Architektúra inteligentných programových systémov

Andrej Lúčny

Matematicko-fyzikálna fakulta, Univerzita Komenského v Bratislave
lucny@fmph.uniba.sk
http://www.fmph.uniba.sk/~lucny

Abstrakt

Príspevok naznačuje paralelu medzi modernými trendmi v umelej inteligencii a softwarovom inžinierstve. Snaží sa odhaliť spoločnú príčinu vážnych problémov s ktorými tieto disciplíny zápasia a navrhnúť metódy, ktorými by ich bolo možné odďaľovať, nakoľko vyriešiť ich je pravdepodobne principiálne nemožné. Dôraz sa pritom kladie na implementačné architektúry a ich vlastnosti. Autor príspevku verí, že práve štruktúry a metódy používané vo fáze implementácie nám bránia vyvinúť systémy s rozsiahlou funkčnosťou či bohatým (inteligentným) správaním (čo je podľa náhľadu autora to isté). Zdá sa, že potenciálne sme schopní vyšpecifikovať všetky vlastnosti, ktoré do spomínaných systémov treba vložiť na to, aby robili to čo od nich očakávame. Nedisponujeme však takou implementačnou architektúrou, ktorá by nám umožnila vložiť tento obrovský súbor vlastností do jedného celku. Postupné pridávanie týchto vlastností má vždy za následok zmätok v syntaktických štruktúrach s ktorými pracujeme, kde buď práca potrebná na zavedenie ďalšej vlastnosti je vždy väčšia a väčšia, výsledkom čoho je vznik chýb až strata celkovej funkčnosti alebo práca potrebná na použitie týchto štruktúr je stále väčšia až astronomicky väčšia, čo má za následok nepoužiteľnosť z časových dôvodov. Za najperspektívnejšie architektúry, ktoré by boli schopné tomuto fenoménu čiastočne odolávať autor považuje architektúry založené na multiagentových systémoch, z nich odporúča do pozornosti variant s použitím čisto reaktívnych agentov nepriamo komunikujúcich cez prostredia, pričom systém z nich zložený je dekomponovaný aktivitou a implementovaný inkrementálne metódou zdola - nahor.

„Slová sú len naše nástroje a ich púha prítomnosť v slovníku nemusí znamenať, že v skutočnom svete niečomu zodpovedajú“

Richard Dawkins

1. Úvod

Zámerom tohto príspevku je načrtnúť možnosti aplikácie myšlienok R. Brooksa, ktoré boli úspešne použité v oblasti mobilnej robotiky, v softwarovom inžinierstve, menovite

pri tvorbe programových systémov komplexného zamerania pracujúcich v reálnom čase s použitím implementačných platforiem známych pod názvom multiagentové systémy. Základné črty Brooksovej subsumpčnej architektúry pritom budeme považovať za ukazovateľ správneho smeru v dnes už veľmi bohatom, ale aj veľmi málo selektovanom spektre multiagentových systémov.

Túto ideovú aplikáciu by sme taktiež chceli začleniť do rámca umelej inteligencie. Naše snaženie sa teda pokúsime prezentovať nielen ako snahu po dosiahnutí určitého súboru želaných vlastností programových systémov, ale ako snahu o vyvinutie novej technológie pre tvorbu systémov, ktoré by si zaslúžili označenie „inteligentné“ o niečo viac než tie, ktoré týmto spôsobom označujeme v súčasnosti. Budeme pátrať po podstate tejto inteligencie a budeme sa ju snažiť zadefinovať na základe pojmov bežných v softwarovom inžinierstve.

Základným predpokladom takejto fúzie softwarového inžinierstva a umelej inteligencie je tušenie, že ďalší pokrok v oboch oblastiach naráža na rovnakú prekážku. Zlyhania oboch menovaných disciplín majú často zdanlivo nevysvetliteľný charakter, keď na jednej strane teoreticky nič nebráni vytvoreniu systému s projektovanými vlastnosťami, avšak na druhej strane sa ho nikomu nepodarí prakticky zrealizovať. Za kľúč k pochopeniu prečo je to tak, navrhneme kvalitu architektúry, ktorá sa pri implementácii systémov používa, t.j. spôsob akým sa sémantika systému zaznamenáva do syntaktickej podoby. Naznačíme, že prirodzenou vlastnosťou každej architektúry je jej znehodnocovanie rastúce s pribúdajúcim počtom zaimplementovaných sémantických vzťahov do jej syntaktických štruktúr. Zároveň navrhneme architektúru, ktorá je touto vlastnosťou čo najmenej postihnutá.

Pritom budeme mnohé pojmy používať v o niečo užšom význame, než je obvyklé. Najvýraznejším obmedzením našich úvah bude zameranie sa na také systémy, ktoré pracujú v reálnom čase, t.j. ich činnosť pretrváva v čase a na každý vstup odpovedajú výstupom do určitého času. Takže napr. pod systémami umelej inteligencie budeme rozumieť napríklad šachový program, ktorý hrá postupne celú partiu a vždy musí potiahnuť do určitého časového limitu, ale nie program, ktorý dostane pozíciu figúrok, vypočíta svoj ťah bez ohľadu na časové obmedzenia a skončí. Vnútorne nemusí byť medzi týmito dvomi programami rozdiel, takže veľa z toho, čo povieme o prvom programe platí aj pre druhý, avšak zdanlivo jemný rozdiel medzi programom, ktorý produkuje výsledok a programom, ktorý produkuje správanie v čase bude pre nás dôležitým obmedzením. Ďalej pod systémom vytvorenom v rámci softwarového inžinierstva budeme prevažne rozumieť systém zložený z viacerých vzájomne komunikujúcich programov bežiacich na jednom alebo viacerých procesoroch, systém založený skôr na princípoch konkurencie (multitasking) než paralelizmu.

2. Inteligencia a komplexnosť

Ako sme v úvode naznačili, žijeme v predstave, že časti umelej inteligencie a softwarového inžinierstva nachádzajúce sa na špici pokroku oboch disciplín idú tou istou cestou a narážajú na tie isté prekážky, navyše bez toho, že by o sebe vzájomne vedeli. Aby aspoň niektorí z čitateľov tohto príspevku boli ochotní túto predstavu zdieľať, vrátime sa na chvíľu k samotným východiskám umelej inteligencie. Budeme sa pritom snažiť zameniť slovo „inteligencia“ za iné slová tak, aby sa spomínané dve disciplíny k sebe priblížili.

Samozrejme, že ciele a teoretické pozadie (neprajníci by na tomto mieste iste poznamenali, že žiadne neexistuje) oboch disciplín sú rôzne, avšak základným

spoločným rysom je, že obe sa zaoberajú konštrukciou určitých systémov, t.j. ich skutočným zrealizovaním. V prípade umelej inteligencie ide o systémy „inteligentné“, v prípade softwarového inžinierstva o „akékoľvek“ či „programové“. Jedným slovom v oboch prípadoch ide o systémy s prívlastkami dosť nejasnými na to, aby sa ich týkali rovnaké implementačné postupy a problémy s nimi spojené.

Čo sa týka programových systémov, je známe, že urobiť malý systém je ľahko možné akýmikoľvek implementačnými postupmi a na žiadny problém pritom nenarazíme. Rovnako možné je urobiť sto malých systémov, ktoré slúžia na rôzne účely. Oveľa ťažšie a často aj nemožné je urobiť jeden rozsiahly systém (čo je aktuálny trend softwarového inžinierstva), ktorý v sebe tých sto malých zahŕňa. Čím komplexnejší systém je potrebné vytvoriť, tým menší okruh implementačných prostriedkov je možné úspešne použiť. Navyše je vývoj a ďalší osud týchto systémov spravidla sprevádzaný množstvom problémov, ktoré majú tendenciu opakovane vyústiť do potreby začať celú prácu odznova. Prečo je taký veľký rozdiel medzi sto malými systémami a jedným veľkým ? Jednou z možných odpovedí je, že prepojenie tých sto malých systémov zabezpečuje človek a k jeho funkčnosti prispieva svojou vlastnou inteligenciou, pričom v komplexnom systéme musí byť tento vklad realizovaný tými istými prostriedkami ako jednotlivé funkcie malých systémov. Komplexnosť teda vyžaduje inteligenciu a to je práve miesto styku softwarového inžinierstva a umelej inteligencie.

Pokúsme sa teraz (bez ohľadu na nekorektnosť takého postupu z hľadiska logiky) obrátiť túto implikáciu. Nevyžaduje inteligencia komplexnosť ? Pozrime sa na vec očami jednej z definícií inteligencie z oblasti psychológie: „Inteligencia je schopnosť systému adekvátne reagovať na zmeny odohrávajúce sa v jeho prostredí“. Tu si hneď treba uvedomiť, že nemožno požadovať od inteligentného systému adekvátnu reakciu na akúkoľvek zmenu jeho prostredia. Musíme množinu spomínaných zmien prostredia obmedziť len na určitú triedu a snažiť sa, aby táto bola netriviálna a čo najširšia. Bude závisieť od šírky a „tvaru“ tejto triedy a od charakteru správania, ktorého produkciu od nášho systému požadujeme, či bude uznaný za inteligentný alebo nie a teda či bude jeho realizácia úspechom umelej inteligencie alebo nie. Môžeme sa na celú vec pozerať tak., že pre každú požadovanú funkčnosť systému existuje určitá latka, ktorú musí šírka a „tvar“ spomínanej triedy preskočiť, aby bol systém, ktorý sa ňou vyznačuje považovaný (ľuďmi) za inteligentný. Existuje teda nejaká spojitá veličina, ktorá je vlastnosťou systému, ktorá ak prekročí určitý limit, systém je inteligentný, inak nie je. (Spomeňme napríklad program ELIZA, ktorý dokázal hrať úlohu psychiatra pri rozhovore s pacientom tak dobre, že ho pacienti jednoznačne označovali za skutočného psychiatra. Pritom tento program napíše každý študent umelej inteligencie za jeden večer. To je vysvetliteľné tým, že spomínaná latka pre správanie psychiatra je tak nízko, že ju aj taký jednoduchý program ako ELIZA dokáže preskočiť.) Takéto nazeranie na vec je zaujímavé preto, lebo inteligentné systémy sú pri ňom v limitnom prípade, keď sa spomínaná spojitá veličina blíži k nule (t.j. v prípade triviálnej triedy zmien, nad ktorou systémy operujú adekvátne), totožné s obyčajnými programovými systémami. To, čo týmto „neinteligentným“ systémom chýba, je teda schopnosť pojať do seba netriviálne širokú triedu zmien a adekvátne nad ňou reagovať. Spomínanou spojitou veličinou nie je teda nič iné než spomínaná komplexnosť.

Medzi inteligenciou a komplexnosťou je teda určitý vzťah a nám sa ponúka možnosť definovať jedno podľa druhého. Keďže inteligencia je vágnejšie slovo než komplexnosť, prijímame pre naše účely nasledovný pohľad:

- **Komplexnosť je schopnosť systému pojať do seba veľa funkcií či správaní bez straty celkovej funkčnosti.** Na prvý pohľad sa zdá, že takú schopnosť musí mať

každý systém. Prax však jasne ukazuje, že to nie je tak. Prečo to tak nie je a čo by museli naše teoretické modely systémov zahŕňať, aby z nich táto skutočnosť vyplývala sa budeme snažiť ukázať nižšie.

- **Inteligencia je dvojhodnotovou kategorizáciou komplexnosti,** t.j. pre každý účel existuje určitý súbor funkcií a správaní, ktoré do seba musí systém pojať, aby bol považovaný za inteligentný. Táto kategorizácia je subjektívnou záležitosťou ľudí, inak povedané inteligencia je len abstraktný pojem, ktorý si vymysleli ľudia ako skratku za „dostatočne komplexný".

- Rozumným cieľom časti umelej inteligencie a softwarového inžinierstva je **hľadať také implementačné postupy, ktoré umožňujú implementovanému systému dosiahnuť čo najvyššiu komplexnosť.**

3. Tradičná a Nová umelá inteligencia

Že to s vložením veľkého množstva vlastností do jedného celku nie je jednoduché, nádherne demonštruje vývoj v umelej inteligencii. Na poli umelej inteligencie sa totiž s komplexnými systémami stretávame už veľmi dlho (oveľa dlhšie než v softwarovom inžinierstve), aj keď tu vystupujú pod hlavičkou inteligentných systémov. Ako sme však už naznačili poriadny inteligentný systém v sebe komplexnosť implicitne zahŕňa. História umelej inteligencie je radom úspechov a hlavne neúspechov. Z nášho pohľadu by sme si trúfli povedať, že ide o rad systémov, ktoré sa vyznačujú určitou mierou komplexnosti: tam, kde sa to s komplexnosťou neprehnalo, dostavil sa úspech, inde neúspech.

Pokiaľ chceme do systému vložiť určitý súbor vlastností máme na to dve riešenia:
- vyjadrovať ich ako dáta určitého všeobecného monolytického algoritmu, alebo
- vyjadrovať ich špecializovanými algoritmami.

Umelá inteligencia siahla najprv po prvej možnosti. Všetky tieto tradičné systémy umelej inteligencie boli založené na predstave, že inteligenciu v systéme zabezpečí určitý špeciálny algoritmus operujúci nad dátami vyjadrenými v jednotnej špeciálnej forme. Pridávanie nových vlastností tu zodpovedá pridaniu ďalších dát, bez zmeny či rozšírenia algoritmu (aspoň v ideálnom prípade), ale za strašnú cenu: za znásobenie času potrebného na prejavenie sa ktorejkoľvek z vlastností. Algoritmus je totiž všeobecný a ku každej vlastnosti musí pristupovať rovnakým spôsobom. Algoritmus nevie čo jednotlivé dáta predstavujú a musí k nim pristupovať len vzhľadom na ich syntaktickú podobu, neuvažujúc ich význam. Prirodzeným následkom toho je, že k sebe vzťahuje všetky dáta, aby na základe porovnania ich syntaktickej podoby odhalil sémantické vzťahy medzi nimi. Keďže len málo veci spolu súvisí, väčšinu času algoritmus strávi pokusmi vzťahovať k sebe sémanticky nesúvisiace dáta. Každý jeden krok v ktorom takýto systém reaguje na vstup svojím výstupom sa vyznačuje exponenciálnou výpočtovou zložitosťou od dĺžky sémantickej väzby medzi dátami, ktorú treba odhaliť. Tým, že očakávame od systému plnenie nejakej náročnej úlohy, dávame mu vlastne odhaliť dlhú reťaz sémantických väzieb medzi jeho dátami, čo vzhľadom na exponenciálny charakter výpočtovej zložitosti tohto procesu, mu nedáva šancu stihnúť to v reálnom čase. Paradoxne rozsah dát, i keď neprispieva k výpočtovej zložitosti ako exponent ale len ako základ, sa ukázal byť ešte ťažším orieškom, lebo v reálnych aplikáciách rastie do vysokých čísiel. V celku možno povedať, že takýmto spôsobom sa dá vyvinúť iba taký

systém, ktorý pracuje s málo faktami a robí nad nimi závery nevyžadujúce dlhé úvahy. V aplikáciách pracujúcich v dynamickom prostredí tu vždy hrozí, že kým systém vypočíta čo podniknúť, tento zámer je už neaktuálny. Celkom prirodzene sa teda môžu vyskytnúť aplikácie, ktoré uvedeným spôsobom vôbec nie je možné zrealizovať (príkladom tu môže byť oblasť mobilnej robotiky).

Časť umelej inteligencie sa preto vydala iným smerom, sledujúc druhú uvedenú možnosť, t.j. používať špecializované algoritmy. I keď začiatky sa týkali jednoduchých systémov, táto nová umelá inteligencia sa tým vydala na cestu vedúcu do príkreho kopca na ktorom číhajú problémy spojené s budovaním komplexných systémov a na ktorý sa snaží nezávisle na nej vygúľať svoj syzifovský kameň softwarové inžinierstvo. Základné prostriedky, ktoré pri lezení na tento kopec používa umelá inteligencia a softwarové inžinierstvo časom splynuli a dali vznik agentom a multiagentovým systémom. Nič menej i v rámci tohto súboru implementačných platforiem existujú jemné a dôležité rozdiely. Podľa názoru autora sa práve na poli umelej inteligencie objavili progresívne myšlienky, ktoré by mohli obohatiť súčasný arzenál vývojových metód softwarového inžinierstva a pomôcť mu s jeho niektorými vážnymi problémami, ktoré sú súhrnne označované názvom „softwarová kríza“.

4. Softwarová kríza

Softwarová kríza sprevádza projekty softwarových inžinierov od doby, kedy im pokrok umožnil budovať distribuované systémy, či už z hľadiska komunikácie medzi procesmi v rámci jedného počítača na báze modernejších operačných systémov alebo v medzi viacerými počítačmi na báze počítačových sietí. Hardware a operačné systémy sa stali vhodné pre budovanie komplexných systémov a samozrejme i záujem o také systémy bol, nakoľko odbremeňujú ľudí nielen od jednotlivých činností, ale od celých oblastí činností. Málokto vtedy pomyslel na to, že postaviť jeden stoposchodový dom je niečo celkom iné ako postaviť sto jednoposchodových. Beztak sa stavali najprv dvoj-, troj-, štvor-poschodové domy a vždy sa na prekonanie nasledovnej výšky našlo nejaké riešenie. Už vtedy však bolo jasné, že ak chce niekto trojposchodový dom prestavať na päťposchodový, musí zbúrať trojposchodový a postaviť päťposchodový. A časom to zašlo tak ďaleko, že treba búrať stoposchodový dom ak je treba postaviť aj strešnú nadstavbu, ba ešte ďalej, že naplánovaný dvestoposchodový dom sa dostavia len do stopäťdesiateho poschodia a aj tak sa od sto dvadsiateho poschodia neodporúča bývať, lebo tam často odpadávajú balkóny. Ale dosť metafor. Vytvorenie každého komplexného systému vyžaduje postupné implementovanie veľkého množstva knižníc a programov, ktoré si medzi sebou vymieňajú množstvo dát. Táto implementácia spočíva vo vytvorení kódov týchto knižníc a programov a v počiatočnom inicializovaní určitej časti dát. Úskalie tohto počínania spočíva v tom, že ho z mnohých príčin nie je možné detailne naplánovať a urobiť jedným dychom ako celok. Robí sa vždy postupne, systém je už vo fáze vývoja podrobovaný rozšíreniam a modifikáciám. V ešte väčšom meradle sa objavia po nasadení systému, keď sa dostane do interakcie s užívateľom. Skúsenosť hovorí, že pri každej modifikácii sa kód znehodnocuje v tom zmysle, že urobiť ďalšiu modifikáciu dá väčšinou viac práce než urobiť predchádzajúcu a že po čase si už nikto netrúfne do kódu niečo pridať – ak si trúfne, výsledok nebude funkčný. Čas od času je teda nevyhnutné miesto modifikácie zahodiť určitú časť kódov, vytvoriť ju odznova a ostatné kódy prispôsobiť vzhľadom na túto modifikáciu.

Metódy boja proti tomuto neduhu sa v softwarovom inžinierstve zakladajú prevažne na fakte, že čím viac pozornosti sa kódovaniu venuje pri vytvorení kódov, tým

pomalšie sa znehodnocujú. Za veľmi perspektívne sa považujú metódy umožňujúce časť tejto práce preniesť na vývojové prostriedky, čo umožní v konečnom dôsledku vynaložiť do kódovania viac práce. Ukazuje sa však, že v žiadnom prípade nedokážu tieto techniky znehodnocovaniu zabrániť, dokážu ho iba spomaliť. Aktuálnou sa stáva teda otázka, či by nebolo možné vynájsť inú špecifickejšiu podobu kódov, ktorá by takémuto znehodnocovaniu nepodliehala.

Je teda možné hľadať rôzne podoby pre kódy (väčšinou uvaľovaním určitých obmedzujúcich podmienok), skúšať na ich základe vytvárať systémy a viacmenej intuitívne vyhodnocovať ktorá forma kódov umožňuje budovať čo najkomplexnejšie systémy. Vedeckejšie by však bolo nájsť teóriu kódovania programových systémov, ktorá by dokázala zachytiť pozorované znehodnocovanie a o navrhovaných architektúrach exaktne ukázať nakoľko sa ich znehodnocovanie dotýka. Žiaľ, takú teóriu sa dosiaľ nepodarilo nájsť. Domnievame sa však, že je možné ju navrhnúť, pričom kľúčovú úlohu by v nej mal hrať fakt, že pri modifikáciách kódov musí programátor často zaznamenať v syntaktickej podobe nielen priame sémantické vzťahy, ale aj vzťahy získané na základe tranzitívnosti. Akonáhle sa programátor dostane pri kódovaní do pozície, keď musí strážiť i nepriame vzťahy, je len otázkou času, kedy mu to prerastie cez hlavu. Táto teória by teda mala zachytiť, že:

- pokiaľ vzťahy nezabudujeme explicitne, dostaneme systém, ktorého kódová podoba nepodlieha znehodnocovaniu, avšak platíme za to katastrofálnou výpočtovou zložitosťou (pochádzajúcou z toho, že si tie vzťahy musí systém sám spočítať)

- **pokiaľ vzťahy zabudujeme explicitne, dostaneme systém, ktorého kódovú podobu postihuje znehodnocovanie, tým väčšie čím nepriamejšie sémantické vzťahy musíme explicitne syntakticky vyjadrovať.**

Zdá sa byť teda nádejné hľadať takú architektúru, v ktorej programátor sleduje len priame vzťahy a prípadne (a kacírska myšlienka je tu) na sledovanie mnohých vzťahov vo svojich kódoch úplne rezignuje. Samozrejme vyvstáva tu otázka ako potom môže zabezpečiť, že jeho výtvor bude robiť to, čo bolo plánované. Nuž na to musí použiť inú techniku: miesto sledovania vzťahov v kódoch a následnej, často pochybnej, predikcie ich správania, môže sledovať globálne správanie systému počas jeho ladenia pri jeho inkrementálnom vývoji. Aby však toto správanie bolo dostatočne transparentné a odzrkadlovalo celkové prejavy systému, musí byť inkrementálny vývoj podstatne ináč organizovaný, čo má dopad aj na vnútornú štruktúru, nešetriac pritom ani niektoré vžité programátorské pravidlá.

Práve v týchto ohľadoch spočíva genialita **subsumpčnej architektúry** (Brooks 91) použitej v mobilnej robotike v druhej polovici 80-tych rokov. V oblasti software síce tiež nastal búrlivý rozvoj podnietený tými istými myšlienkami, ktoré viedli k vynájdeniu subsumpčnej architektúry a tento vývoj viedol k vzniku nepreberného množstva multiagentových systémov, ale bez tejto dôležitej jemnôstky.

5. Multiagentové systémy

Multiagentové systémy vznikli na rozhraní takmer všetkých disciplín majúcich niečo dočinenia s počítačmi. Základným hnacím impulzom bola snaha o tvorbu modulárnych systémov, ktoré by boli schopné zachytiť štruktúrovanosť procesov v nich prebiehajúcich. V umelej inteligencii sa to prejavilo obdobím produkujúcim heslá proti

reprezentácií, za budovanie jednoduchších ale reálne fungujúcich systémov, za alternatívne dekompozície systémov, či používania metódy zdola – nahor pri ich tvorbe. S určitým odstupom to však môžeme chápať ako snahu o hľadanie takej modulárnej štruktúry, ktorá by zabezpečila:

- **aby boli k sebe vzťahované iba sémanticky príbuzné dáta**
- **aby jednotlivé sémantické vzťahy zabezpečovali špecializované algoritmy**

Tieto sledované vlastnosti priviedli časť umelej inteligencie do úzkeho kontaktu s komunikáciou medzi procesmi, konkurenčným objektovo orientovaným programovaním a inými formami modularity decentralizovaných programových systémov. Umelá inteligencia sa včlenila do prúdu, ktorý dal vzniknúť veľkému množstvu frakcií s rôznym stupňom relevantnosti k umelej inteligencii. Hranice medzi týmito frakciami sú také nejasné a terminológia nimi používaná tak variabilná, že ťažké je ich čo i len rozumne kategorizovať. Problém je ich aj spoločne pomenovať: Nová UI, Distribuovaná UI, sociorobotika, softwaroví agenti, či multiagentové systémy sú skôr názvy častí než celku, obsahujúce navyše k nášmu ponímaniu problematiky irelevantný balast. Pokiaľ si máme predsa len vybrať určitý spoločný názov, prikláňame sa k názvu „multiagentové systémy“. Väčšina týchto frakcií sa zamerala na pre ňu vlastnú časť prostriedkov, ktorými sa snaží obohatiť tradičné softwarové či robotické produkty a povýšiť ich na nejakú vyššiu kvalitu. Niektoré kladú dôraz na normalizáciu komunikácie medzi procesmi (KQML, FIPA), iné na mobilitu algoritmov (Telescript), ďalšie na softwarové (DESIRE, BDI) či robotické architektúry (subsumpčná architektúra). Každá z týchto frakcií ponúka určitú **architektúru**, t.j. určitý typ základných stavebných kameňov, pravidlá pre zlúčenie ich jednotlivých prejavov do celkového prejavu a metódu ako z nich tvoriť systém, vyznačujúci sa určitými zaujímavými vlastnosťami.

Agenti

Nazeranie na základný stavebný kameň systémov je v rámci všetkých frakcií zhruba rovnaké. Svorne ho nazývajú **agent** a bez ohľadu na to, čo si na ňom najviac cenia (autonómnosť, vnútornú stavbu, mobilitu, komunikačné rozhranie, …) rozumejú pod ním **niečo, čo neustále (resp. opakovane) vníma svoje prostredie, na základe toho volí akcie, ktoré má vo tomto prostredí vykonať, aby sa dosiahol stanovený cieľ a následne tieto akcie vykonáva** (mod. Doran 93). Pritom pod prostredím sa rozumie nielen napr. fyzické okolie mobilného robota, ale čokoľvek, k čomu môže agent pristupovať bez následného zablokovania jeho riadenia. Pri interakcii (t.j. vnímaním a vykonávaním akcií) agenta s prostredím môže ísť spravidla o

- prijímanie signálov zo senzora (vstupného zariadenia) alebo vysielanie signálov na aktuátor (motor) (výstupné zariadenie)
- (**priamu**) **komunikáciu** s inými agentami, t.j. príjem alebo vyslanie správy
- čítanie a zápis zdieľaných dát (**nepriamu komunikáciu**), realizovanú obyčajne ako vyslanie správy a príjem odpovede na špeciálny proces stelesňujúci prostredie

Pritom vo všetkých prípadoch pristupujeme ku každej vstupnej informácii ako keby ju agent vnímal určitým receptorom a ku každej výstupnej ako keby agent vykonal nejakú akciu určitým efektorom. Z hľadiska vnútra agenta, ktoré zabezpečuje voľbu akcie, má teda vstup i výstup rovnakú povahu bez ohľadu na konkrétny spôsob interakcie.

Spôsob, akým je toto vnútro agenta realizované, t.j. povaha algoritmu, ktorý volí akcie, určuje druh agenta. Na prvý pohľad sa môže táto povaha zdať nepodstatná, nakoľko agent je formalizovateľný napr. funkciou, ktorá histórii vstupov priraďuje určitý výstup. Nám však nejde o skúmanie výpočtovej sily alebo inej podobnej vlastnosti uvažovaného systému, ale o hľadanie spôsobov, ako ho tvoriť (programovať, vyvíjať). Spôsob voľby akcie je z tohto pohľadu kľúčový. Niektoré frakcie vkladajú do agenta celý tradičný systém a „agentovosť" sa tým pádom stáva iba nástrojom normalizovanej komunikácie rôznych tradičných systémov. Iné obmedzujú tvar vstupov a výstupov vstupujúcich do procesu voľby, t.j. definujú určitý jazyk, či protokol. Ďalšie zase definujú tvar voliaceho algoritmu, uvaľujú naň určité obmedzenia, prípadne používajú jediný pevný algoritmus v rôznych konfiguráciách[1].

Ďalším dôležitým parametrom agenta je spôsob akým je do jeho vnútra zahrnutý cieľ. Pokiaľ si agent buduje na základe histórie vstupov určitú reprezentáciu a používa na to jednotný vyjadrovací prostriedok, je možné týmto prostriedkom **explicitne** vyjadriť aj cieľ. Pre takéhoto agenta sa ujalo pomenovanie cieľavedomý, t.j. **deliberatívny agent** a je veľmi obľúbenou a rozšírenou platformou (najmä v rámci distribuovanej UI) (Goodwin 93). Explicitné vyjadrenie cieľa mu umožňuje voliť akciu napríklad výberom zo všetkých možných akcií na základe predvídania ich následkov. To je na jednej strane v zhode s tým ako prebieha (resp. ako si predstavujeme, že prebieha) riešenie problémov v našej mysli, avšak na strane druhej od tradičných systémov umelej inteligencie sa to líši iba v tom, že v rámci jednotlivých agentov je možné používať rôznu reprezentačnú platformu a špecializovať algoritmy predvídania následkov akcií. Práca systému zloženého z takýchto jednotiek v reálnom čase je tiež dosť otázna. Ide tu o to, nakoľko agent následky možných akcií domyslí.

Ak začne kvalitu voľby akcie posudzovať podľa ďalšieho možného vývoja a bude zisťovať, či skutočne povedie k cieľu, nemôže byť o práci v reálnom čase ani reči. Navyše to bude v rozpore s tým, že navrhujeme za základný stavebný kameň inteligentných systémov niečo, čo vie riešiť problémy (t.j. postaviť adekvátny plán na základe vnútorných virtuálnych pochodov) a pritom poznáme veľa živých systémov, ktoré sa neštítime nazývať inteligentnými a riešiť problémy (v nami používanom užšom význame slova) vôbec nevedia. V istých prípadoch by teda celok mal mať menšie schopnosti než jeho časti. Zaujímavé by bolo dosiahnuť skôr opak.

Ak deliberatívny agent vyhodnotí iba určité kvantitatívne ukazovatele priamych následkov, zrejme ho to príliš nezdrží, avšak na druhej strane sa cieľ preňho stáva menej dôležitý než súbor trikov na predvídanie následkov akcií. Pokiaľ bude tento súbor dostatočne špecializovaný, bude dobre fungovať iba pre tento cieľ a pre nijaký inakší. Spomínané triky sa v hraničnom prípade stanú implicitným vyjadrením explicitne vyjadreného cieľa. Pokiaľ si prestanú všímať cieľ úplne, agent prestáva byť deliberatívnym a stáva sa agentom, pre ktorého sa ujalo pomenovanie reaktívny.

Reaktívni agenti sú založení na predstave o adekvátnych podvedomých reakciách. Reaktívny agent pracuje spôsobom „urobím toto, lebo som v situácii, kedy sa to robieva" na rozdiel od deliberatívneho, pre ktorého platí „urobím toto, lebo chcem dosiahnuť tamto a takto to (snáď) dosiahnem". Hovorí sa, že kým deliberatívny agent sa „rozhoduje", reaktívny „reaguje" (Kelemen 94) – i keď tu v podstate ide spojité spektrum, ktoré sa polarizuje iba v rámci snahy o kategorizáciu.

---

[1] Posledne menované môže sa zdať v rozpore s druhou požadovanou vlastnosťou, t.j. aby jednotlivé sémantické vzťahy zabezpečovali špecializované algoritmy. Tieto špecializované algoritmy však nemusíme ztotožňovať s algoritmami pre voľbu akcie. Niekoľko agentov používajúcich ten istý algoritmus na voľbu akcie môže spoločnými silami realizovať špecializovaný algoritmus. Požadovaná špecializovanosť sa teda nemusí objaviť v základných stavebných kameňoch systému, ale až pri stavbách z nich postavených.

Reaktívny agent je iba súborom trikov definujúcich, čo sa za akých okolností robieva na dosiahnutie jediného cieľa. O tomto cieli nemá ani potuchy, lebo nemá potuchy vôbec o ničom. Na receptoroch (vstupoch) neustále prijíma signály, na ktoré sa vrhá algoritmus pevne nadrátovaný v jeho vnútri, následkom čoho sa produkujú signály na jeho efektory (výstupy). Na tento algoritmus je možné klásť ďalšie ohraničenia. S jednoduchosťou voľby akcie sa pritom dostávame na hranice možností, kde zdanlivo hrozí, že z tak jednoduchých elementárnych jednotiek bude problém vybudovať zložitý systém, t.j. že ak to vôbec bude možné, bude to ťažšie než pri použití zložitejších elementárnych jednotiek. Jedno z týchto ohraničení je používanie vnútorného stavu agenta. Algoritmus voľby akcie normálne môže používať globálnu pamäť, ktorá mu umožňuje vziať do úvahy nepriamo aj hodnoty vstupov, výstupov a vnútorných stavov z minulosti. Pokiaľ to zakážeme, agent si nemôže pamätať nič z predchádzajúcich krokov. Pamäť môže využiť len v jednom kroku ako jednorázovú pomôcku. Takýto okliešťený agent sa zvykne nazývať **čisto reaktívnym.**

Metódy tvorby

Význam každého stavebného kameňa nespočíva ani tak v tom, čo sa dá z neho postaviť, ale hlavne ako sa dá z neho stavať. Teoreticky to totiž poväčšine vychádza tak, že z každého typu kameňa sa dá postaviť akýkoľvek dom. Prakticky však každá stavba naráža na určité ohraničenia vyplývajúce zo spôsobu ako sa dom stavia. Na každej architektúre je vlastne podstatnejšia metóda tvorby než typ agenta, na druhej strane musí typ agenta danú metódu podporovať (jednotlivé syntaktické štruktúry reprezentujúce agentov sú síce väčšinou vzájomne konvertovateľné, avšak nie každá syntax je rovnako vhodná na používanie tvorcom systému). Metóda tvorby multiagentových systémov pritom spočíva vo zvolení postupu ako budovať vnútro systému, aby na povrchu produkoval požadované správanie. Od tohto postupu očakávame, že nám napovie ako systém dekomponovať (t.j. ako ho pri návrhu rozdeliť na jednotlivé časti) a ako tieto časti postupne implementovať. Pritom pod časťou rozumieme určitú skupinu kódov a dát organizačne izolovanú od ostatných. V prípade multiagentových systémoch ide o teda o kódy a dáta prislúchajúce určitej skupine agentov.

Pri dekompozícii systému je podstatné, ktoré atribúty budú dáta a kódy systému spájať a ktoré oddeľovať. V tradičných systémoch jednotlivé časti tvorili spravidla algoritmicky príbuzné dáta a kódy – išlo o dekompozíciu funkciou. Inou možnosťou je orientovať sa na sémantickú príbuznosť kódov a dát. Dostaneme tak systém rozdelený na jednotlivé aktivity – **dekompozíciu aktivitou**. Dekompozícia funkciou má výhodu v tom, že môže byť rovnaká pre dva systémy vykonávajúce rôznu činnosť a teda je možné poznatky získané pri jednom systéme znovu použiť pri systéme druhom. Naproti tomu dekompozícia aktivitou je podobná iba ak systémy vykonávajú podobné činnosti. Vedieť správne dekomponovať systém na aktivity je spravidla umenie. Sila tejto dekompozície však spočíva v tom, že podporuje inkrementálnu implementáciu navrhnutých častí. Umožňuje zamerať sa najprv na základné aktivity a potom postupne ďalšie aktivity pridávať. Pritom systém je vždy po pridaní ďalšej aktivity spustenia schopný a vieme aké správanie by mal mať pri daných implementovaných aktivitách – môžeme ho teda ladiť ako celok. Tým pádom nám nehrozí, že keď jednotlivé (bárs aj osobitne odskúšané) časti systému spojíme, vzniknutý celok nebude funkčný. Toto nešťastie je charakteristické práve pre systémy, pri tvorbe ktorých bola použitá dekompozícia funkciou.

Čo sa týka implementácie navrhnutých častí, t.j. ich nakódovania, panuje v programátorskom svete jednotný názor: správny postup je zhora – nadol. Implementácia určitých typov multiagentových systémov je však náchylná otriasť i touto

pravdou. Každý komplikovanejší systém sa totiž tvorí **inkrementálne**. Pokiaľ v ňom nejaká časť využíva prítomnosť časti druhej, pri metóde zhora nadol bude najprv implemntovaná časť, ktorá využíva a až potom časť, ktorá je využívaná. Pokiaľ teda chceme systém ladiť, musíme využívanú časť dočasne nahradiť nejakým triviálnym modulom. Takýto celok však nebude vykazovať žiadne relevantné správanie a preto takéto ladenie nič nepovie o funkčnosti implementovanej časti. Ak však použijeme metódu **zdola – nahor**, ladenie je plne umožnené. Samozrejme tu však hrozí nebezpečenstvo, že pri návrhu neodhalíme všetky potrebné detaily na to, aby neskôr implementovaná časť dokázala využiť skôr implementovanú. Toto nebezpečenstvo môžeme odstrániť tým, že z princípu nebudeme implementovať určitú časť kvôli tomu, aby ju nejaká iná v budúcnosti využívala, ale len kvôli samotnej aktivite, ktorú zabezpečuje. Nebudeme vopred chystať žiadny interface pre časti vyvíjané neskôr, napr. nebudeme robiť žiadne knižnice. Pokiaľ potom budeme chcieť pri implementácii ďalšej časti využiť inú existujúcu časť, musíme toto využitie zrealizovať nie tým, že nová časť bude existujúcu časť „volať“, ale tým že nová časť sa „votrie“ do jej činnosti, t.j. bude infiltrovať externé dáta používané existujúcou časťou. Keď to prevedieme do reči multiagentových systémov, nová skupina agentov bude infiltrovať prostredia stávajúcich agentov. Pritom k tejto infiltrácií nesmie dochádzať neustále, nakoľko by to v podstate vyradilo existujúce časti z ich pôvodnej činnosti. Avšak počas situácie na ktorú pôvodné časti nevedia adektvátne zareagovať, je infiltrácia želaná, ak sa jej pôsobením spomínaná adekvátna reakcia zabezpečí. Takéto pripojenie novej časti do systému v zásade neovplyvní podiel stávajúcich častí na správaní systému, iba do neho zavedie určité výnimky. To nám umožňuje pri inkrementálnej tvorbe po každom pridaní novej časti systém testovať a ladiť. Tým získavame istotu, že pri tvorbe sledujeme výsledné želané správanie systému – že postupujeme správnym smerom a nestane sa nám, že po ukončení tvorby bude systém robiť niečo iné než sme pôvodne zamýšľali.

Tento spôsob, ktorý používa subsumpčná architektúra je len jedným z mnohých spôsobov ako dosiahnuť, aby sa na povrchových prejavoch systému podielala vždy správna zostava jeho častí. Umenie vytvoriť multiagentový systém totiž spočíva práve v tom, ako zariadiť, aby sa **v každom okamihu aktivovali tí agenti, ktorých zlúčený prejav dáva na povrchu systému želané správanie** (Minsky 86). Pritom je podstatné ako sú prejavy aktivovaných agentov (t.j. tých, ktorí práve zvolili a vykonávajú určité akcie spôsobujúce zmenu v ich prostredí) zlúčené do celkového prejavu systému. Spravidla to vyzerá tak, že pokiaľ sa v prostredí systému nič nedeje, v systéme prebiehajú len najzákladnejšie aktivity. Tie sú potlačené vyššími v okamihu, keď príde určitý podnet. Za istý čas po jeho odznení, keď už je tento podnet „vybavený“, vyššie aktivity opäť ustúpia do ústrania. Aktivity teda pracujú synchrónne, je medzi nimi zavedená prísna hierarchia. Ešte zaujímavejšie by podľa nás bolo (to je aj v podstate jediná modifikácia subsumpčnej architektúry, ktorú presadzujeme), keby hierarchicky nižšia aktivita mala šancu odolať potlačeniu zo strany hierarchicky vyššej aktivity, akurát by to bolo málo pravdepodobné. Napríklad, keď sme hladní, ale veľmi túžime ísť na seminár z umelej inteligencie, dokážeme tento hlad potlačiť tak, že celé minúty o ňom nevieme. On sa však bude v pravidelných intervaloch hlásiť o slovo ako aj predtým – nám sa iba podarí predĺžiť tieto intervaly zo sekúnd na minúty a tak znížiť pravdepodobnosť toho, že si tento hlad všimne časť našej mysle, ktorá ma na starosť odchod na obed.

Zaujímavé vlastnosti

Ako sme naznačili, multiagentové systémy majú istý vzťah k umelej inteligencii, ale často majú oveľa bližšie k iným oblastiam. Ich „verejným“ cieľom nie je vytvoriť

inteligentné systémy, ani komplexné programové systémy, ale systémy, ktoré sa vyznačujú určitými zaujímavými vlastnosťami. Pritom ide väčšinou o kvalitatívne vlastnosti týkajúce sa spôsobov tvorby systémov (čo si samozrejme vyžaduje aj adekvátnu vnútornú štruktúru), ktoré umožňujú rozšíriť limity pre ich kvantitatívne parametre. Pokúsime sa vymenovať aspoň najčastejšie uvažované vlastnosti:

- **decentralizovanosť systému**: V tradičných systémoch je každý proces naviazaný na procesy nižšej úrovne, tie na procesy ešte nižšej úrovne, … až v konečnom dôsledku na zariadenia. Toto naviazanie má taký charakter, že samotný proces bez nižších vrstiev nedokáže vyvíjať vlastnú aktivitu. Proces spravidla požiada nižšie vrstvy o určitú službu (napríklad „chcem čítať znaky zo sériovej linky“) a potom je už odkázaný na to, že mu tieto vrstvy dajú určitý podnet k jeho činnosti („prišiel ti znak, prečítaj si ho“). Zdá sa, že to tak musí byť, lebo proces by musel inak čítať znaky neustále – cyklil by sa a vyčerpával by všetku výpočtovú kapacitu procesora. „Neustále“ však stačí interpretovať ako dostatočne často. (Ak je buffer na príjem znakov 1kB, a rýchlosť linky 9600 bps, stačí procesu raz za 0.5 s prečítať všetky znaky z buffra – a nepotrebuje aby ho niekto upozorňoval, že mu nejaké znaky prišli.) Pritom už procesor nemusí zaťažiť vôbec a nižšie vrstvy volá „z vlastnej vôle“ – nie je centrálne riadený. Aplikovaním takejto zmeny na všetky časti systému sa dá dosiahnuť, že medzi procesmi v systéme sa nachádzajú iba „služobníci“ a žiadni „riaditelia“. Aj samotné časovanie procesu je iba služba – nikto procesu nehovorí „teraz sa zobuď“, ale proces sám vraví „teraz chcem pol sekundy spať“.

- **práca v reálnom čase**: Je ľahké si všimnúť, že v predchádzajúcom príklade sme zamlčali skutočnosť, že je nevyhnutné, aby bol v systéme niekto, kto vie vykonať žiadosť „chcem pol sekundy spať“ tak, že to bude naozaj 0.5 s a nie 2 s. Na to je potrebné, aby žiadny proces neokupoval permanentne procesor. Musí teda ísť o systém reálneho času. Ďalším zamlčaným faktom je, že decentralizované procesy vykonávajú množstvo jalovej práce (v našom príklade ide o čítanie z linky aj vtedy, keď z nej žiadne znaky neprichádzajú). Tu si treba uvedomiť, že táto jalová práca neznižuje možnosti vykonávania užitočnej práce (systém musí byť totiž dimenzovaný tak, že tie znaky môžu prichádzať kedykoľvek, t.j. aj stále).

- **normalizácia komunikačných rozhraní**: Komunikácia medzi procesmi v programových systémoch je spravidla založená na známom vzťahu klient - server. V týchto aplikáciách poskytuje obyčajne každý server sebe vlastné komunikačné rozhranie pre svojich klientov. Klient sa potom na každý server musí obrátiť spôsobom, ktorý je vlastný danému serveru, t.j. na každý server iným spôsobom. Pritom kódy v serveri zabezpečujúce komunikáciu s klientami sú veľmi podobné. Vzniklo preto niekoľko pokusov normalizovať komunikáciu medzi klientom a serverom (napr. CORBA). Z nich sú pre nás zaujímavé tie, ktoré dekomponujú klientov a serverov na agentov a ich prostredia.

- **zníženie nárokov na výpočtovú a prenosovú kapacitu**: V rámci architektúry multiagentového systému je potrebné sledovať, aby od počtu agentov rástla výpočtová zložitosť zlúčenia ich prejavov do výsledného prejavu systému v najhoršom prípade lineárne. T.j. metóda jeho tvorby musí dôsledne podporovať implementáciu sémantických vzťahov, aby bola interakcia medzi agentami iba lokálna. Pokiaľ napríklad agenti používajú priamu adresnú komunikáciu, nesmie autor zaviesť v systéme komunikáciu každý s každým, pričom každý zahodí

z prijatých správ tie, ktoré nie sú preň podstatné. Významným faktorom zlúčenia prejavov agentov je taktiež prenosová kapacita, obzvlášť vtedy, keď je programový systém vykonávaný na vzdialených procesoroch. Ak z hľadiska sémantickej povahy veci určitý agent nepotrebuje spojenie so vzdialeným agentom permanentne, môže byť výhodnejšie dočasne presťahovať na vzdialený počítač tohto agenta, než komunikovať na diaľku (pripomíname, že toto by nebolo možné urobiť s centrálne riadeným procesom, respektíve bolo by to možné iba za strašnú cenu).

- **procesná modularita**: Dômyselné použitie určitých typov agentov môže mať následok, že v systéme vznikne oveľa viac (a menších) procesov než by tomu bolo v analogickom programovom systéme (pokiaľ by taký bolo možné reálne vytvoriť).To sa môže kladne prejaviť napr. na znovuvyužiteľnosti kódov. Táto síce funguje iba na princípe zhody, avšak zhody tak malých a jednoduchých častí, že je jej použitie dosť pravdepodobné. Čím jednoduchšie ciele priradíme k agentom, tým je potenciálne všetkých možných cieľov (a následne kódov agentov) menej a častejšie sa vyskytujú. Máme tu dočinenia s novým druhom modularity systémov.

- **zotaviteľnosť**: Pri tradičných systémoch často reálne hrozí, že narazia na prekážku, ktorú nedokážu zdolať. Napr. nejaký proces, ktorý plní určitú dôležitú službu má v sebe chybu (s čím sa musí reálne počítať) a za istých okolností spadne. Tento je možné znovu spustiť a ak už pominul podnet, ktorý vyvolal pád, proces bude ďalej bežať. Ale ak procesy, ktoré jeho službu používali, sa naň odkazovali jeho adresou, táto už bude neplatná, a zreštartovaný proces bude pre ne nepoužiteľný. Použitie nepriamej komunikácie (t.j. komunikácie odkazom cez prostredie) v multiagentových systémoch umožňuje, aby sa procesy využívajúce službu chybného procesu o jeho páde a reštartovaní vôbec nedozvedeli. Je taktiež možné vytvoriť zálohovanie služieb, keď jednu a tú istú službu vykonávajú dvaja rôzni agenti rôznym spôsobom, ktorý dáva zhruba tie isté výsledky. Pritom užívatelia služby nevedia koľko procesov ju v skutočnosti realizuje.

- **robustnosť**: Môže sa taktiež stať, že pri implementácií nejakého procesu niečo prehliadneme, následkom čoho nebude určitá služba v systéme pre danú situáciu adekvátna. V podstate tu ide tiež o chybu, len nespôsobuje pád procesu. Architektúra systému by pre tento prípad mala zabezpečiť, že sa táto chyba natrvalo nevypropaguje do vonkajších prejavov systému. Malo by to spôsobiť buď žiadne, alebo len chvíľkové zaváhanie v správaní systému.

- **emergencia**: Už sme naznačili, že medzi jednotlivými typmi agentov uprednostňujeme tie jednoduchšie. Je takýto postoj oprávnený? Napríklad už na prvý pohľad je jasné, že reaktívny a tobôž čisto reaktívny agent nedokáže sledovať zložitejšie ciele. To je aj pravda, ale reaktívni a dokonca aj čisto reaktívni agenti to dokážu. Ich sila spočíva v schopnosti vytvárať celky, ktoré prejavujú vlastnosti, ktoré oni sami prejaviť nevedia. Väčšinou na tom nie je nič svetoborné a ľahko si je predstaviť syntaktickú transformáciu spôsobujúcu prenos podstaty týchto vlastností z nižších úrovní systému do vyšších.

  Niektoré typy deliberatívnych agentov napríklad vedia riešiť problémy. Reaktívni agenti to nevedia. Ale je ľahké predstaviť si, že z deliberatívnych agentov vytiahneme všetky reprezentačné štruktúry do ich prostredia (toto sa dá urobiť len vtedy, keď agenti používajú nepriamu komunikáciu – každá priama komunikácia sa

však dá transformovať na nepriamu) a proces logického odvodzovania zrealizujeme ako reakcie teraz už reaktívnych agentov na hodnoty týchto štruktúr.

Podobne reaktívni agenti majú vnútorný stav. Čisto reaktívni ho nemajú. Ale opäť nie je nič jednoduchšie, než vyhodiť globálnu pamäť z reaktívneho agenta do jeho prostredia. Bude si ju musieť akurát zakaždým z tohto prostredia prečítať a zmenenú ju tam zapísať.

Medzi jednotlivými architektúrami sa dá teda prechádzať. Otázne je, čo sa tým získa. Jednou z nádejí je, že pri určitých architektúrach sa po implementácií n-vlastností v správaní systému, objaví želaná (n+1)-vá, a my budeme prekvapení a vzrušení kde sa tam sama od seba vzala. Tento jav sa nazýva emergencia – samoobjavenie sa. Je otázne, či emergenciou môže vzniknúť niečo naozaj zaujímavé. Isté však je, že naše vzrušenie sa veľmi rýchlo stratí, lebo keď si položíme otázku „kde sa tá vlastnosť vzala?“, zrejme si na ňu budeme vedieť odpovedať. Pri emergencií máme teda dočinenia skôr s neschopnosťou tvorcu vidieť vopred určité sémantické vzťahy a schopnosťou architektúry automaticky urobiť nad explicitne danými priamymi sémantickými vzťahmi tranzitívny uzáver.

Bez ohľadu na individuálny význam týchto vlastností sa domnievame, že ich sledovanie je púhym prostriedkom na dosiahnutie rozsiahlej komplexnosti systémov a to za účelom naplnenia vedomej či nevedomej snahy o vytvorenie inteligentných programových systémov. Multiagentové systémy sú, zdá sa, vhodnou implementačnou platformou pre takéto systémy. Medzi jednotlivými odrodami multiagentových systémov sú však isté rozdiely, čo sa tejto vhodnosti týka. Ideálnou odrodou sú podľa autora tohto príspevku **multiagentové systémy zložené z čisto reaktívnych agentov používajúcich nepriamu komunikáciu cez prostredie, dekomponované aktivitou a implementované inkrementálne metódou zdola-nahor** (Lúčny 1997). Tieto systémy sú prirodzeným vylepšením tradičných programových systémov. Vylepšením, ktoré prináša pokrok pri všetkých spomínaných zaujímavých vlastnostiach (modulárnosť, otvorenosť, modifikovateľnosť, konfigurovateľnosť, …).

Tradičné vzťahy klient - server zo súčasných programových systémov sú v týchto systémoch jednoduchou syntaktickou transformáciou prebudované na princípe decentralizovaného systému. Ide o to, že server sa dekomponuje na časť, ktorá slúži ako komunikačné rozhranie pre klientov (túto nazveme v zhode s „multiagentovou“ terminológiou prostredie) a na jednotlivé obsluhy služieb, ktoré sa stanú samostatnými procesmi – agentami. Klienti sa stanú rovnakými agentami ako služby. Prostredie, ktoré vzišlo zo servera, musíme ďalej podriadiť pevne definovanému komunikačnému rozhraniu, ktoré bude rovnaké pre agentov bez ohľadu na to, či vzišli zo služieb servera alebo z klientov. Z každého servera teda vzíde rovnaké prostredie a klienti, ktorí dovtedy používali rôzne rozhrania pre rôznych serverov vystačia s jediným rozhraním pre prostredie.

Agenti v systéme nebudú tým pádom komunikovať priamo, ale iba odkazmi, ktoré budú zanechávať v prostrediach. Týmto odkazom budú rozumieť iba tí agenti, ktorí ich do prostredia zapisujú, alebo ich odtiaľ čítajú. Pre prostredie to budú nič neznamenajúce buffre určitej dĺžky.

Agenti o sebe navzájom nič nevedia, čo ich núti žiť typickým agentím životom: každý agent sa musí opakovane pozerať do prostredia a na základe aktuálnych hodnôt odkazov tam zapisovať iné odkazy. Prostredie by síce agentovi dokázalo hovoriť, že nastala určitá zmena v odkazoch, čo by mu umožnilo nevykonávať jalovú prácu, ale kazilo by to decentralizačný charakter architektúry. Navyše, pokiaľ by agent chcel využiť zmeny v prostredí, musel by mat globálnu pamäť (aby si pamätal predchádzajúcu

hodnotu). Taktiež jeho tvorcu by to mohlo zviesť k písaniu čo najefektívnejšieho kódu, ktorý by sa snažil neprepočítavať to, čo by sa beztak nezmenilo a kód by sa tak stával zložitejším. Náš systém však každopádne musí počítať s tým, že sa všetky odkazy, týkajúce sa agenta, môžu zmeniť naraz. Preto si môžu naši agenti dovoliť zakaždým prečítať všetky svoje odkazy z prostredia, spracovať ich jednotným kódom a zapísať do prostredia všetky vypočítané odkazy pre iných agentov (a prípadne aj pre seba). Výhoda čisto reaktívnych agentov spočíva v tom, že žiadny iný prístup neumožňujú a tým nedovoľujú tvorcovi pustiť sa na bočné chodníčky. Otázne je iba to, či je prípustné zdržanie spôsobené časovým rozdielom medzi zmenou v prostredí a najbližším zobudením agenta. Prostredie je samozrejme schopné zobudiť cez tzv. trigger agenta pokiaľ sa zmení niečo, o čom agent vopred deklaroval, že je preňho dôležité. Je však otázne, či sú tieto triggery nevyhnutné, alebo či sa bez nich možno zaobísť – lepšia by bola druhá možnosť.

Keďže prostredia sú všetky (okrem tých, ktoré stelesňujú prepojenie na fyzické prostredie) rovnaké (a to nielen v rámci jedného systému) a krajne jednoduché, je dosť pravdepodobné, že ich dokážeme napísať bez chyby. Agenti naopak môžu chyby obsahovať a môžu napríklad padnúť. Je však možné ich zreštartovať bez toho, že by sa museli reštarovať akékoľvek ďalšie procesy. Z toho vyplýva veľká schopnosť zotaviteľnosti systému.

Skutočnosť, že všetka pamäť agentov sa nachádza v prostrediach sa veľmi pozitívne prejaví na konfigurovateľnosti systému. Z každej konštanty v systéme sa totiž musí urobiť odkaz. Tento sa zdanlivo celkom zbytočne pri každom kroku agenta číta. Zbytočné je to však len do chvíle, kedy sa zistí že túto konštantu treba meniť počas behu systému. Táto zmena, v obyčajných systémoch bez globálneho reštartu veľmi komplikovaná, je tu zadarmo.

Keďže čisto reaktívny agent má všetku svoju pamäť vyhodenú do prostredia, môže ju ktorýkoľvek iný agent infiltrovať a niekedy tak celkom pozmeniť funkciu agenta. (Ideálne by tu bolo, keby bolo možné v agentovi zmeniť takýmto spôsobom nielen dáta ale aj kód, napr. sčítanie na násobenie – tak ďaleko však ešte nie sme.) To umožňuje širokú mieru otvorenosti a modifikovateľnosti systémov.

Modifikovateľnosť ide pritom ruka v ruke s inkrementálnou implementáciou. Práve na základe infiltrácie odkazov v prostrediach sa môže vyššia a neskôr implementovaná vrstva votrieť do činnosti nižších vrstiev. Pritom je nádej, že také postupy rozširovania a modifikácie systémov budú len veľmi pomaly neznehodnocovať kódy, lebo tieto kódy sa v podstate nebudú meniť.

Povaha vtierania v tejto architektúre zodpovedá možnostiam, ktoré umožňuje prepisovanie odkazu v prostredí. Ako najlepšie toto prepisovanie zrealizovať je zatiaľ stále predmetom výskumu. Základný spôsob založený na priorite prístupov k odkazom je totožný s mechanizmom vtierania sa v subsumpčnej architektúre. Už sme však naznačili, že by sme vtieranie v nami navrhovanej architektúre chceli voči subsumpčnej architektúre obohatiť. Napríklad o mechanizmus zohľadňujúci časovanie agentov. Pokiaľ je časovanie jedného agenta desaťkrát rýchlejšie než iného a obaja agenti sa snažia zmeniť určitý odkaz v prostredí, hodnota navrhovaná pomalšie časovaným agentom má malú šancu byť prečítaná nejakým čitateľom tohto odkazu aj keď sú zápisy oboch agentov z hľadiska prostredia rovnocenné. Je možné taktiež uvažovať o premenlivom časovaní agentov, čím by sa systém vyhol potrebe príliš rýchleho časovania určitého agenta iba za účelom potlačenia činnosti iného agenta. Iný mechanizmus vtierania možno založiť na prídavných informáciách k odkazom – odkaz by mohol obsahovať obe konkurujúce si hodnoty a čitateľ by si z nich mohol vybrať na základe prídavných informácií.

Samotná povaha odkazov v prostrediach je tiež zatiaľ otázna. Skúšali sme použiť štruktúry typu záznam, ktorým rozumejú iba ich zapisovatelia a čitatelia, skúšali sme taktiež použiť pre tieto odkazy elementárne typy. Je však celkom možné, že ideálnym riešením sú nejaké bohatšie štruktúry s množstvom prídavných informácií.

Čo sa týka metódy tvorby, sledovať vzťahy medzi algoritmami, ktoré sú vložené v spoločenstve čisto reaktívnych agentov je takmer nad ľudské sily. Potvrdzuje sa však pôvodný kacírsky zámer, že tvorca systému to nemusí robiť: nemusí vedieť ako mu systém pracuje, stačí mu vedieť čo systém robí. Presnejšie, musí si úzkostlivo pamätať aký význam majú jednotlivé odkazy v prostrediach a ako sa tento význam upravoval. To by mu malo plne postačiť na to, aby dokázal využiť nižšie vrstvy systému a vtierať sa do ich činnosti. Táto registrácia významu odkazov by mohla byť aj podporovaná vývojovými prostriedkami. Tieto by mali taktiež automaticky robiť archív všetkých verzií, nakoľko pri testovaní sa môže ukázať, že sa systém odchýlil zo správnej cesty k produkovaniu zamýšľaného správania.

Niektoré prvky tejto architektúry boli použité v praktických systémoch a prejavili sa tam v celku pozitívne. Bol taktiež vyvinutý jeden systém výlučne postavený na tejto architektúre. Získané poznatky však zďaleka nestačia na vynesenie záveru a nestačia ani na rozhodnutie niekoľkých postulovaných otázok ohľadne jemných detailov (triggery vtieranie, povaha odkazov), žiada si to čas.

6. Záver

Autor príspevku si je plne vedomý, že poskytuje čitateľovi hypotézy, dohady, návrhy jedným slovom nič, čo by sa vyznačovalo istotou. Predkladáme tu iba niekoľko myšlienok a nevieme ako je široká oblasť, ktorej sa dotýkajú. Že pritom nejde o prázdnu množinu sa môžeme presvedčiť iba ďalším rozvíjaním týchto myšlienok. Potrebovali by sme mať viacej skúseností so systémami z oblasti praxe, ktoré by boli založené na navrhovanej architektúre a taktiež teóriu, ktorá by dokázala zachytiť a potvrdiť spomínané vlastnosti syntaktických štruktúr používaných pri implementácii. Väčšina týchto myšlienok má pôvod v oblasti novej umelej inteligencie, našim originálnym príspevkom je ich nová syntéza a ich aplikovanie do novej oblasti v ktorej raz možno zapustia pevné korene.

Literatúra

Brooks R. A.: *Achieving artificial intelligence through building robots.* A. I. Meno 899, AI Lab, MIT, Cambridge, Mass., (1986).

Brooks R. A.: *A Robust Layered Control System for a Mobile Robot*. IEEE Journal of Robotics and Automation, RA2, (1986), 14--23.

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

Brooks R. A.: *Intelligence without representation.* Artificial Intelligence 47, (1991), 135-159.

Brooks R. A.: *Building Brains for Bodies.* A. I. Memo No. 1493, AI Lab, MIT, Cambridge, Mass., (1993).

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 A., Kalaš I., Kelemen J.: *Počítač Hamlet*. Mladé letá, Bratislava, (1990).

Jones L. J., Flynn A. M.: Mobile robots: *Inspiration to implementation*. AK Peters. Ltd., Wellesley, Mass., (1993).

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

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

Kelemen J.: *Myslenie, počítač...* Sprektrum, Bratislava, (1989).

Kelemen J., Ftáčnik M., Kalaš I. Mikulecký P.: *Základy umelej inteligencie.* Alfa, Bratislava, (1992).

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

Kelemen J.: *Strojovia a agenty*, Archa, Bratislava, (1994).

Knight K.: *Are Many Reactive Agents Better than a Few Deliberative Ones ?* In: Proc IJCAI `93, Chambery, (1993), 132 – 137.

Andrej Lúčny: *Architektúra inteligentných programových systémov*, Projekt dizertačnej práce, Matematicko-fyzikálna fakulta, Univerzita Komenského v Bratislave (1997).

Andrej Lúčny: *Reaktívny model inteligentného systému*, CogSci `99, Chemicko-technologická fakulta, Slovenská technická univerzita, Bratislava (1999).

McFarland D., Bosser T.: *Intelligent Behavior in Animals and Robots.* The MIT Press, Cambridge, Mass., (1993).

Minsky M.: *The Society of Mind*. Simon & Schuster, New York, (1986).

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

Renáta Mlichová: *Niektoré experimenty so subsumpčnou architektúrou v nedeliberatívnej robotike*, diplomová práca, Matematicko-fyzikálna fakulta, Univerzita Komenského v Bratislave (1993)

Resnick M.: *Turtles, Termites and Traffic jams.* The MIT Press, Cambridge, Mass., (1994).

Singh M. P.: *Multiagent Systems.* Springer-Verlag, Berlin, (1994).

An Architecture for Intelligent Software Systems

Abstract

Relationship between modern trends of artificial intelligence and software engineering is indicated. The branches of science encounter difficult problems with practical realization of the systems for which theories expect no problems. Properties of implementation architectures are proposed as the main source of these problems. The author believes that just the nature of structures and methods currently used during phase of implementation disables us to create systems with a lot of functions or abundant (intelligent) behavior (what is recognized to be the same). It seems we are able to propose all features of such systems, however we have no such implementation architecture which would enable us to insert all these features to one unit. Adding them sequentially, the syntactic structures we use are devalued, i.e. the work required for adding the next feature is larger than for the previous one (when we express data relations explicitly using specialized algorithms) or the work for their utilization increases dramatically due to their count (when we express data relations implicitly using one general algorithm). Unfortunately the current theories cannot explain and confirm these observations, however a hint how to extend them is discussed. A new architecture potentially resistant to these phenomena is proposed. It is based on multiagent system composed of purely reactive agents which communicate only indirectly through environments, decomposition by activities and bottom-up incremental development.