenie sedlovky



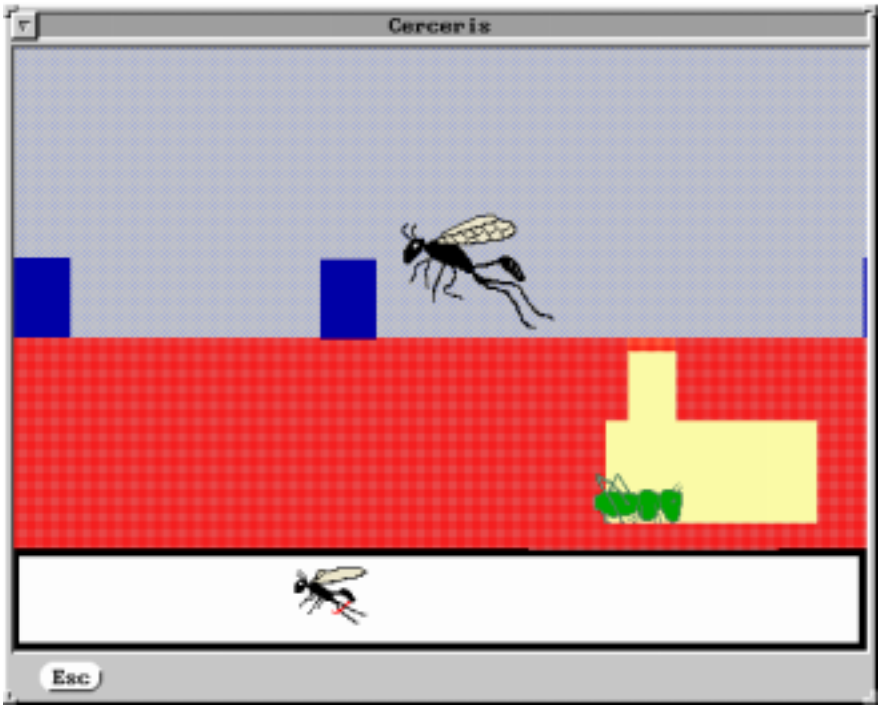
Obrázok 3.8: Prinesenie sedlovky ku komôrke



Obrázok 3.9: Kontrola komôrky



Obrázok 3.10: Vtiahnutie sedlovky do komôrky



Obrázok 3.11: Vajíčko nakladené, komôrka zapečatená

3.12 Zobrazenie aktivity systému

Zaujímala nás taktiež otázka ako by sme mohli znázorniť aktivitu v takomto systéme. Vzhľadom na to, že agenty sú v neustálej činnosti, rozhodli sme za zobrazovať zápisy do jednotlivých blokov. Takto sme mohli zobraziť kutavčí "mozog" ako vidno na obrázku 3.3.

Voľným okom možno na takomto zobrazení pozorovať ako synchrónny výskyt aktivity, tak aj občasné vynechanie aktivity v miestach, ktoré spravidla aktívne sú. Keď zvážime zložitosť teórii vysvetľujúcich vznik synchrónnych oscilácii a vynechávanie aktivity na zázname EEG, celkom sa nám pozdáva myšlienka že by mohli pochádzať z agentovej povahy modulov v mozgu. Zatiaľ sa nám však nepodarilo toto zobrazenie previesť na meranie, ktoré by dalo záznam čo len podobný EEG.

3.13 Zhrnutie

Zvolili sme si konkrétne správanie jednoduchého živého systému. Toto správanie sa nám podarilo aproximovať umelým systémom, ktorý bol vyvinutý podľa našej verzie subsumpčnej architektúry. Získali sme skúsenosti i zaujímavé podnety pre vývoj zložitejších, úplnejších, či kvalitnejšie simulovaných systémov.

4. Záver

V predchádzajúcej kapitole sme zrealizovali jednu aproximáciu konkrétneho správania živého systému systémom umelým. Tento systém sme budovali na princípoch subsumpčnej architektúry. Využili sme pritom viacero zjednodušení, avšak napriek môžeme tvrdiť, že náš pokus posilnil naše presvedčenie o biologickej relevancii tejto architektúry. A to nielen v zmysle evolučnom, ako to zamýšľal Brooks, ale i v zmysle epistemologickom. Táto architektúra môže slúžiť na opis jednoduchších živých systémov a utváranie teórií o vzťahu ich štruktúry a správaní, ktoré produkujú.

V tomto zmysle nás lákajú ďalšie pokusy. Napríklad pozorovanie osy z rodu vespula pri transporte ulovenej muchy do osieho hniezda, ktoré opísal prvý už E. Darwin (starý otec známeho tvorcu evolučnej teórie) [Chalifman 1983]. Osa z rodu vespula sa totiž po ulovení muchy, snaží túto odniesť do hniezda. Mucha je však pre ňu dosť ťažká (samozrejme záleží na tom ako je vypasená), takže po niekoľkých neúspešných pokusoch o vzlietnutie, jej odkusne krídla a skúša to zas. Ak sa jej opäť nepodarí vzlietnuť, odkusne nohy a skúša to zas. Ak sa jej to opäť nepodarí porcuje ju a obžiera až s ňou vzlietne a odletí do hniezda, kde ňou nakŕmi larvy. Celý tento proces vyzerá v očiach pozorovateľa div nie tak, že osa vykonáva vedomú činnosť. Všeobecne sa považuje za dôkaz, že hmyz nemá až tak strojovú konštitúciu, ako by to chceli kognitívni vedci. Predstavme si však dve vrstvy subsumpčnej architektúry. Nižšiu, ktorá realizuje obradné žratie ulovenej muchy, kde najprv poobhrýzame čo nám nechutí a potom sa pustíme do toho čo nám chutí. A vyššiu vrstvu, ktorá nám navráva, že je našou povinnosťou nosiť do hniezda potravu larvám a blokuje nižšiu vrstvu, aby sme potravu nezožrali sami. Vyššia vrstva je aktívna, pokiaľ to vyčerpanie osy umožňuje. Preto osa chytí muchu a pokúša sa s ňou letieť. Robí to dovtedy kým sa neunaví natoľko, že vyššia vrstva zmĺkne. Vtedy sa automaticky aktivuje nižšia vrstva a odhryzneme krídla. Tým sa ale váha muchy výrazne zníži a zároveň si predsa len trochu oddýchneme, teda i pri danom stave vyčerpanosti má zmysel sa ju pokúsiť odniesť znovu. Znova sa teda aktivuje vyššia vrstva a osa sa snaží odletieť. Keď sa unaví, vrstva sa opäť deaktivuje a nižšia vrstva pokračuje v jedení kde prestala. Tak to ide až sa podarí odletieť. Naozaj by to tak mohlo byť. Keby sme navyše mali šťastie, že nižšia vrstva je použitá i pre jedenie za normálnych podmienok, mohli by sme do osieho hniezda umiestniť dostatočný počet zabitých múch a pri pozorovaní toho, ako ich budú jesť, si potvrdiť, že ich i za normálnych okolností jedia obradným spôsobom "najprv dať preč čo nechutí". To by sme mohli považovať za dôkaz toho, že naša architektúra je dobrý prostriedok na opis fungovania hmyzu. K takémuto výkonu nám samozrejme chýbajú entomologické vedomosti a žiada si spoluprácu s odborníkmi z oblasti entomológie a etológie.

Ďalšou témou nášho záujmu je dovŕšiť fúziu Minského teórie mysle postulovanej v [Minsky 1986] a neskorších prácach so subsumpčnou architektúrou. V pôvodnom znení subsumpčnej architektúry na to nie je veľa možností, avšak s možnosťou pracovať s pomenovanými blokmi, miesto nepomenovaných vedení, sa tieto možnosti výrazne zlepšujú. (Aby sme boli presní, Brooks pomenúval jednotlivé vedenia, ale malo to len dokumentačný význam, jeho modul si nemohol povedať: "a teraz chcem na vstup pripojiť vedenie XY".) S pomocou rôznych druhov Minského agentov, by sme mohli byť schopní modelovať napríklad rôzne zlyhania v psychologických experimentoch.

Ďalšou cestou rozvoja by bolo zdokonaliť technológiu simulátora. Použitie VRML by prinieslo nielen lepšiu grafiku, ale i možnosť simulovať reálne krokové motory riadiace robotický systém, či jednotlivé svaly hmyzu. Žiaľ pod QNX4 nie je VRML prístupné.

Ďalšou možnosťou je znižovať rozdiel medzi kódom a dátami v použitom modeli natoľko, aby bolo možné zaviesť univerzálneho agenta, ktorý interpretuje kódový zápis, ktorý prijíma ako dáta. V praxi by to znamenalo prechod od jazyka C k jazyku LISP, s reprezentačnou sémantikou zahrňujúcou ako dáta tak kód. Takýmto jazykom je napr. jazyk LINDA, ktorý však zatiaľ nemá ambície na poli

umelej inteligencie. Čiastočne je to preto, že neoperuje pod operačnými systémami, ktoré by výkonom umožňovali realizovať zložitejšie systémy. Pod výkonom pritom rozumieme hlavne schopnosť striedať v procesore (prípadne vo viacerých procesoroch) jednotlivé procesy. QNX4 vydrží na bežnom hardwari rádovo tisíce procesov a rýchlosť ich prepínania sa pohybuje na úrovni nano až mikrosekúnd. Aby sme si uvedomili aké priepastné rozdiely z hľadiska jednotlivých operačných systémov tu sú, uvedieme, že pomer prepnutí na rovnakom hardware medzi Windows NT a QNX4 je 1:100. Ani systém vyvinutý v tejto práci by pravdepodobne nemohol fungovať pod inou súčasnou platformou, minimálne by to bolo podstatne ťažšie. Uvidíme teda čo prinesie pokrok v operačných systémoch v najbližších rokoch a kam sa posunú hranice našich možností.

References

[Brooks 1986a] Brooks R. A.: Achieving Artificial Intelligence through Building Robots. (A. I. Memo No. 899), MIT, AI Lab, Cambridge, Mass., 1986

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

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

[Brooks 1991] Brooks R. A.: Intelligence without representation. Artificial Intelligence 47, (1991), 139–159

[Brooks 1993] Brooks R.: Building Brains for Bodies. (A. I. Memo No. 1439), MIT AI Lab, Cambridge, Mass., 1993

[Brooks 1997] Brooks, R.A., From Earwigs to Humans. Robotics and Autonomous Systems, Vol. 20, Nos. 2–4, June 1997, pp. 291–304

[Cianciarini 1999] Cianciarini P.: Lectures on coordination models and languages. EASSS '99, (1999)

[Dawkins 1996] Dawkins, R.: Rieka z raja. Bratislava: Archa 1996

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

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

[Gál 2000] Gál E.: Intuitívna psychológia a kognitívne vedy. In: Kognitívne vedy III (Kvasnička V., Pospíchal J., eds.), (2000), 109-114.

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

[Chalifman 1983] Chalifman J.: Okrídlení korzári. Bratislava: Mladé letá 1983

[Kelemen 1989] Kelemen J.: Myslenie, počítač... Spektrum, Bratislava, 1989

[Kelemen a kol. 1992] Kelemen J., Ftáčnik M., Kalaš I., Mikulecký P.: Umelá inteligencia, Alfa, Bratislava, 1992

[Kelemen - Kelemenová 1992] Kelemen J., Kelemenová A.: A grammar-theoretic treatment of multiagent systems. Cybernetics and Systems 26, (1992), 621–633

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

[Kelemen 1994] Kelemen J.: Strojovia a agenty, Archa, Bratislava, 1994

[Knight 1993] Knight K.: Are Many Reactive Agents Better Than a Few Deliberative Ones? In: Proc. IJCAI '93, Chambery, 1993, 132–137

[Lúčny 1994] Lúčny A.: Emergentné správanie v kolóniách agentov. Diplomová práca, Katedra umelej inteligencie, Matematicko-fyzikálna fakulta, Univerzita Komenského, Bratislava, 1994

[Lúčny 1997] Lúčny A.: Lúčny A.: Architektúra inteligentných programových systémov. Projekt dizertačnej práce, Matematicko-fyzikálna fakulta, Univerzita Komenského, Bratislava, 1997

[Lúčny 1999a] Lúčny A.: Reaktívny model inteligentného systému. In: Kognitívne vedy II (Kvasnička V., Pospíchal J., eds.), (1999), 75-84

[Lúčny 1999b] Lúčny A.: Architektúra inteligentných programových systémov. In: Gramatické, multiagentové a znalostné systémy (Kelemen J., ed.), (1999), 75-98

[Lúčny 2001a] Lúčny A.: Hľadanie kvalitatívneho rozdielu. In.: Kognice a umělý život (Kelemen J., Kvasnička V., Pospíchal J. eds.), zborník česko-slovenskej konferencie, Smolenice, Sliezka univerzita v Opave, 2001

[Lúčny 2001b] Lúčny A.: Refine rather than design. sumbmited to SOFSEM '2001

[Matarić 1994] Matarić 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

[McFarland, Bosser 1993] McFarland D., Bosser T.: Intelligent Behavior in Animals and Robots. The MIT Press, Cambridge, Mass., 1993

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

[Minsky 1996] Minsky M.: Konštrukcia mysle. (Kelemen J., ed.), Archa, Bratislava, 1996

[Mlichová 1993] Mlichová R.: Niektoré, experimenty so subsumpčnou architektúrou v nedeliberatívnej robotike. Diplomová práca, Katedra umelej inteligencie, Matematicko-fyzikálna fakulta, Univerzita Komenského, Bratislava 1993

[Nilson 1997] Nilson J. N.: Artificial Intelligence, A new synthesis. Morgan Kaufmann Publishers, Inc., San Francisco, Ca., 1997.

[Resnick 1994] Resnick M.: Turtles, Termites, and Traffic jams. The MIT Press, Cambridge, Mass., 1994

[Singh 1994] Multiagent Systems. Springer-Verlag, Berlin, 1994

[Stein 1991] Stein L. A.: Imagination and Situated Cognition. (A. I. Memo No. 1277), MIT AI Lab, Cambridge, Mass., 1991.