Agentovo-orientované programovanie v perspektíve vývoja programovania

Andrej Lúčny

Katedra aplikovanej informatiky, Fakulta matematiky, fyziky a informatiky, Univerzita Komenského
Mlynská Dolina, 842 48 Bratislava
lucny@fmph.uniba.sk

Abstrakt

Programovacie jazyky a techniky ich použitia vieme historicky usporiadať pomocou prostriedkov, ktorými sa entity reálneho sveta prenášajú do virtuálneho prostredia počítača. Na základe tohto pohľadu rozlišujeme tradične neštruktúrované, štruktúrované a objektovo - orientované programovanie. Ukážeme, že ako logicky ďalší krok k nim možno položiť špecifický typ programovania, ktoré môžeme nazvať agentovo - orientovaným. Vyhodnotíme výhody tohto typu programovania a vhodnú aplikačnú oblasť z ktorej sú mnohé úlohy považované za tzv. úlohy umelej inteligencie.

1 Úvod

Pojem agentovo-orientované programovanie pochádza od Yoava Shohama [Shoham 1993] a z obdobia kedy si viac rôznych výskumných prúdov uvedomovalo, že pracujú s analogickým druhom modularity a pokúsili sa ju vyjadriť v abstraktnej forme. Tým vznikol obor s názvom multiagentové systémy [Wooldridge–Jennings 1995] [Wooldridge 2009]. Tento proces od začiatku sprevádzal zápas o to, ako rozsiahla bude táto abstrakcia, čo dobre dokumentuje monotematické číslo Communication of ACM z júla 1994 [Riecken 1994], v ktorom sú zastúpené aj prúdy, ktoré sa neskôr používania názvu „agent" a „multiagentový" zriekli ako napr. Marvin Minsky. Tento zápas nie je uzavretý dodnes a aj tento príspevok možno vnímať ako jeho súčasť. Na druhej strane je užitočný, lebo pri ňom dochádza k vzájomnej inšpirácii rôznych názorových prúdov. Yoav Shoham každopádne vnímal ním zavedenú predstavu o vývoji software ako súčasť sieťového programovania, v ktorom je multiagentový systém vnímaný ako špecifický druh middleware t.j. implementácie relačnej a prezentačnej vrstvy v OSI modeli [ITU-T 1993], teda riadia komunikačný dialóg a definujú formát dát pre proces marshallingu a demarshallingu. Dôraz taktiež kládol na pripisovanie „mentálnych stavov" agentom. Kým prvé zodpovedá reálnemu stavu – multiagentové systémy sú dodnes predovšetkým záležitosťou sieťového programovania – nad druhým by sme sa dnes skôr pousmiali s tým, že každá motivácia môže osožiť, pokiaľ sa dielo od tých menej vhodných neskôr očistí.

V tomto článku sa budeme snažiť pristupovať k multiagentovému systému z čisto abstraktného pohľadu. Vďaka tomuto prístupu sme v [Lucny 2004] navrhli multiagentovú architektúru Agent-Space, ktorá umožňuje vyjadriť myšlienky Brooksovej subsumpčnej architektúry [Brooks 1999] a Minského societného modelu mysle [Minsky 1986], ktoré slovenskej vedeckej komunite priblížil Jozef Kelemen [Kelemen 1994, 2003]. Základným technickým prvkom tejto architektúry je kalkul dátovej výmeny medzi agentmi realizovaný cez pomenované odkazy na „nástenke" zvanej *space*, čím tento prístup nadväzuje na podobné koncepty ako je Gelerntnerov tupple space [Gelernter 1985] a Java Space [Waldo 2001], vyvinutý v rámci technológie pre tzv. „sieť vecí" – Java Jini. Z hľadiska multiagentového prístupu tu – z istého uhľa pohľadu – teda ide o obmedzenie sa na nepriamu komunikáciu, podobne ako je tomu u multiagentového prostredia Cougaar. Tento spôsob komunikácie sa nazýva aj stigmergickým [Valckenaers 2001].

Agent-Space slúži na vývoj riadiacich systémov robotov a modelov robotov či biologických organizmov v rôznych simulátoroch [Lucny 2007, 2011]. Ako sme už uviedli, neviaže sa nevyhnutne na sieťové programovanie. Nie je teda implementovaná ako druh middleware, hoci sa dá aj týmto spôsobom ľahko rozšíriť. Jej implementácia sa opiera o multivláknové prostredie, v ktorom *space* slúži ako prostriedok dátovej výmeny medzi vláknami a k tomu potrebnej synchronizácie. (Implementácie v jazykoch Java a C++ sú dostupné na www.agentspace.org/download) Z hľadiska našej argumentácie nebude zásadný fakt, že ako príklad multiagentovej architektúry budeme mať na mysli práve Agent-Space, väčšina toho čím sa budeme zaoberať je rovnako platná pre ľubovoľnú multiagentovú platformu.

V našom príspevku sa pokúsime rozvinúť predstavu Shohama, ktorý v ním navrhnutom štýle programovania videl nový spôsob ako vyvíjať software. Preto mu aj dal špecifické pomenovanie. Vychádzajúc z filozofie, ktorá stála za zrodom objektovo-orientovaného programovania budeme argumentovať v prospech oprávnenosti tohto prístupu. Túto filozofiu, ktorá ako mnoho užitočných vecí – napríklad grafické užívateľské rozhranie – vznikla vo firme Xerox PARC pri zrode jazyka Smalltalk,

popisujeme a jej rozšírenie na rôzne spôsoby programovania podávame v kapitole 2. Výklad týchto spôsobov na konkrétnom príklade, podávame v kapitole 3. V kapitole 4 začleníme do tejto koncepcie agentovo - orientované programovanie. V závere diskutujeme o význame takéhoto pohľadu a pravdepodobnom vývoji programovania v budúcnosti, ktoré z neho vyplývajú.

2 Filozofia prenosu reálneho do virtuálneho

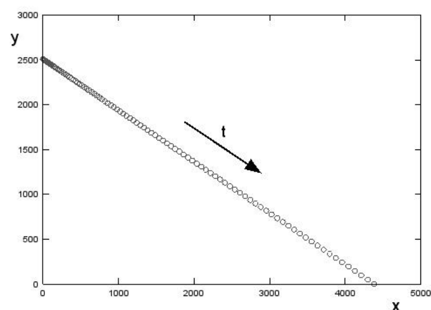
Objektovo-orientované programovanie pri svojom vzniku bolo podložené predstavou, že ide o novátorský spôsob ako v počítači reprezentovať entity reálneho sveta. Entity mali byť reprezentované ako tzv. objekty a vzťahy medzi entitami realizované posielaním správ medzi objektmi. V triedno-inštančnom modeli zodpovedá toto posielanie správ zavolaniu metódy s argumentmi, ktoré zodpovedajú obsahu správy, čo má za následok vygenerovanie návratovej hodnoty, ktorý zodpovedá odpovedi na správu. Kľúčovým pokrokom tu bolo priradenie kódu, ktorý metódu implementuje, do definície reprezentácie entity. To predtým bolo nezvyčajné, hoci sa to niekedy robievalo pomocou smerníkov na funkcie, ktoré mohli byť hodnotou niektorého z atribútov štruktúry reprezentujúcej danú entitu.

Bývali ale časy, keď nebolo v počítači vôbec možné vyjadriť, že niektoré premenné opisujú jedinú entitu a teda patria logicky k sebe. Prvé programovacie jazyky ako FORTRAN poznali len základné typy a neumožňovali ich kompozíciu do typov zložených. Bolo úlohou programátora strážiť, že určitá skupina premenných patrí k sebe, lebo popisujú rovnakú entitu. Veľmi problematické bolo taktiež tieto entity namnožiť. Najobľúbenejší spôsob bol reprezentovať ich opisné premenné ako polia a použiť index do týchto polí – tzv. deskriptor – na určenie, s ktorou z entít daného druhu sa práve bavíme. Išlo tu teda o spôsob reprezentácie, ktorý by sme mohli – vzhľadom na to, čo prišlo po ňom – nazvať neštruktúrovaným.

S jazykmi ALGOL a COBOL prišla éra tzv. štruktúrovaného programovania. Okrem zloženého príkazu, ktorý vystriedal používanie skokov a dal tomuto štýlu programovania meno, priniesol zložený typ – tzv. štruktúru (struct, record) – pomocou ktorej bolo možné spojiť všetky premenné opisujúce určitú entitu do jedného celku. Prístup k týmto premenným bol zabezpečený smerníkom na danú štruktúru, čím sa vyriešila aj správa viacerých inštancií štruktúr rovnakého typu. Štruktúrou reprezentovaná entita už teda tvorila celok nielen v mysli vývojára, ale aj v ním napísanom programe. Bola to však entita pasívna, čokoľvek sa na nej malo zmeniť, muselo sa udiať externe. Nebol jej formálne priradený žiaden kód a takéto priradenie existovalo opäť len v mysli vývojára. Jedinou výnimkou bola už vyššie zmienená možnosť priradiť do premennej smerník na funkciu určitého prototypu, čím vzniká tzv. dynamická väzba umožňujúca dokonca niečo ako dedenie.

Ďalší pokrok prináša objektovo-orientované programovanie, ktoré v podobe objektov obohacuje štruktúru o kód. Tento kód dokáže vykonať potrebnú manipuláciu atribútov objektu, kedykoľvek je zvonku vyvolaná.

Na rôzne etapy vývoja programovania sa teda môžeme dívať ako na spôsoby prenosu entít reálneho sveta do virtuálneho sveta počítača. Neštruktúrované programovanie realizuje tento prenos pomocou premenných, ktoré formálne nesúvisia. Štruktúrované programovanie robí to isté pomocou štruktúr, ktoré v počítači tieto premenné formálne spoja. Objektovo-orientované programovanie k nim vie pomocou objektov formálne pripojiť aj kód. Postupne sa teda formálna reprezentácia reálneho vo virtuálnom stáva bohatšou a schopnejšou zachytiť viac aspektov reálnej entity a formálne ich vyjadriť. Ako by mal logicky tento proces pokračovať ďalej? Aký ďalší aspekt reálnej entity je na rade, aby bol formálne vyjadrený a obohatil jej virtuálnu reprezentáciu?



Obr. 1.: Príklad deja simulovaného v počítači

3 Motivačný príklad

Hľadať odpoveď na túto otázku budeme na konkrétnom príklade. Zoberme si nejakú vhodnú entitu reálneho sveta, napríklad guličku. Ignorujme jej polomer a hmotnosť a zamerajme sa polohu a rýchlosť. Ako prostredie, v ktorom sa nachádza, budeme pre jednoduchosť uvažovať len naklonenú rovinu, ktorú popisuje uhol φ (napr. 30°), pričom poloha bude vyjadrená len dvomi súradnicami x a y (na počiatku 0 a 2500 cm). Ako dej, ktorý sa bude v počítači s virtuálnym náprotivkom guličky odohrávať, uvažujeme jej kotúľanie sa po danej naklonenej rovine, pričom čas považujeme za diskrétny s krokom 100ms. Celá simulácia bude trvať 10 sekúnd, čiže 100 krokov (obr. 1). V kóde abstrahujeme od potreby guličku nejako zobrazovať, či vypisovať parametre pre toto zobrazenie, stačí nám, že sa v pamäti počítača tieto parametre správne v čase menia.

Takže ako by také kotúľanie vyzeralo pri neštruktúrovanom programovaní? Každému parametru bude zodpovedať jedna premenná, označme ich typicky poloha – *x, y*, rýchlosť – *v* a gravitačné zrýchlenie *g*. Čas označíme identifikátorom *t* a jeho zmenu *dt*. Súčin *g.dt* označíme ako *gdt* (kód 1).

```
int main() {
  float fi = 0.52;
  float x = 0, y = 2500;
  float v = 0;
  float gdt = 10.0*100/1000;
  for (int t=0; t<10000; t+=100) {
    v += gdt;
    x += v*cos(fi);
    y -= v*sin(fi);
    usleep(100000);
  }
  return 0;
}
```

Kód 1: Príklad kódu podľa neštruktúrovaného programovania

```
#define MAXN 10
float x[MAXN];
float y[MAXN];
float v[MAXN];
int dscr = 0;

void set (int index) {
  dscr = index;
}

int init (float x0, float y0) {
  x[dscr] = x0;
  y[dscr] = y0;
}

int roll (float gdt, float fi) {
  v[dscr] += gdt;
  x[dscr] += v[dscr]*cos(fi);
  y[dscr] -= v[dscr]*sin(fi);
}

int main() {
  float fi = 0.52;
  float gdt = 10.0*100/1000;
  set(0); init(0,2500);
  set(1); init(0,2500);
  for (int t=0; t<10000; t+=100) {
    set(0); roll(gdt,fi);
    set(1); roll(gdt,fi);
    usleep(100000);
  }
  return 0;
}
```

Kód 2: Riešenie viacerých entít pomocou deskriptorov

Vidíme, že príslušný program je jednoduchý. Čo v ňom však nevidíme je virtuálna gulička. Podstatne zložitejšie by bolo kotúľať viac guličiek (kód 2), čo nás prinúti vytiahnuť z hlavného programu časti kódu do funkcií.

Ale ani v tomto riešení nevidíme formálnu reprezentáciu guličky. To nám umožní až riešenie v štýle štruktúrovaného programovania (kód 3).

```
typedef struct {
  float x;
  float y;
  float v;
} BALL;

void init (BALL *b, float x0, float y0) {
  b->x = x0;
  b->y = y0;
  b->v = 0;
}

void roll (BALL *b, float fi, float gdt) {
  b->v += gdt;
  b->x += b->v*cos(fi);
  b->y -= b->v*sin(fi);
}

int main() {
  float fi = 0.52;
  float gdt = 10.0*100/1000;
  BALL *b = (BALL *) malloc(sizeof(BALL));
  init(b,0,2500);
  for (int t=0; t<10000; t+=100) {
    roll(b,fi,gdt);
    usleep(100000);
  }
  return 0;
}
```

Kód. 3: Príklad kódu podľa štruktúrovaného programovania

Spravovať viac guličiek je pri tomto riešení ľahké, stačí alokovať ďalší pamäťový priestor a smerník naň používať na prístup k nemu. Pretože tu už máme formálne guličku v podobe zloženého typu. Táto gulička so sebou nič nevie urobiť, ale už je tam. Ďalším krokom, ako ju obohatiť o kód, t.j. ako formálne povedať, že funkcia na jej kotúľanie k nej patrí, je použitie smerníkov na funkciu (kód 4).

```
typedef struct ball {
  float x;
  float y;
  float v;
  void (*init) (struct ball *, float, float);
  void (*roll) (struct ball *, float, float);
} BALL;

void ball_init (BALL *b, float x0, float y0) {
  b->x = x0;
  b->y = y0;
  b->v = 0;
}
void ball_roll (BALL *b, float fi, float gdt) {
  b->v += gdt;
  b->x += b->v*cos(fi);
  b->y -= b->v*sin(fi);
}
```

```
int main() {
  float fi = 0.52;
  float gdt = 10.0*100/1000;
  int t;
  BALL *b = (BALL *) malloc(sizeof(BALL));
  b->init = ball_init;
  b->roll = ball_roll;
  b->init(b,0,2500);
  for (t=0; t<10000; t+=100) {
    b->roll(b,fi,gdt);
    usleep(100000);
  }
  return 0;
}
```

Kód. 4: Použitie smerníkov na funkciu

V skutku takto podobne vyzerá kód v jazyku C skompilovaný z objektovo-orientovaného jazyka C++. Nakoniec môžeme tieto konštrukty začleniť do programovacieho jazyka a dostaneme riešenie v štýle objektovo-orientovaného programovania (kód 5).

```
class Ball {
  private:
    float x;
    float y;
    float v;
  public:
    Ball(float x0, float y0);
    ~Ball();
    void roll (float gdt, float fi);
};

Ball::Ball (float x0, float y0) {
  x = x0;
  y = y0;
  v = 0;
}
Ball::~Ball () {};

void Ball::roll (float fi, float gdt) {
  v += gdt;
  x += v*cos(fi);
  y -= v*sin(fi);
}

int main() {
  float fi = 0.52;
  float gdt = 10.0*100/1000;
  Ball *b = new Ball(0,2500);
  for (int t=0; t<10000; t+=100) {
    b->roll(fi,gdt);
    usleep(100000);
  }
  return 0;
}
```

Kód. 5: Príklad kódu podľa objektovo-orientovaného programovania

Ukázali sme si na konkrétnom príklade vývoj programovania v troch fázach: neštruktúrované, štruktúrované a objektovo-orientované. Vidíme, že každá fáza zavádza do jazyka nejaký zásadný konštrukt: štruktúrované programovanie zložený typ (štruktúru) a objektovo orientované triedy s atribútmi aj metódami. Vidíme i to, že tomuto konštruktu predchádza určitá praktika používania prostriedkov bez neho, ktorá ho predznamenáva.

Ako to teraz pôjde ďalej? Na to porovnávajme hlavné programy jednotlivých riešení. Vidíme, že kým v prvej fáze sa dal celý program napísať ako hlavný, postupne z neho boli kusy kódy vytiahnuté mimo neho. Najprv šla preč definícia parametrov modelovanej entity, potom detaily jej pohybu a inicializácia jej reprezentácie. Čo je ďalšie na rade, čo by mohlo z hlavného programu zmiznúť?

4 Agentovo-orientované programovanie

Pozorný čitateľ si v tejto chvíli pravdepodobne vie aj sám na otázku položenú na konci prechádzajúcej kapitoly odpovedať: z hlavného programu zmizne cyklus! Proste spravíme guličku a necháme ju nech sa sama kotúľa (kód 6).

Na prvý pohľad nemusí byť zrejmé v čom nám to prináša výhodu. A to preto, lebo zatiaľ sme kotúľali len jednu guličku, prípadne dve guličky ale iba synchrónne. Pokiaľ by sme však kotúľali každú guličku inokedy, cyklus v hlavnom programe by nebol taký jednoduchý. Zatiaľ čo keď ho eliminujeme, tento problém odpadne.

```
class Ball : public Agent {
  private:
    float x;
    float y;
    float v;

  protected:
    void init (string args) {
      v = 0;
      timer_attach(100,100);
    }
    void sense_select_act (int pid) {
      float dflt = 0.0;
      float gdt = (float) space_read("gdt",dflt);
      float fi = (float) space_read("fi",dflt);
      v += gdt;
      x += v*cos(fi);
      y -= v*sin(fi);
    }

  public:
    Ball (float x0, float y0) : Agent("") {
      x = x0;
      y = y0;
    };
};
int main () {
  float dgdt = 10.0*100/1000;
  float dfi = 0.52;
  Space_write("gdt",dgdt);
  Space_write("fi",dfi);
  Ball b(0,2500);
```

```
  delay(10000);
}
```

Kód. 6: Príklad kódu podľa agentovo-orientovaného programovania

Iný problém sa však objaví: nemáme ako do volania roll() odovzdať parametre naklonenej roviny (predpokladajme, že by sa teoreticky mohli meniť počas simulácie, takže sa to nedá vybaviť pri inicializácii). Multiagentový systém to umožňuje riešiť komunikáciou medzi agentami – a to priamou, t.j. poslaním správy (napr. zavolaním metódy), alebo nepriamou, t.j. vložením hodnôt týchto parametrov na nejakú „nástenku“, onen vyššie zmienený *space*, ktorá je v rámci celého programu tzv. singletonom, takže každý má k nej prístup. V riešení, ktoré prezentuje kód 6 je použitý druhý zo zmieňovaných prístupov.

```
class Ball {

  private:
    float x;
    float y;
    float v;

  protected:
    .Ball(int pid) {
      float gdt = .gdt(0.0);
      float fi = .fi(0.0);
      v += gdt;
      x += v*cos(fi();
      y -= v*sin(fi);
    }

  public:
    Ball (float x0, float y0) {
      x = x0;
      y = y0;
      v = 0;
      timer 100,100;  // pid = trigger .gdt;
    };
};

int main () {
  float .dgdt = 10.0*100/1000;
  float .dfi = 0.52;
  Ball b(0,2500);
  delay(10000);
}
```

Kód. 7: Fikcia kódu podľa agentovo-orientovaného programovania

V tomto riešení vidíme aj to, že z hľadiska pokroku technológie ide o určitú medzifázu. Podobne ako boli prvé štruktúry naprogramované pomocou deskriptorov a prvé objekty pomocou smerníkov na funkcie, tu je agent naprogramovaný ako objekt s vlastným vláknom (čo je ukryté v implementácii triedy Agent od ktorej je náš konkrétny agent `Ball` odvodený). Je preto možné očakávať nejaké zakotvenie tohto mechanizmu v jazyku. Našu fikciu možno vidieť na kód 7. Okrem špeciálnej metódy, ktorá obsluhuje jeden prechod základným cyklom agenta a špeciálnej notácie pre operácie čítania a zápisu v *space*, tam vidíme kľúčové slová *timer* a (nepoužitý ale teoreticky potrebný) *trigger*, ktoré definujú podmienky budenia tohto cyklu. Pre zložitejšie úlohy by sme potrebovali toho ešte viac, najmä syntaktické vyjadrenie hromadných operácii nad dátami v *space*.

Hoci existuje viacero programovacích prostredí, ktoré si dávajú do názvu „agent programming language“ ako napr. SARL [Rodriguez 2014], JADEX či dokonca JADE, nielen že nezapadajú do našej úvahy, ale neprinášajú žiadne nové jazykové konštrukty, ktoré by sa museli riešiť kompiláciou podobne ako sa kód C++ kompiluje do C. Taký programovací jazyk nám nie je známy.

5 Záver

V príspevku sme sa pokúšali o zaradenie techniky programovania, ktorá sa v rôznych podobách objavuje posledných dvadsať rokov a je známa pod názvom agentovo-orientované programovanie. Snažili sme mu vykresať takú podobu, v ktorej by sme ho mohli položiť do hlavnej vývojovej línie programovacích jazykov. To si žiadalo značné očistenie konceptu, čo na jednej strane možno hodnoť ako našu opovážlivosť, na druhej ako náš netriviálny vklad k danej problematike. Ostáva zodpovedať otázku, čo sme tým sledovali.

Kým štruktúrované programovanie prinieslo možnosť implementovať systémy s komplikovanými dátami a objektovo - orientované s komplikovaným kódom, prínosom agentovo - orientovaného programovania by mali byť systémy skomplikovaným riadením. Ide o komplexné systémy, v ktorých prebieha množstvo procesov, ktoré sa vzájomne ovplyvňujú a podporujú. Jednou vetou systémy, ktoré sa viac podobajú živým organizmom či našej mysli.

Zmyslom použitia tohto nástroja je teda práve vývoj systémov s umelou inteligenciou.

Poďakovanie

Tento príspevok vznikol za podpory grantovej agentúry ASFEU v rámci grantovej úlohy KC-INTELINSYS, ITMS 26240220072.

Literatúra

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

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

ITU-T: *Architecture Framework for the development of signalling and OA&M protocols using OSI concepts.* Telecommunication standardization sector. Recommendation Q.1400, 03/1993

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

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

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

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

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

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

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

Riecken, D. *Intelligent Agents.* Communication of ACM 37 (7).

Rodriguez, S. - Gaud, N. - Galland, S.: *SARL: a general-purpose agent-oriented programming language.* In The 2014 IEEE/WIC/ACM International Conference on Intelligent Agent Technology. Warsaw, Poland: IEEE Computer Society Press, 2014

Shoham, Y. *Agent-Oriented Programming*. Artificial Intelligence 60 (1993), pp. 51–92

Šešera, L. - Mičovský, A.: *Objektovo-orientovaná tvorba systémov (analýza, návrh, implementácia) a jazyk C++*. Bratislava: Alfa, 1993

Valckenaers, P. - Van Brussel, H. - Kollingbaum, M. - Bochmann O.: *Multi-agent Coordination and Control Using Stigmery Applied*. In: Multi-Agent Systems and Applications (Luck, M. - Maøík, V. - Štepánková, O. - Trappl, R., eds.), EASSS, Prague, 2001, pp. 317-334.

Waldo J: *Mobile Code, Coordination and Changing Networks.* CONCOORD, Lipari, 2001

Wooldridge, M.: *An Introduction to MultiAgent Systems - Second Edition*, John Wiley & Sons, 2009

Wooldridge, M. – Jennings, N. R.: *Agent Theories, Architectures, and Languages: a Survey. Intelligent Agents.* Ed.: Wooldridge, M., Jennings, N. R. Berlin: Springer-Verlag, 1995.