Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Bc. Andrej Bosík

Komunikace v Multi-Agentních systémech

Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: prof. RNDr. Petr Štěpánek, DrSc.
Studijní program: Informatika, Teoretická informatika

Poďakovanie: Moje poďakovanie patrí môjmu vedúcemu diplomovej práce, pánovi prof. RNDr. Petrovi Štěpánkovi, DrSc., za to, že ma vždy vedel dobre usmerniť, dať mi užitočnú radu a nestrácal so mnou trpezlivosť. Takže Ďakujem.

Prehlasujem, že som svoju diplomovú prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce.

V Prahe dňa : 8. 12. 2010 Andrej Bosík

Název práce: Komunikace v Multi-Agentních systémech
Autor: Bc. Andrej Bosík
Katedra: Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: prof. RNDr. Petr Štěpánek, DrSc.
e-mail vedoucího: estepanek@ktiml.mff.cuni.cz
Abstrakt: Táto práca rozoberá problematiku komunikácie medzi agentami v MAS. Najskôr som sa pokúsil priblížiť logických agentov, špeciálne racionálnych logických agentov. Na popis týchto agentov som použil logiku LORA. Ukázal som syntax a sémantiku LORA-y. Následne som sa zaoberal architekúrov MAS podľa FIPA špecifikácii. Ako implementáciu tejto architektúry som použil prostredie JADE, ktoré je plne implementované v JAVA jazyku. Skúsil som popísať ako vyzerá komunikácia medzi dvoma JADE agentami, aký tvar a parametre majú správy podľa ACL jazyka, čo je jazyk vyvinutý FIPA-ou pre komunikáciu medzi agentami. Popísal som ako agenti používajú protokoly. JADE neobsahovalo žiadny protokol pre vyjednávanie. Preto som doplnil JADE o nový VETO protokol, ktorý môžu využívať všetci JADE agenti. Aby som to mohol urobiť, doplnil som JADE o nové parametre správy, ktoré rozšírujú parametre správ určených ACL jazykom. Na ukážku použitia VETO protokolu som naprogramoval v JADE dve triedy agentov (pre iniciátora a aj pre respondéra), ktoré medzi sebou komunikujú pomocou tohto protokolu. Následne som ukázal, ako sa do JADE dajú pridávať ďalšie nové vyjednávacie protokoly. Ako príklad som pridal do JADE vyjednávací NEGO protokol. Nakoniec som použil moju novo implementovanú funkcionalitu vyjednávania na zložitejšom MAS.
Klíčová slova: Komunikácia, Multi-agentné systémy, LORA, protokol, FIPA, JADE

Title: Communication in Multi-Agent Systems
Author: Bc. Andrej Bosík
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: RNDr. Petr Štěpánek, DrSc.
Supervisor's e-mail address: estepanek@ktiml.mff.cuni.cz
Abstract: This thesis analyses the questions of communication between agents in MAS. Firstly, I tried to introduce a logic agents, particularly logic rational agents. For rational logic agents characterization I used LORA logic. I illustrated the syntax and the semantic of LORA. Then I concentrated on architecture MAS according to a FIPA specification. I used the JADE environment as implementation of this architecture. JADE is fully implemented in the JAVA language. I described how the communication between two JADE agents works and what form and parameters the messages according to the ACL language have. FIPA developed the ACL language for communication between agents. I described how agents use the protocols. JADE didn´t contain any protocol for negotiation. That is why I integrated a new VETO protocol into JADE. This protocol can use all the JADE agents. In order to do it, I integrated new message parameters into JADE. This message parameters extend the ACL message parameters. For demonstration I implemented two classes in the JADE (one for the initiator and one for the responder) using a Veto protocol. Further, I described how new negotiation protocols could be add to JADE. For example I added to JADE another negotiation protocol, called NEGO. At last I used my new implemented ability to negotiation on more complicated MAS.
Keywords: Communication, Multi-agent systems, LORA, protocol, FIPA, JADE

Obsah :

1. Kapitola : Úvod

Základné znalosti a pojmy o agentoch som získal a prijal z [1]. V tejto knihe sú veľmi dobre vysvetlené základné pojmy z teórie multi-agentných systémov.

Jedným zo základných problémov výpočtovej logiky je popísanie problému. V MAS sa často používa logický popis agentov. V mojej práci predstavím logiku LORA [14], ktorá je veľmi vhodná na popis racionálnych agentov. Ukážem syntax aj sémantiku tejto logiky. Základné znalosti a pojmy o racionálnych agentoch, ako aj o ich logickom popise som získal a prijal z [8].

Multi-agentné systémy sú systémy zložené z viacerých vzájomne reagujúcich výpočtových prvkov - agentov. Agenti sú výpočtové systémy s troma základnými schopnosťami :

- Sú schopné autonómnych akcií
- Sú schopné vnímať a reagovať na určité vonkajšie vnemy
- Sú schopné vzájomnej reakcie medzi sebou

Touto vzájomnou reakciou sú myslené spoločenské aktivity, ktoré zažíva aj človek v každodennom živote ako: spolupráca, koordinácia, vyjednávanie ...
Aby vôbec bola takáto schopnosť reakcie uskutočniteľná, agenti musia vedieť medzi sebou komunikovať. Komunikácia medzi agentami je veľmi dôležitá súčasť vývoja multi-agentných systémov a v tejto práci jej budeme venovať pozornosť.

V mojom chápaní komunikácia prebieha podľa tzv. “Speech Act“ teórie. Táto teória berie komunikáciu ako akciu. Je založená na predpoklade, že komunikovanie je vykonávané agentami k presadzovaniu svojich cieľov, tak ako ostatné akcie. Táto teória ovplyvnila mnohé prístupy a jazyky pre komunikáciu medzi agentami.

V tejto práci naviažem na prácu FIPA (The Foundation for Intelligent Physical Agents) [2], ktorá vyvíja štandardy pre multi-agentné systémy. Základná architektúra FIPA je zhrnutá v [3].

Agenti medzi sebou komunikujú tak, že si navzájom posielajú správy. Na “vonkajší“ popis správy FIPA vyvinula ACL (Agent communication language), ktorý definuje “obal“ správy. To znamená, že definuje napríklad typ správy, odosielateľa, príjemcu, protokol ... Bližší popis štruktúry správy je v [4]. ACL sa nezaoberá

obsahom správy. Pre tento účel FIPA vyvinula SL Content Language, ktorý presne určuje, ako má vyzerať obsah spávy. Lepšie zoznámenie s týmto jazykom nájdete v [5].

Zásadnú úlohu v komunikácii medzi agentami hrajú protokoly. Protokoly určujú pravidlá medzi dvoma, alebo viacerými komunikujúcimi agentami. Obmedzujú rozsah povolených prejavov pre každého agenta v ľubovoľnom stave počas komunikácie. V mojej práci som sa zameral na protokoly a ich používanie.

FIPA sa zaoberá len teoretickými štandardmi, ktoré ešte treba implementovať v praxi. Na implementáciu som použil prostredie JADE. Bližší popis JADE nájdete v [6]. Čo mi v JADE chýbalo, bola funkcionalita vyjednávania. Vyjednávanie je veľmi dôležitá súčasť komunikácie medzi agentami. Preto som do JADE pridal protokol určený na dvojfázové vyjednávanie medzi agentami (nazval som tento protokol VETO) a taktiež som doplnil do FIPA štandardov ACL, ktoré používa JADE, nové parametre správ. Taktiež v tejto práci píšem, ako vytvárať nové JADE protokoly, a ukazujem ich používanie. Ako príklad pridávania nových vyjednávacích protokolov som implementoval do JADE ďalší protokol, ktorý som nazval NEGO. Taktiež som ukázal funkcionalitu vyjednávania na zložitejšom MAS.

2. Kapitola : Agenti

Svet okolo nás je plný agentov. Myslím tým takých agentov, ktorí sú aktívni, vykonávajú akcie, ktoré si sami zvolia. Tieto akcie sú vykonávané, aby menili a tvarovali prostredie, ktoré agenti obývajú. V reálnom svete je najčastejší príklad agenta človek. Agenti sú samozrejme aj iné entity ako napríklad vláda, firma...

Základné vlastnosti, ktoré požadujeme od agentov, sú :

AUTONÓMIA – schopnosť fungovať nezávisle. To znamená minimálne, že agent robí nezávislé rozhodnutia (tj. jeho rozhodovací proces je pod jeho kontrolou a nie riadený inými agentami). Avšak môžeme predpokladať, že autonómny agent má vlastnú vieru, priania a zámery, ktoré nie sú podriadené iným agentom.

PROAKTÍVNOSŤ – agent má určitý cieľ, alebo zámer a očakávame, že sa ho agent pokúsi dosiahnúť. Táto vlastnosť nám vylučuje pasívnych agentov, tj. agentov, ktorí sa nikdy o nič nepokúsia.

REAKTÍVNOSŤ – agent vníma a reaguje na zmeny prostredia.

SOCIÁLNA SCHOPNOSŤ – Schopnosť vzájomnej reakcie medzi sebou. Ale nemyslím tým len výmenu bitov (ako napr. milióny počítačov, ktoré si medzi sebou vymieňajú data), ale aj schopnosť vyjednávať a spolupracovať. Bez týchto schopností by mnohé ciele agentov boli nesplniteľné.

Všimnime si, že zostrojenie čisto reaktívneho systému, ktorý jednoducho odpovedá na podnety prostredia, je pomerne jednoduché. Môžeme implementovať tabuľku, v ktorej budú stavy prostredia a k nim priamo priradené akcie. Podobne nie je veľmi ťažké zostrojiť systém, ktorý sa snaží dosiahnuť svoj cieľ, bez ohľadu na to, ako sa mení prostredie, v ktorom pracuje. To je napríklad bežný počítačový program naprogramovaný v Pascale. Avšak zostrojenie systému, ktorý efektívne kombinuje vyššie spomenuté prístupy, je veľmi náročný problém.

Jedno zo základných delení agentov je na racionálnych agentov a "len" agentov. Agent je racionálny, ak si vyberá na vykonávanie akcie, ktoré sú v jeho najlepšom záujme, vzhľadom k jeho viere o svete (tj. prostredie, ktoré obýva). Napríklad: Ak je môj cieľ ostať suchý a verím, že vonku prší, potom je pre mňa v najlepšom záujme si zobrať dáždnik, keď odchádzam z domu. Avšak môže sa stať, že je vonku pekne slnečno a v tom prípade nie je rozumné brať si dáždnik so sebou. Moje správanie je však racionálne v každom počasí, ak verím, že vonku prší. Na tomto príklade som chcel ukázať, že rozhodujem o tom, akú akciu mám vykonať za predpokladu: Ak je moja viera o svete správna, vykonanie určitej akcie dosiahne určitý cieľ.

2.1 BDI model:

BDI model (belief-desire-intention model) dostal svoje meno na základe faktu, že považuje viery, priania a zámery za prvoradé v racionálnych akciách. Intuitívne, agentové viery sú informácie, ktoré má agent o svete. Viery môžu byť neuplné, ale aj nesprávne. Agentové prianie je situácia (stav), ktorú si agent praje, aby nastala (v ideálnom svete).
Väčšinou požadujeme, aby agentové priania boli konzistentné. A agentov zámer je jeho prianie, ktoré sa snaží dosiahnuť. Agent vo všeobecnosti nie je schopný dosiahnuť všetky jeho priania, a to ani vtedy, ak sú jeho priania konzistentné. Agent musí preto určiť nejakú podmnožinu jeho prianí a snažiť sa ju dosiahnuť. Táto podmnožina sú jeho zámery. Niekedy určiť zámery agenta býva veľmi obtiažne. Vyžaduje si to veľa času ako aj zdrojov. Proces určovania zámerov agenta musí byť efektívny a hlavne konečný. Ďalej by malo platiť, že ak príjmem nejaký zámer, tak ho nezruším bez toho, aby som vynaložil aspoň nejaké úsilie ho dosiahnuť. A zasa, keď príjmem nejaký zámer, neznamená to, že ho nemôžem zrušiť skôr, ako ho dosiahnem. Keď príjmem nejaký zámer, tak tento zámer by mal obmedzovať moje rozmýšľanie v budúcnosti (pretože zámery sú konzistentné). Všimnime si, že zámery sú úzko späté s mojou vierou o budúcnosti. Nie je racionálne mať zámer X, ak verím, že X v budúcnosti nenastane. Ale ako racionálne považujeme mať zámer X a nemám žiadnu vieru o tom či X v budúcnosti nastane, alebo nenastane.

Čo robí BDI model určitým spôsobom zaujímavý je, že kombinuje 3 dôležité elementy :

1. Filozofický element : Je založený na dobre známej a rešpektovanej teórii racionálnych akcií u ľudí
2. Software-ová architektúra : Bol implementovaný a úspešne používaný v mnohých zložitých aplikáciach
3. Logický element : Táto teória bola dôsledne formalizovaná pomocou logiky

Teóriu racionálnych akcií u ľudí vyvinul Micheal Bratman. Je to teória praktického rozmýšlania. Aspoň stručne spomeniem, že Bratmanova teória sa obzvlášť

venuje roli zámerov v praktickom rozmýšľaní. Bratman tvrdí, že zámery sú dôležité, pretože obmedzujú rozmýšľanie agenta o výbere ďalšej akcie, ktorá sa má vykonať. Pre bližšie informácie odporúčam knihu [11].

BDI model bol implementovaný už mnohokrát. Pôvodne bol realizovaný v IRMA (Intelligent Resource-bounded Machine Architecture [12]), čo bola viacmenej priama realizácia Bratmanovej teórie. Jedna z najznámejších implementácií je PRS (Procedural reasoning system [13]). Existuje veľa následníkov tejto implementácie.

Uvažujme o BDI modele ako o logickej komponente. Táto logická komponenta poskytuje nástroje, ktoré nám umožňujú uvažovať o BDI agentoch. Použitie logiky na popis BDI modelu má veľa výhod. To, že sme si zvolili dobre definovaný, štruktúrovaný jazyk (narozdiel od nedobre definovaného, neštruktúrovaného prirodzeného jazyka), nám umožňuje skúmať otázku, čo môžeme vyjadriť presne, matematickým spôsobom. Ďalej sa vyhneme dvojzmyslom a logika je pomerne prehľadný jazyk. Tým, že príjmeme prístup založený na logike, môžeme používať výsledky a techniky jednej z najpodstatnejších, najstarších a najtradičnejších vetiev matematiky.

Keď chceme používať logiku, tak najskôr treba určiť, akú logiku použiť. Prvé rozhodnutie je, či používať logiku 1. rádu, alebo použiť nejakú obohatenú logiku. V logike 1. rádu sa dá dostatočne vyjadriť takmer každá forma poznania, ktorú potrebujeme. Ja budem ale skôr rozoberať obohatenú logiku LORA (Logic Of Rational Agents). Osobne si myslím, že formuly v logike 1. rádu sú horšie čitateľné a pochopiteľné pre človeka ako formuly napísané pomocou LORA. Napríklad: výrok : "Nie sme manželia, pokiaľ sa nezosobášime." Odpovedá formule logiky 1. rádu :

$(\exists t)\ (Cas\ t)\ \&\ (t > teraz)\ \&\ Sobas(ja, ty, t)\ \&\ ((\forall t1)\ Cas(t1)\ \&\ (teraz <= t1 < t) \Rightarrow not(Manzelia(ja,ty,t1)))$,

A formula LORA vyzerá : $not(Manzelia(ja,ty))\ U\ Sobas\ (ja,ty)$. Kde U je modálna spojka, ktorá sa číta ako pokiaľ.

Modálnu a časovú logiku môžeme chápať ako špeciálny jazyk pre reprezentovanie vlastností relačných štruktúr. V prípade časových vlastností, relačná štruktúra je časová štruktúra sveta (uzly štruktúry sú časové body a hrany sú priechod času). Ak nebudeme používať modálne spojky na vyjadrenie vlastností relačnej

štruktúry, potom musíme relačnú štruktúru reprezentovať tak ako vo vyšie uvedenom príklade, pomocou časových bodov.

3. Kapitola : LORA

LORA (Logic Of Rational Agents) bola vyvinutá duom Anand Rao a Michael Georgeff, uverejnená v [14]. Je to logika, ktorá reprezentuje vlastnosti racionálnych agentov a dovoľuje nám uvažovať o nich jednoznačným a dobre definovaným spôsobom. Ako každá logika, má aj LORA svoju syntax a sémantiku. Syntax definuje množinu prípustných konštrukcií (formulí). Sémantika popisuje presný význam každej formuly.

Jazyk LORA-y kombinuje 4 odlišné komponenty :

1. Komponenta logiky 1. rádu.
2. Belief – desire – intention komponenta
3. Časová komponenta
4. Akčná komponenta

Tieto komponenty skúsim popísať.

3.1 Komponenta logiky 1. rádu :

LORA je rozšírením logiky 1. rádu, takže obsahuje všetky spojky z tejto logiky ako : negáciu "not", konjukciu "$\wedge$", disjunkciu "$\vee$", implikáciu "$\Rightarrow$", ekvivalenciu "$\Leftrightarrow$", ako aj existenčný "$\exists$" a univerzálny "$\forall$" kavantifikátor. Vo zvyšku tejto kapitoly budeme uvažovať negáciu a disjunkciu ako základné spojky a ostatné spojky budeme považovať za odvodené. To znamená, že ich budeme písať ako skrátenie zápisu. Teda :

- $\varphi \wedge \psi$ je skratka za : $not(not(\varphi) \vee not(\psi))$
- $\varphi \Rightarrow \psi$ je skratka za : $(not(\varphi) \vee \psi)$
- $\varphi \Leftrightarrow \psi$ je skratka za : $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

Podobne definujeme aj existenčný kvantifikátor pomocou univerzálneho :

- $\exists x.\varphi$ je skratka za $not(\forall x)\ not(\varphi)$

3.2 Belief – desire – intention komponenta

Jedným z rozšírení logiky 1. rádu je pridanie modálnych spojok pre reprezentáciu vier, prianí a zámerov agenta. Prehľad týchto spojok je :

- Formula (Bel i Ψ) znamená, že agent i verí Ψ.
- Formula (Des i Ψ) znamená, že agent i si praje Ψ.
- Formula (Int i Ψ) znamená, že agent i zamýšla Ψ.

Vo formule (Bel i Ψ) nie je Bel predikát z logiky 1. rádu s dvoma argumentmi (aj keď tak vyzerá). Argumenty predikátov z logiky 1. rádu musia byť termy logiky 1. rádu. Avšak modálna spojka Bel má ako prvý argument špeciálny druh termu, ktorý musí predstavovať agenta. Ako druhý argument má Bel LORA formulu.To znamená aj fakt, že nemôžeme kvantifikovať cez druhý argument. Modálna spojka Bel môže byť vložená, to znamená, že napríklad následujúca formula je legálna LORA formula : (Bel andrej (Bel michal Prezident(Klaus))) a znamená, že agent andrej verí, že agent michal verí, že prezidentom je Klaus.

To isté platí aj pre modálne spojky Des a Int. Navyše platí, že Bel, Des a Int môžu byť vložené jedna druhej. Napríklad : (Des andrej (Bel michal Prezident(Klaus))) je legálna LORA formula a znamená, že agent andrej si praje, aby agent michal veril, že prezidentom je Klaus.

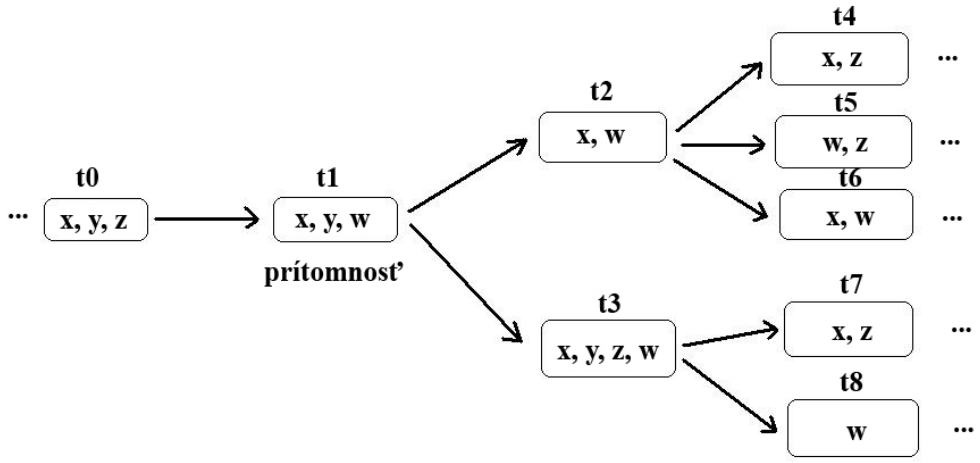
3.3 Časová komponenta

Formuly logiky 1. rádu a modálne spojky (Bel ,Des, Int) nám dovolujú vyjadriť vlastnosti okamžitých stavov. Je potreba ešte zaviesť spojky, ktoré nám pomôžu vyjadriť dynamiku agentov a ich prostredia. Skúsme priblížiť náš časový model.

Časový model :

Majme predpoklad, že prostredie môže byť v ktoromkoľvek stave z množiny prípustných stavov. Jeden z týchto stavov je prítomnosť (teraz). Minulosť je len jedna postupnosť stavov. Budúcnosť nie je určená (vetví sa). Od daného časového bodu (stavu) sa vetví viacero postupností časových bodov (každý časový bod je postupnosť stavov), ktoré reprezentujú možnú budúcnosť. Toto vetvenie záleží na agentoch, aké akcie vykonávajú (ako zmenia prostredie), ako aj na nedeterministických zmenách

prostredia. Používame teda časový model, ktorý je diskrétny, ohraničený a lineárny do minulosti (má začiatok a je len jedna minulosť), nie je ohraničený do budúcnosti (nekonečný čas) a do budúcnosti sa vetví (nie je určené, ktorý stav nastane). Jedna vetva z tohto vetvenia je jedna možná budúcnosť a budem ju volať dráha (v ďalšom texte budem slovo dráha používať iba v tomto význame). Napríklad na Obrázku 3.1 je dráha (t1, t2, t4, ...), alebo (t1, t3, t7, ...). (Všimnime si, že každý časový bod je označený nejakými výrazmi. Sú to výrazy, ktoré sú pravdivé v týchto časových bodoch. Tieto výrazy vlastne reprezentujú stav sveta v určitom časovom bode.)



Obrázok 3.1

LORA používa určité modálne spojky, ktoré nám umožňujú vyjadriť vlastnosti týchto dráh. Sú to časové operátory:

- "□" – znamená "vždy" - □Ψ je pravdivá teraz, ak Ψ je pravdivá teraz a v každom budúcom momente
- "○" – znamená "ďalej" – ○Ψ je pravdivá teraz, ak Ψ je pravdivá v ďalšom momente
- "◊" – znamená "niekedy" - ◊Ψ je pravdivá teraz, ak Ψ je pravdivá buď teraz, alebo v nejakom budúcom momente
- "U" – znamená "pokiaľ" – ΞUΨ je pravdivá teraz, ak Ψ je pravdivá v nejakom budúcom momente a Ξ je prevdivá dovtedy
- "W" – je ako U, ale pripúšťa možnosť, že druhý argument nebude nikdy pravdivý

Tieto operátory môžeme aj kombinovať. Napríklad ◊(xUz) je legálna LORA formula. Vo zvyšku tejto kapitoly budeme považovať za základné operátory ○ a U. Ostatné operátory odvodíme :

- ◊φ je skratka za (true U φ)
- □φ je skratka za (not(◊)not(φ))
- φ W ψ je skratka za ((φ U ψ) ∨ □φ)

Ďalšie spojky nám umožňujú kvantifikáciu dráh :

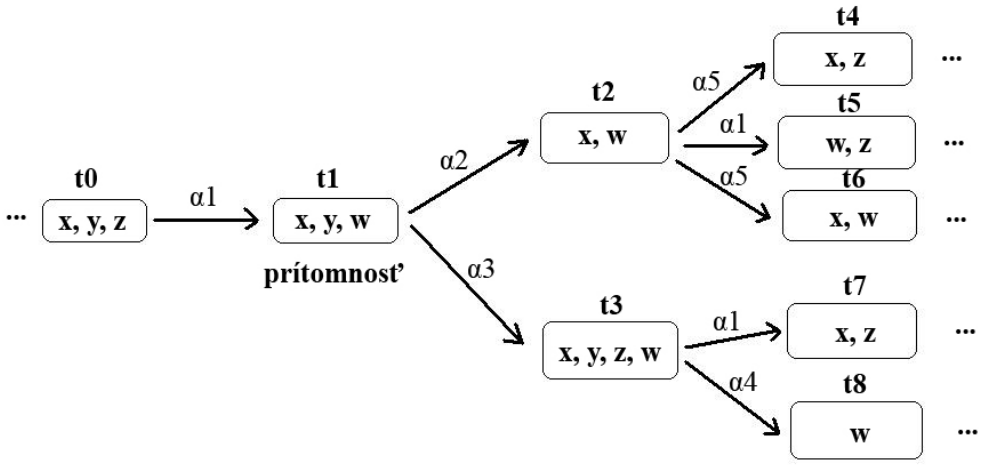
- "A" – univerzálny kvantifikátor dráh, formula AΨ znamená : "na všetkých dráhach vychádzajúcich odteraz, je Ψ pravdivé"
- "E" – existenčný kvantifikátor dráh, formula EΨ znamená : "existuje dráha vychádzajúca odteraz, na ktorej je Ψ pravdivé"

Existenčný kvantifikátor dráh je odvodený z univerzálneho kvantifikátoru dráh a to :

- Eφ je skratka za (not(A)not(φ))

3.4 Akčná komponenta

Táto komponenta nám dovoľuje reprezentovať akcie, ktoré agenti vykonávajú, a efekty týchto akcií. Najskôr zakomponujeme akcie do nášho časového modela. Základna idea je označiť prechody medzi časovými bodmi (stavmi) akciami (tak ako na Obrázku 3.2).



Obrázok 3.2

Teda, vykonanie nejakej akcie spôsobí zmenu stavu. Predpokladáme, že v danom čase sa vždy vykoná iba jedna akcia. Avšak vykonanie jednej akcie v jednom stave môže mať viacero výsledkov.

LORA poskytuje spojky, ktoré nám umožňujú vyjadriť vlastnosti akcií. Sú to :

- Formula (Happens α) znamená, že vyjadrenie akcie α sa stane ďalej.
- Formula (Achvs α φ) znamená, že akcia α nastala a dosiahla φ
- Formula (Agts α g) znamená, že skupina g je požadovaná k vykonaniu akcie α

Aby sme vyjadrili, že prvá nastane nejaká akcia, použijeme modálnu spojku Happens, ktorá má jeden argument, a to vyjadrenie akcie. Napríklad (Happens α) je pravdivá, ak α je prvá akcia, ktorá nastane na dráhe. V tomto príklade je α term jazyka LORA. Termy môžeme aj kombinovať a máme zložitejšie vyjadrenie akcie. Konštruktory pre vyjadrovanie akcií sú :

- Vyjadrenie α;α´ znamená, že α´ nasleduje po α
- Vyjadrenie α|α´ znamená, že buď α, alebo α´
- Vyjadrenie α* znamená, že α sa môže opakovať viac ako raz, alebo vôbec
- Vyjadrenie φ? Zanmená, že φ je splnené

"?" je testový konštruktor, ktorý nám umožňuje vytvárať vyjadrenia akcie, ktoré závisia na pravdivosti, alebo nepravdivosti formuly. Napríklad : Všimnime si formulu (Happens (φ?;α) | (not(φ) ?;α´)), argument tejto formuly vlastne znamená test (ak φ potom α inak α´). Vyjadrenie [(φ?;α) | (not(φ) ?)]* vlastne znamená (while φ do α).

Agenti vykonávajú akcie zvyčajne preto, aby spôsobili nejakú situáciu. (Achvs α φ) reprezentuje dva fakty :

- vyjadrenie akcie α je možné, tj. α sa vyskytuje na nejakej dráhe vychádzajúcej zo súčasneho stavu
- ak je α vykonané, máme zaručené, že φ je pravdivé

Aby sme mohli určiť agentov, ktorých potrebujeme na vykonanie nejakej postupnosti akcií, máme operátor Agts. Tento operátor má dva argumenty, vyjadrenie akcie a term označujúci množinu agentov. (Agts α g) znamená, že skupina agentov označená ako g, sú presne tí agenti, ktorých potrebujem k vykonaniu α. Ako vidíme,

LORA obsahuje aj termy na označenie skupiny agentov. Preto v LORA-e máme aj prácu s množinami. Formula ($i \in g$) tvrdí, že agent i je členom skupiny g, kde g je neprázdna skupina agentov. Podobne používame aj $\subset$ a $\subseteq$ pre vyjadrenie podmnožín, tak ako v teórii množín :

- ($g \subseteq g'$) je skratka za ($\forall i\,(i \in g) \Rightarrow (i \in g')$)
- ($g \subset g'$) je skratka za (($g \subseteq g'$) $\wedge$ not($g = g'$))

3.5 LORA abeceda

Takže abeceda LORA-y obsahuje :

1. Spočetnú množinu Pred – predikátových symbolov
2. Spočetnú množinu Const – konštantných symbolov, ktorá je zjednotením následujúcich vzájomne disjunktných množín:

- $Const_{Ag}$ – konštanty označujúce agentov
- $Const_{Ac}$ – konštanty označujúce postupnosti akcií
- $Const_{Gr}$ – konštanty označujúce množiny agentov
- $Const_{U}$ – konštanty označujúce ostatné individuá

3. Spočetnú množinu Var – premenných symbolov, ktorá je zjednotením následujúcich vzájomne disjunktných množín:

- Var_{Ag} – premenné označujúce agentov
- Var_{Ac} – premenné označujúce postupnosti akcií
- Var_{Gr} – premenné označujúce množiny agentov
- Var_{U} – premenné označujúce ostatné individuá

4. Klasické spojky:

- "$\vee$" a "not"

5. Modálne spojky

- "true" – logická konštanta pre pravdu
- "Bel" – modálna spojka pre viery agenta
- "Des" - modálna spojka pre priania agenta
- "Int" – modálna spojka pre zámery agenta
- "A" – univerzálny kvantifikátor dráh
- "U" – binárna časová spojka "pokiaľ"
- "$\circ$" – unárna časová spojka "ďalej"

6. Pridaní operátori :

- “$\in$“ – patrí do skupiny
- “Happens“ – nasledujúca akcia
- “Agts“ – agenti požadovaný na vykonanie akcie

7. Univerzálny kvantifikátor “$\forall$ “
8. Akční konštruktori :

- “;“, “|“, “*“, “?“

9. Pomocné symboly

- zátvorky

Trieda je buď Ag, Ac, Gr, alebo U (odteraz budem pojem trieda používať v tomto význame). Ak σ je trieda, potom množina $Term_\sigma$ (termov triedy σ) je definovaná ako: $Term_\sigma = Var_\sigma \cup Const_\sigma$. A množina všetkých termov je : $Term = Term_{Ag} \cup Term_{Ac} \cup Term_{Gr} \cup Term_U$.

3.6 Syntax

Syntax LORA-y je definovaná následujúcou gramatikou:

<ag-term>	::=	nejaký element z $Term_{Ag}$	/* termy agentov*/
<ac-term>	::=	nejaký element z $Term_{Ac}$	/* termy akcií*/
<gr-term>	::=	nejaký element z $Term_{Gr}$	/* termy skupín*/
<ac-exp>	::=	<ac-term>	
	|	<ac-exp> ; <ac-exp>	postupné skladanie
	|	<ac-exp> | <ac-exp>	neurčitá voľba
	|	<state-fmla>?	test akcií
	|	<ac-exp>*	opakovanie
<term>	::=	nejaký element z Term	ľubovolný term
<pred-sym>	::=	nejaký element z Pred	predikátový symbol
<var>	::=	nejaký element z Var	premenná
<state-fmla>	::=	true	pravdivá konštanta
	|	<pred-sym>(<term>, ..., <term>)	predikáty
	|	(Bel <ag-term> <state-fmla>)	formula viery

	| (Des <ag-term> <state-fmla>)	formula priani
	| (Int <ag-term> <state-fmla>)	formula zámerov
	| (Agts <ac-exp> <gr-term>)	agenti vyžadovaní pre akciu
	| (<term> = <term>)	rovnosť
	| (<ag-term> ∈ <gr-term>)	členstvo agenta v skupine
	| A<path-fmla>	kvantifikátor cez dráhy
	| not(<state-fmla>)	negácia
	| <state-fmla> ∨ <state-fmla>	disjunkcia
	| ∀ <var> <state-fmla>	kvantifikácia
<path-fmla>	::= (Happens <ac-exp>)	akcia nastane
	<state-fmla>	stavová formula
	<path-fmla> U <path-fmla>	pokiaľ
	○<path-fmla>	ďalej
	not(<path-fmla>)	negácia
	<path-fmla> ∨ <path-fmla>	disjunkcia
	∀ <var> <path-fmla>	kvantifikácia

3.7 Sémantika

Definovanie sémantiky LORA-y je pomerne zložité. Na začiatok skúsim priblížiť základnú komponentu LORA-y, a to : Agenti, akcie a čas. Nech D_{Ag} je množina všetkých agentov a nech D_{Ac} je množina všetkých akcií, ktoré môžu títo agenti vykonať. Na reprezentáciu nášho časového modelu použijem vetviacu časovú štruktúru, ktorá je úplným, spätne lineárnym, orientovaným grafom nad množinou T, kde T je množina časových bodov. Relácia *R* je úplná, ak každý uzol *R* má aspoň jedného následníka. Teda binárna relácia *R* je úplná, ak spĺňa : $\forall t\,(t \in T) \Rightarrow \exists t'\,[\,t' \in T \wedge (t, t') \in R]$. Nech $R \subseteq T \times T$ je úplná vetviaca časová štruktúra, ktorá kóduje všetky možnosti, do ktorých môže systém dospieť. Hrany v *R* predstavujú vykonávanie základných akcií agentami v systéme a označíme tieto hrany akciami, ktoré predstavujú, pomocou funkcie Act : $R \rightarrow D_{Ac}$. Každá akcia je spojená s jedným agentom, to nám určuje funkcia Agt : $D_{Ac} \rightarrow D_{Ag}$.

Možné svety:

Stav agenta je definovaný pomocou jeho viery, prianí a zámerov. Ich sémantika je daná pomocou Kripkeho sémantiky (možné svety). Agentové viery v danej situácii sú charakterizované množinou situácií – tých, ktoré sú konzistentné s agentovými vierami. Agent teda verí φ, ak φ je pravdivé vo všetkých týchto možných situáciach (tieto situácie označujeme ako situácie prijateľných vier). Podobne agentové priania v danej situácii sú charakterizované množinou situácií – tých, ktoré sú konzistentné s agentovými prianiami (tieto situácie označujeme ako situácie prijateľných prianí). A agentové zámery v danej situácii sú charakterizované množinou situácií – tých, ktoré sú konzistentné s agentovými zámermi (tieto situácie označujeme ako situácie prijateľných zámerov).

Svety v LORA sú vetviace časové štruktúry. Tieto štruktúry reprezentujú nielen agentovu neistotu o tom, ako vyzerá svet teraz, ale aj o tom, ako sa vyvinie.

Formálne : Svet je dvojica $<T', R'>$, kde $T' \subseteq T$ je neprázdna množina časových bodov a $R' \subseteq R$ je vetviaca časová štruktúra nad T'.

Nech W je množina všetkých svetov nad T, teda $W = \{<T', R'> \mid T' \subseteq T, R' \subseteq R, T' = (\text{dom } R' \cup \text{ran } R')\}$, kde dom R' je definičný obor R' a ran R' je obor hodnot R'.

Ak $w \in W$ je svet, potom T_w je množina časových bodov vo w a R_w je množina vetviacich časových relácií vo w. Dvojica $<w, t>$, kde $w \in W$ a $t \in T_w$ je situácia. Množina všetkých situácií vo w je : $S_w = \{<w, t> \mid w \in W \text{ a } t \in T_w\}$.

Na charakterizovanie vier každého agenta používame funkciu $\beta : D_{Ag} \rightarrow P(W \times T \times W)$, pričom $P(X)$ označuje potenčnú množinu X (toto značenie potenčnej množiny budeme používať aj v nasledujúcom texte). β je relácia prijateľných vier.

Ďalej $\beta_{w,t}(i) = \{w' \mid <w, t, w'> \in \beta(i)\}$ označuje množinu prijateľných svetov pre agenta i v situácii $<w, t>$.

Relácie prijateľných vier (β) musia spĺňať nasledujúce vlastnosti :

- (vlastnosť svetovo/časového bodu kompatibility) Ak svet w′ je prijateľný pre agenta v situácii $<w, t>$, potom t musí byť časový bod aj vo w aj w′. Teda ak $w' \in \beta_{w,t}(i)$ potom $t \in w$ a zároveň $t \in w'$.

- (následná vlastnosť) Pre každú situáciu <w, t> existuje nejaký svet w´ taký, že $w' \in \beta_{w,t}(i)$
- (tranzitivita) Ak $w' \in \beta_{w,t}(i)$ a $w'' \in \beta_{w',t}(i)$ potom $w'' \in \beta_{w,t}(i)$
- (Euklidova vlastnosť) Ak $w' \in \beta_{w,t}(i)$ a $w'' \in \beta_{w,t}(i)$ potom $w' \in \beta_{w'',t}(i)$

Podobne ako pri vierach postupujeme aj v prípade prianí a zámerov. Teda priania agenta charakterizujeme funkciou $\Delta : D_{Ag} \rightarrow P(W \times T \times W)$ a zámery agenta charakterizujeme funkciou $Ɩ : D_{Ag} \rightarrow P(W \times T \times W)$. Δ aj $Ɩ$ majú vlastnosť svetovo/časového bodu kompatibility a následnú vlastnosť. Ďalej, svety prijateľných prianí agenta i v situácii <w, t> budeme označovať $\Delta_{w,t}(i)$ a svety prijateľných zámerov agenta i v situácii <w, t> budeme označovať $Ɩ_{w,t}(i)$. $\Delta_{w,t}(i)$ a $Ɩ_{w,t}(i)$ sú definované podobne jako v prípade $\beta_{w,t}(i)$.

Pre prácu s vetviacou časovou štruktúrou definujeme nasledujúce označenia. Nech $w \in W$ je svet. Potom dráha cez w je postupnosť $(t_0, t_1, ...)$ taká, že pre všetky $u \in N$, máme $(t_u, t_{u+1},) \in R_w$. Nech paths(w) označuje množinu všetkých dráh cez w. Ak p je dráha a $u \in N$, potom p(u) označuje (u + 1) – tý element p (p(0) je prvý časový bod, p(1) druhý ...). Ak p je dráha a $u \in N$, potom dráhu získanú z p odstránením prvých u časových bodov označujeme $p^{(u)}$.

Ďalej potrebujeme identifikovať objekty, na ktoré môžeme odkazovať. LORA jazyk obsahuje termy, ktoré symbolizujú tieto objekty. Doménou LORA-y sú práve tieto objekty. Táto doména obsahuje agentov (D_{Ag}), akcie (D_{Ac}), skupiny agentov (D_{Gr}, kde $D_{Gr} \subseteq P(D_{Ag})$) a ostatné objekty (D_U). Formálne : Doména D je štruktúra : $D = < D_{Ag}, D_{Ac}, D_{Gr}, D_U >$, kde

- $D_{Ag} = \{1, \ldots, n\}$ – neprázdna množina agentov
- $D_{Ac} = \{\alpha, \alpha', ... \}$ – neprázdna množina akcií
- $D_{Gr} = P(D_{Ag}) \setminus \{\Theta\}$, $P(D_{Ag})$ je potenčná množina množiny D_{Ag} a Θ je prázdna množina
- D_U je neprázdna množina ostatných individuí.

Ak D je doména $< D_{Ag}, D_{Ac}, D_{Gr}, D_U >$, tak definujeme $\underline{D}$ ako $\underline{D} = D_{Ag} \cup D_{Ac}^* \cup D_{Gr} \cup D_U$ a ak $u \in N$, potom $\underline{D}^u$, značí množinu u – tic nad $\underline{D}$, teda $\underline{D}^u = \underbrace{\underline{D} \times \ldots \times \underline{D}}_{u - krát}$

Aby sme dali význam LORA formulám, potrebujeme rôzne funkcie, ktoré spájajú symboly jazyka s objektmi v doméne. Interpretáciou predikátov ϕ je funkcia, ktorá prideľuje každému predikátu v každom časovom bode množinu u-tic, ktorá reprezentuje rozšírenie predikátu v časovom bode. Teda $\Phi : Pred \times T \rightarrow P(\bigcup_{u \in N} \underline{D}^u)$.
Funckia ϕ musí zachovávať aritu, napríklad ak ϕ prideľuje predikátovému symbolu P množinu u-tic, potom arita P by mala byť u. Ak arity(P) = u potom $\phi(P,t) \subseteq \underline{D}^u$ pre všetky $t \in T$.

Interpretáciou konštánt C je funkcia, ktorá zoberie konštantu a časový bod a priradí označenie týmto konštantám v týchto časových bodoch. Teda $C : Const \times T \rightarrow \underline{D}$. Táto funkcia musí zachovávať triedy, teda ak $x \in Const_\sigma$ a $t \in T$ potom $C(x, t) \in D_\sigma$. Poznamenajme však, že konštanty nemusia mať fixnú interpretáciu (konštanta x označujúca $d \in \underline{D}$, v čase $t \in T$ môže označovať $d' \in \underline{D}$, v čase $t' \in T$ a $d \neq d'$), ale pokiaľ nebude uvedené inak, tak predpokladáme, že konštanty majú fixnú interpretáciu.

Funkcia V je priradenie premennej. Prideľuje premenným prvok domény. $V : Var \rightarrow \underline{D}$. Táto funkcia musí taktiež zachovávať triedy a premenné majú fixnú interpretáciu.

Ako označenie ľubovoľného termu používame funkciu $[[...]]_{V,C}$. Ak V je priradenie premennej a C je interpretácia konštánt, potom $[[...]]_{V,C} : Term \times T \rightarrow \underline{D}$, interpretuje term zviazaný s V a C nasledovne : $[[x, t]]_{V,C} = \begin{cases} C(x, t) & ak \quad x \in Const \\ V(t) & inak \end{cases}$.
Ak bude jasný časový bod, tak túto funkciu budeme zjednodušene zapisovať ako [[x]].

MODEL

LORA model je štruktúra $M = <T, R, W, D, Act, Agt, \beta, \Delta, Į, C, \phi>$, kde

- T je množina všetkých časových bodov

- $R \subseteq T \times T$ je úplná, spätne lineárna vetviaca časová štruktúra nad T
- W je množina svetou nad T
- $D = < D_{Ag}, D_{Ac}, D_{Gr}, D_U >$ je doména
- $Act : R \rightarrow D_{Ac}$ je priradenie akcie každej hrane z R
- $Agt : D_{Ac} \rightarrow D_{Ag}$ je priradenie agenta každej akcii
- $\beta : D_{Ag} \rightarrow P(W \times T \times W)$ je relácia prijateľných vier
- $\Delta : D_{Ag} \rightarrow P(W \times T \times W)$ je relácia prijateľných prianí
- $Ɩ : D_{Ag} \rightarrow P(W \times T \times W)$ je relácia prijateľných zámerov
- $C : C : Const \times T \rightarrow \underline{D}$ je interpretácia konštánt
- $\Phi : Pred \times T \rightarrow P(\bigcup_{u \in N} \underline{D}^u)$ je interpretácia predikátov

Sémantika dráhových formúl :

Sémantika LORA-y je definovaná v dvoch častiach. Pre dráhové formuly a pre stavové formuly. Sémantika dráhových formúl je daná pomocou splňovacej relácie pre dráhové formuly "$|=_P$", ktorá platí medzi interpretáciou dráhových formúl a dráhovými formulami. Interpretácia dráhových formúl je štruktúra <M, V, w, p>, kde M je model, V je priradenie premennej, w svet v M a p je dráha cez w. Ak <M, V, w, p> $|=_P \varphi$, hovoríme, že <M, V, w, p> splňuje φ (alebo φ je pravdivá v <M, V, w, p>). Poznamenajme, že $|=_S$ je splňovacia relácia pre stavové formuly.
Pravidlá definujúce splňovaciu reláciu pre dráhové formuly sú nasledujúce :

<M, V, w, p> $|=_P \varphi$	ak <M, V, w, p(0)> $|=_S \varphi$, kde φ je stavová formula
<M, V, w, p> $|=_P$ not(φ)	<M, V, w, p> $\mid \neq_P \varphi$
<M, V, w, p> $|=_P \varphi \vee \psi$	ak <M, V, w, p> $|=_P \varphi$ alebo <M, V, w, p> $|=_P \psi$
<M, V, w, p> $|=_P \forall x \varphi$	ak <M, V : {x $\rightarrow$ d}, w, p> $|=_P \varphi$, pre všetky d $\in \underline{D}$ také, že x a d sú z rovnakej triedy
<M, V, w, p> $|=_P \varphi U \psi$	ak $\exists u \in N$ také, že <M, V, w, $p^{(u)}$> $|=_P \psi$ a zároveň $\forall v \in N$, ak $(0 \leq v < u)$, potom <M, V, w, $p^{(v)}$> $|=_P \varphi$
<M, V, w, p> $|=_P \circ\varphi$	ak <M, V, w, $p^{(1)}$> $|=_P \varphi$
<M, V, w, p> $|=_P$ (Happens α)	ak $\exists u \in N$ také, že occurs(α, p, 0, u)

V poslednom prípade používame pomocnú definíciu occurs. Táto definícia sa používa na definovanie operátora Happens. Značenie occurs(α, p, u, v) znamená, že akcia α sa vyskytuje medzi časmi p a u (p, u $\in$ N) na dráhe p. occurs je meta-predikát, teda používame ho v našom popise LORA-y, ale nepoužíva sa vo formulách LORA-y. Predikát occurs je definovaný piatimi pravidlami (pre vykonanie primitívnej akcie a pre každý konštruktor vyjadrovania akcií).

1. pravidlo vykonania primitívnej akcie :
   occurs(α, p, u, v) ak v = u + k, a zároveň $[[\alpha, p(u)]] = \alpha_1, ..., \alpha_k$, a $Act(p(u), p(u + 1)) = \alpha_1, ..., Act(p(u + k – 1), p(u + k)) = \alpha_k$, kde ($\alpha \in Term_{Ac}$)
2. pravidlo nasledujúcej akcie :
   occurs(α;α´, p, u, v) ak $\exists$ n $\in$ {u, ..., v} také, že occurs(α, p, u, v) a zároveň occurs(α, p, u, v)
3. pravidlo neurčitej voľby :
   occurs(α | α´, p, u, v) ak occurs(α, p, u, v), alebo occurs(α´, p, u, v)
4. pravidlo opakovania
   occurs(α*, p, u, v) ak u = v, alebo occurs(α;(α*), p, u, v)
5. pravidlo testu
   occurs(φ?, p, u, v) ak <M, V, w, p(u)> $\models_S$ φ.

Sémantika stavových formúl :

Sémantika stavových formúl je daná pomocou splňovacej relácie pre stavové formuly "$\models_S$", ktorá platí medzi interpretáciou stavových formúl a stavovými formulami. Interpretácie stavových formúl sú štruktúry <M, V, w, t>, kde M je model, V je priradenie premennej, w svet v M a t $\in T_w$ je časový bod vo w. Ak <M, V, w, t> $\models_S$ φ, hovoríme, že <M, V, w, t> splňuje φ (alebo φ je pravdivá v <M, V, w, t>).

Pravidlá definujúce splňovaciu reláciu pre stavové formuly sú nasledujúce :

<M, V, w, t> $\models_S$ true	vždy
<M, V, w, t> $\models_S$ Pred(x_1, ..., x_n)	ak <[[x_1]], ..., [[x_n]]> $\in$ ϕ(Pred, t)
<M, V, w, t> $\models_S$ not(φ)	ak <M, V, w, t> $\not\models_S$ φ
<M, V, w, t> $\models_S$ φ $\vee$ ψ	ak <M, V, w, t> $\models_S$ φ, alebo <M, V, w, t> $\models_S$ ψ

$\langle M, V, w, t\rangle \models_S \forall x\,.\,\varphi$	ak $\langle M, V : \{x \to d\}, w, t\rangle \models_S \varphi$, pre všetky $d \in \underline{D}$ také, že x a d sú z rovnakej triedy
$\langle M, V, w, t\rangle \models_S (\text{Bel } i\ \varphi)$	ak $\forall w' \in W$, ak $w' \in \beta_{w,t}([[i]])$, tak $\langle M, V, w', t\rangle \models_S \varphi$
$\langle M, V, w, t\rangle \models_S (\text{Des } i\ \varphi)$	ak $\forall w' \in W$, ak $w' \in \Delta_{w,t}([[i]])$, tak $\langle M, V, w', t\rangle \models_S \varphi$
$\langle M, V, w, t\rangle \models_S (\text{Int } i\ \varphi)$	ak $\forall w' \in W$, ak $w' \in \text{I}_{w,t}([[i]])$, tak $\langle M, V, w', t\rangle \models_S \varphi$
$\langle M, V, w, t\rangle \models_S (\text{Agts } \alpha\ g)$	ak $AGTS(\alpha, t) = [[g]]$
$\langle M, V, w, t\rangle \models_S (x = x')$	ak $[[x]] = [[x']]$
$\langle M, V, w, t\rangle \models_S (i \in g)$	ak $[[i]] \in [[g]]$
$\langle M, V, w, t\rangle \models_S A\varphi$	ak $\forall p \in paths(w)$, ak $p(0) = t$, tak $\langle M, V, w, p\rangle \models_P \varphi$

Ešte jedna definícia funkcie AGTS(α, t) :

$$AGTS(\alpha, t) = \begin{cases} \{Agt(\alpha_1),..., Agt(\alpha_k)\} & \text{kde } [[\alpha, t]] = \alpha_1, ..., \alpha_k \\ AGTS(\alpha_1, t) \cup AGTS(\alpha_2, t) & \text{kde } \alpha = \alpha_1; \alpha_2, \text{ alebo } \alpha = \alpha_1 \mid \alpha_2 \\ AGTS(\alpha_1, t) & \text{kde } \alpha = \alpha_1 * \\ \theta & \text{kde } \alpha \text{ je tvaru } \varphi? \end{cases}$$

3.8 Základné vlastnosti LORA-y

Po tom, ako sme si definovali syntax a sémantiku LORA-y, ukážeme ešte nejaké vlastnosti tejto logiky. LORA vlastne spája dva jazyky. Pre stavové formuly a pre dráhové formuly. Hovoríme, že dráhová formula φ je splniteľná, ak pre nejakú štruktúru $\langle M, V, w, p\rangle$ platí $\langle M, V, w, p\rangle \models_P \varphi$ a hovoríme, že φ je platná, ak pre všetky štruktúry $\langle M, V, w, p\rangle$ platí $\langle M, V, w, p\rangle \models_P \varphi$ (píšeme skrátene $\models_P \varphi$). Podobne definujeme aj pre stavové formuly. Hovoríme, že stavová formula φ je splniteľná, ak pre nejakú štruktúru $\langle M, V, w, t\rangle$ platí $\langle M, V, w, t\rangle \models_S \varphi$ a hovoríme, že φ je platná, ak pre všetky štruktúry $\langle M, V, w, t\rangle$ platí $\langle M, V, w, t\rangle \models_S \varphi$ (píšeme skrátene $\models_S \varphi$).

Stavové formuly sú zároveň dráhové formuly a sémantické pravidlá pre vyhodnocovanie sú pre stavové formuly rovnaké ako pre dráhové. Teda ak $\models_S \varphi$, potom $\models_P \varphi$. Okrem iného to znamená, že určité formuly sú platné len podľa tvaru týchto formúl, bez ohľadu na to, či sú to stavové alebo dráhové formuly (napríklad

φ ∨ not(φ)). V takomto prípade píšeme |= φ. Podobne platí aj pre splniteľné formuly: ak φ je splniteľná stavová formula potom φ je splniteľné dráhová formula.
Pre vzťah medzi LORA-ou a klasickou logikou platí :
Ak φ je substitúcia tautológie výrokovej logiky, tak |= φ. LORA je zovšeobecnením výrokovej logiky.

Na charakterizáciu vier, priani a zámerov používame vety LORA logiky. Tieto vety nám popisujú rôzne vlastnosti agentov. Skúsim ukázať nejaké vety, ktoré platia pre operátory Bel, Des a Int.

.

Bel :

1. $\models_S$ (Bel i φ) ⇒ not(Bel i not(φ))
2. $\models_S$ (Bel i φ) ⇒ (Bel i (Bel i φ))
3. $\models_S$ not(Bel i φ) ⇒ (Bel i not(Bel i φ))
4. $\models_S$ (Bel i (φ ⇒ ψ)) ⇒ ((Bel i φ) ⇒ (Bel i ψ))
5. Ak $\models_S$ φ potom $\models_S$ (Bel i φ)

Des :

1. $\models_S$ (Des i φ) ⇒ not(Des i not(φ))
2. $\models_S$ (Des i (φ ⇒ ψ)) ⇒ ((Des i φ) ⇒ (Des i ψ))
3. Ak $\models_S$ φ potom $\models_S$ (Des i φ)

Int :

1. $\models_S$ (Int i φ) ⇒ not(Int i not(φ))
2. $\models_S$ (Int i (φ ⇒ ψ)) ⇒ ((Int i φ) ⇒ (Int i ψ))
3. Ak $\models_S$ φ potom $\models_S$ (Int i φ)

Nakoniec ešte ukážem nejaké vety o vyjadrovaní akcií a o operátorovi Happens :

1. $\models_P$ (Happens α ; α′) ⇔ (Happens α ; (Happens α′)?)
2. $\models_P$ (Happens α | α′) ⇔ ((Happens α) ∨ (Happens α′))
3. $\models_P$ (Happens α?) ⇔ α
4. $\models_P$ (Happens α*)

5. $\models_P \exists \alpha$ (Happens α)

Tieto vety sú pomerne jednoduché a dajú sa pomocou nich zaviesť do systému také vlastnosti, ktoré požadujeme aby naši agenti a náš systém mal.
Zároveň treba dodať, že viery, priania a zámery v skutočnom živote (napríklad u ľudí) sú príliš zložité a neurčité na to, aby ich LORA úplne zachytila. Formálna teória, ktorá by to dokázala, by zasa bola príliš zložitá a skomplikovala a zneprehľadnila by všetko ostatné. Preto pre popis racionálnych agentov nám stačí logika LORA.
Všimnime si, že axiomatizácia multi-modálnej logiky je v súčasnosti nedostatočne definovaná a taktiež nemáme ani dostatočnú axiomatizáciu logiky LORA. Avšak napriek tomuto faktu môže byť LORA použitá ako logika pre charakterizovanie racionálnych agentov.

Problém vedľajšieho efektu

Ukážem ešte jednoduchý príklad, ako sa dá v LORA-e riešiť problém vedľajšieho efektu, ktorý sa bližšie rozoberá v [15]. Problém : Napríklad: Môj zámer je ísť k doktorovi, aby ma zaočkoval proti chrípke. Som si vedomý toho, že očkovanie bolí. Je môj zámer trpieť? Odpoveď je, že nechcem trpieť. Teda, ak má agent i zámer φ, nemusí mať zároveň zámer vedľajšieho efektu φ. Tento problém sa v LORA-e rieši pomerne jednoducho. Stačí, ak budeme požadovať, aby schéma $((\text{Int i } \varphi) \wedge (\text{Bel i } (\varphi \Rightarrow \psi)) \wedge \text{not}(\text{Int i } \psi))$ bola splniteľná. Tento príklad ilustruje, že riešenia niektorých známych problémov sú pomocou LORA-y vcelku jednoduché.

Konkrétnych racionálnych agentov popisujem pomocou LORA formulí. Pre mňa podstatné situácie nastávajú vtedy, keď agentov zámer je nejako komunikovať s ostatnými agentami. Napríklad agent A má zámer, aby druhý agent B mal vieru o určitej informácii. V tomto prípade môže agent A poslať agentovi B správu s s touto určitou informáciou. Tu nastupuje problematika komunikácie medzi agentami.

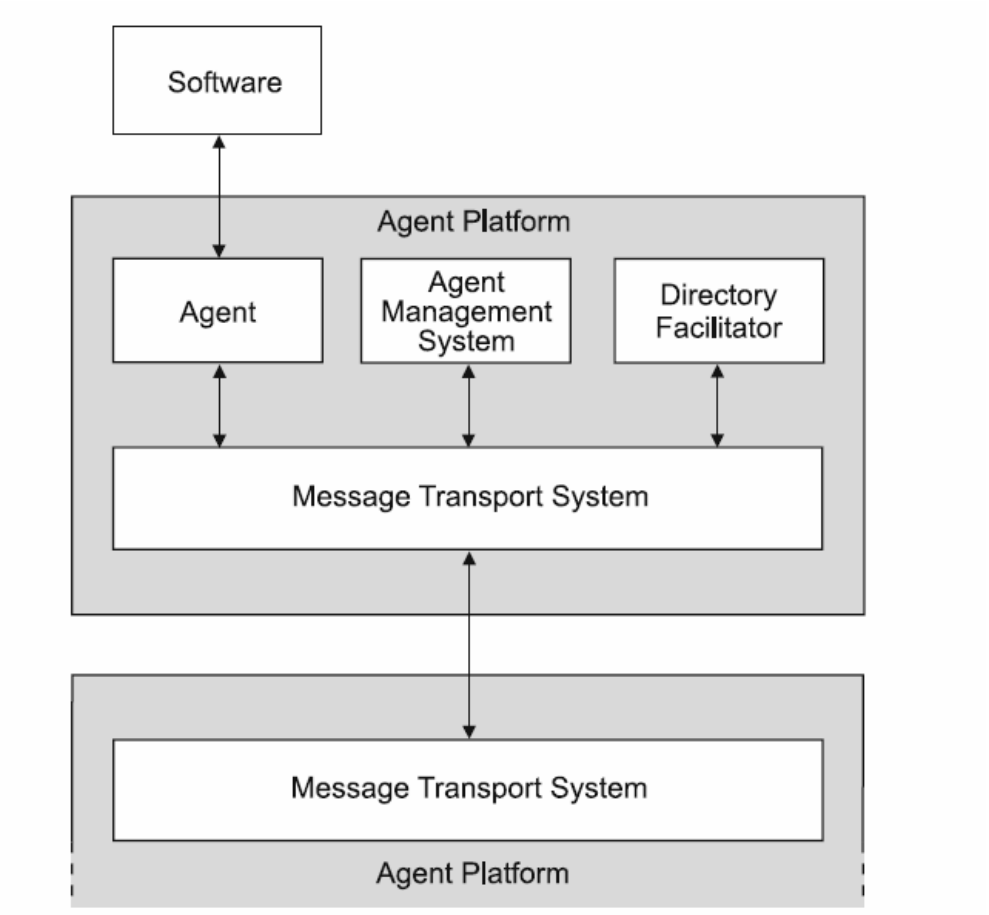
4. Kapitola : Architektúra a implementácia multi-agentného systému

Zatiaľ sme sa venovali len agentom, špeciálne racionálnym agentom, ukázali sme si logiku LORA, ktorá slúži na ich popis. Ďalej musíme agentov umiestniť do nejakého prostredia (sveta). Keď je toto prostredie zložené z viacerých agentov, ktorí vnímajú toto prostredie a sú schopní vzájomnej komunikácie, hovoríme o multi-agentných systémoch. Multi-agentné systémy (ďalej MAS) sú teda systémy zložené z viacerých agentov, ktorí medzi sebou komunikujú. MAS sa väčšinou používajú na riešenie problémov, ktoré sú príliš zložité pre jedného agenta. Agenti v MAS majú určité vlastnosti (okrem vlastností agenta, ktoré sme rozoberali vyššie, ako je autonómia, reaktívnosť, proaktívnosť a sociálna schopnosť) :

- majú lokálny pohľad na svet – to znamená, že neexistuje agent, ktorý by mohol pozorovať celý systém, alebo je tento systém príliš zložitý, aby ho nejaký agent mohol pozorovať
- decentralizácia – neexistuje žiadny riadiaci agent (agent, ktorý by riadil ostatných agentov)

Keď máme funkčný MAS, tak jedna z jeho najväčších výhod je flexibilita – do tohto systému môžeme pridávať agentov (taktiež môžeme agentov zo systému odoberať), môžeme meniť agentov bez toho, aby sme museli detailne prepisovať celú aplikáciu. Jeden z najznámejších, najrozšírenejších a najpoužívanejších príkladov MAS, by mohol byť internet (v tomto prípade sú agentami internetové stránky). Pri návrhu MAS musíme začať jeho architektúrou.

V mojej práci sa zaoberám hlavne FIPA architektúrou. FIPA je organizácia, ktorá vyvíja štandardy pre agentne orientované technológie. FIPA si všíma aj štandardy ostatných technológií a snaží sa ich zapájať do svojich štandardov. Hlavné súčasti multi-agentného systému podľa FIPA sú zobrazené na obrázku 4.1 (tento obrázok som našiel v [18]).



Obrázok 4.1

Agent : je výpočtový proces, ktorý implementuje autonómnu a komunikačnú funkcionalitu aplikácie. Agent je základný prvok Agent Platform (AP), ktorý poskytuje jednu, alebo viac služieb. Agent musí mať aspoň jedného vlastníka a musí podporovať aspoň jedno označenie identifikátorom (AID). AID je unikátne v celom agentnom priestore. Agent môže mať viacero adries, na ktorých ho môžeme kontaktovať.

Directory Facilitator (DF): je voliteľná komponenta vo FIPA architektúre, ktorá ak je prítomná, stará sa o informácie o službách, ktoré poskytujú ostatní agenti. Sú to vlastne také zlaté stránky, do ktorých si agenti môžu zaregistrovať svoje služby, alebo agenti môžu požiadať DF o nájdenie služieb ponúkaných

ostatnými agentami. Na jednej AP môže existovať viacero DF a tieto DF môžu vzájomne spolupracovať.

Agent Management System (AMS): je povinná komponenta vo FIPA architektúre, ktorý vykonáva hlavnú a stálu kontrolu nad vstupom a používaním agentnej platformy (AP). V jednej AP je práve jeden AMS. V AMS sa udržiavajú mená a adresy agentov, ktorí sú v AMS zaregistrovaní. Každý agent sa musí zaregistrovať v AMS (svojej platformy), aby dostal platný AID.

Message Transport Service (MTS): je štandardná komunikačná metóda medzi agentami na rôznych AP.

Agent Platform (AP): je fyzická štruktúra, v ktorej môžu byť agenti umiestnení.

Keď už vieme, ako bude architektúra nášho systému vyzerať, potrebovali by sme ju implementovať. Ako implementáciu takéhoto modelu som použil JADE. Tento freeware (od TELECOM ITALIA) zjednodušuje implementáciu MAS. Je to vlastne middle-ware, ktorý vyhovuje FIPA štandardom, a ktorý má viacero grafických pomôcok pre vývoj a ladenie agentov. Je plne naprogramovaný v Java jazyku. AP môže byť rozložená cez viaceré počítače, s rôznymi operačnými systémami. Konfigurácia AP môže byť ovládaná pomocou GUI (okna), ktoré nám pomáha sprehľadňovať celý systém. Taktiež môžeme počas behu systému presúvať agentov z jedného počítača na druhý, vtedy, keď je to potrebné.

Aby som priblížil, aké výhody a uľahčenie práce nám JADE poskytuje, skúsim popísať príklad, ako vytvoriť vlastného agenta (nejedná sa o racionálneho agenta, ale o "hlúpeho" agenta). Chcem vytvoriť triviálneho agenta, ktorý pri svojom "narodení" vypíše na výstup "Ahoj svet". Keďže celé JADE je naprogramované v JAVA-e, tak je rozumné, aby sme aj nášho agenta programovali v JAVA jazyku. Vytvorenie JADE agenta je jednoduché. Stačí rozšíriť triedu jade.core.Agent. Táto trieda ako mnohé iné sú v JADE už naprogramované a programátor ich môže meniť, ak chce. Programátor by mal hlavne preprogramovať metódu setup(), ktorá sa spustí pri "narodení" agenta.

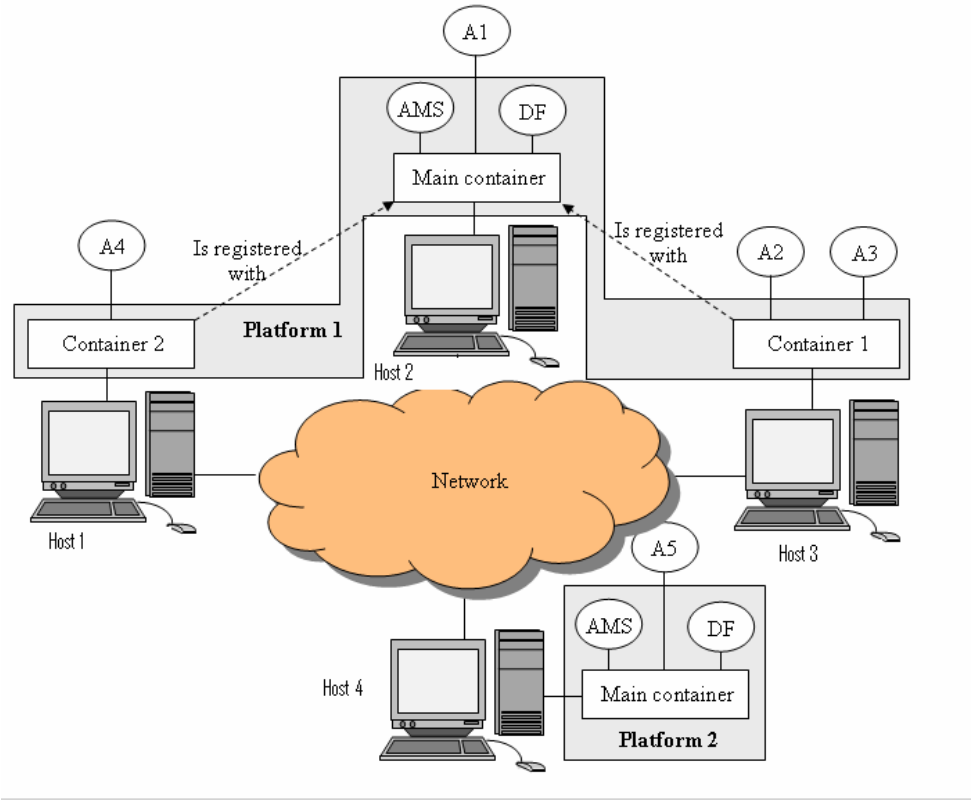
Teda pre triedu môjho agenta stačí skompilovať nasledujúci kód a mám JADE triedu pre agenta :

```
1. import jade.core.Agent;
2. public class MojAgent extends Agent {
3.         protected void setup() {
4.                 System.out.println("Ahoj svet");
5.         }
6. }
```

Naprogramovanú triedu JADE agenta už mám (je to trieda MojAgent). Z tejto triedy môžem vytvoriť ľubovoľný počet agentov. Títo agenti musia existovať v nejakom systéme. Systém JADE môžem spúšťať z príkazového riadku pomocou príkazu jade.Boot. Keď navyše pridáme tomuto príkazu argument HlupyAgent:MojAgent, po vytvorení systému sa mi okamžite vytvorí agent s menom HlupyAgent, ktorý je odvodený z triedy MojAgent a teda neurobí nič iné, len vypíše "Ahoj svet". Na tomto príklade je vidno, že programátor a vývojár MAS sa môže venovať hlavne vývoju agentov bez toho, aby musel programovať základy a prostredie, v ktorom budú jeho agenti pracovať.

JADE bola vyvinutá pre účely telekomunikačnej firmy. Za agentov sa považovali mobilné telofóny (alebo iné telekomunikačné zariadenia). Mobilné telofóny môžu byť považované za agentov, pretože majú vlastnosti, ktoré by agenti mali mať. Pri troche predstavivosti si vieme predstaviť, že mobilné telefóny sú autonómne, sú schopné vnímať a reagovať na vonkajšie zmeny a sú schopné komunikácie. Preto svet mobilov tvorí MAS. Avšak mobilné telefóny nemajú zatiaľ skoro žiadnu inteligenciu, alebo racionalitu. Preto sa v JADE nič nespomína o racionálnych (ani o logických) agentoch a ani nie sú žiadne pomôcky pre prácu s týmito agentami. Napriek tomu, že JADE nebolo vyvinuté špeciálne pre racionálnych (logických) agentov, dajú sa títo agenti v JADE implementovať, a je len na programátoroch a vyvojároch agentov, aké vlastnosti budú títo agenti mať.

Príklad JADE architektúry je zobrazený na obrázku 4.2 (obrázok som našiel v [19]).



Obrázok 4.2

Každé spustenie JADE vytvára nový kontajner, ktorý môže obsahovať niekoľko agentov. Pri prvom spustení JADE sa nám automaticky vytvorí Hlavný kontajner, ktorý vždy obsahuje aspoň dvoch špeciálnych agentov a to AMS a DF, čo sú agenti definovaní vo FIPA architektúre. Platforma je množina aktívnych kontajnerov, pričom práve jeden Hlavný kontajner je v jednej platforme a všetky ostatné kontajnery sú pri štarte zaregistrované v tomto Hlavnom kontajneri. Ak vytvoríme niekde v sieti ďalší Hlavný kontajner, znamená to vytvorenie novej platformy.

Agentov v JADE vytvára AMS, ktorý im určí unikátny identifikátor (AID, ktorý má formu "meno_agenta@názov_platformy"). AMS sa taktiež stará o "zabíjanie" agentov. Každý agent v JADE je jedno Java vlákno. Na plnenie agentových úloh sa využívajú takzvané "behaviour". Každá úloha môže byť

implementovaná ako behaviour objekt. Agent môže vykonávať niekoľko behaviour naraz a o rozvrhovanie týchto behaviour sa stará špeciálny agent, ktorý je skrytý pre vonkajší svet. Nám stačí informácia, že tento agent rozvrhuje behaviour-y cyklicky, bez priority.

Komunikácia agentov je založená na "Speech Act" teórii [16]. Táto teória je založená na predpoklade, že rečové prejavy, ktoré sa nazývajú "Speech Acts" majú charakteristiku akcie, teda môžu meniť stav sveta, podobne ako fyzické akcie. Napríklad : Keď farár povie : "A týmto Vás prehlasujem za muža a ženu.", alebo vyhlásenie vojny. Tieto výroky určite menia svet. Existuje viacero výkonných slov, ktoré korešpondujú s rôznymi typmi rečových prejavov, ako napríklad: REQUEST, INFORM ... Tieto slová prebrala aj FIPA a v správe, ktorá je naprogramovaná v ACL jazyku, to je výkonný parameter. Ďalej táto teória určuje tri typy podmienok, ktoré musia byť splnené (keď medzi sebou komunikujú hovorca a poslucháč) :

1. Normálne I/O podmienky – určujú stav, v ktorom je poslucháč schopný prijať správu
2. Prípravné podmienky – určujú, čo má byť splnené vo svete, aby hovorca správne vybral svoj rečový prejav. Teda poslucháč musí byť schopný vykonať akciu a hovorca musí veriť, že poslucháč je schopný vykonať túto akciu. To ešte ale nemusí znamenť, že poslucháč túto akciu vykoná
3. Úprimné podmienky – tieto podmienky rozlišujú úprimný a neúprimný rečový prejav. Neúprimný rečový prejav sa vyskytuje, ak hovorca pošle žiadosť o vykonanie akcie, ale v skutočnosti nechce, aby bola akcia vykonaná.

Avšak tieto typy podmienok nie sú veľmi praktické (napríklad s úprimnými podmienkami sa nepracuje, pretože sa predpokladá úprimnosť agentov). V praxi sa skôr používajú pre a post podmienky, tak ako napríklad v teórii plánovania (v stručnosti : ak sú splnené pre-podmienky, tak nastane určitý prejav, ktorý spôsobí post-podmienky).

Agenti medzi sebou komunikujú pomocou správ. Komunikácia v JADE medzi agentami je transparentná a nezáleží na tom, či sú agenti v rôznych kontajneroch v rôznych platformách. Teda programátor sa nemusí starať o mechanizmus doručovania

správ. Správa je naprogramovaná v ACL jazyku. Správa je podľa tohto jazyka štruktúrovaná, obsahuje určité parametre. Jediný povinný parameter je výkonný parameter (performative), ktorý určuje typ komunikačnej akcie. Samozrejme predpokladá sa, že správa bude obsahovať aj ďalšie parametre, ako je : príjemca, obsah… Všetky parametre sú :

Performative	výkonný parameter
Content	obsah správy
Sender	odosielateľ
Receiver	príjemca
reply-to	nasledujúca odpoveď
Language	jazyk obsahu správy
Encoding	kódovanie obsahu správy
Ontology	ontológia obsahu správy
Protocol	protokol
conversation-id	Id konverzácie
reply-with	označenie odpovede
in-reply-to	označenie od predošlého reply-with
reply-by	čas do kedy treba doručiť odpoveď

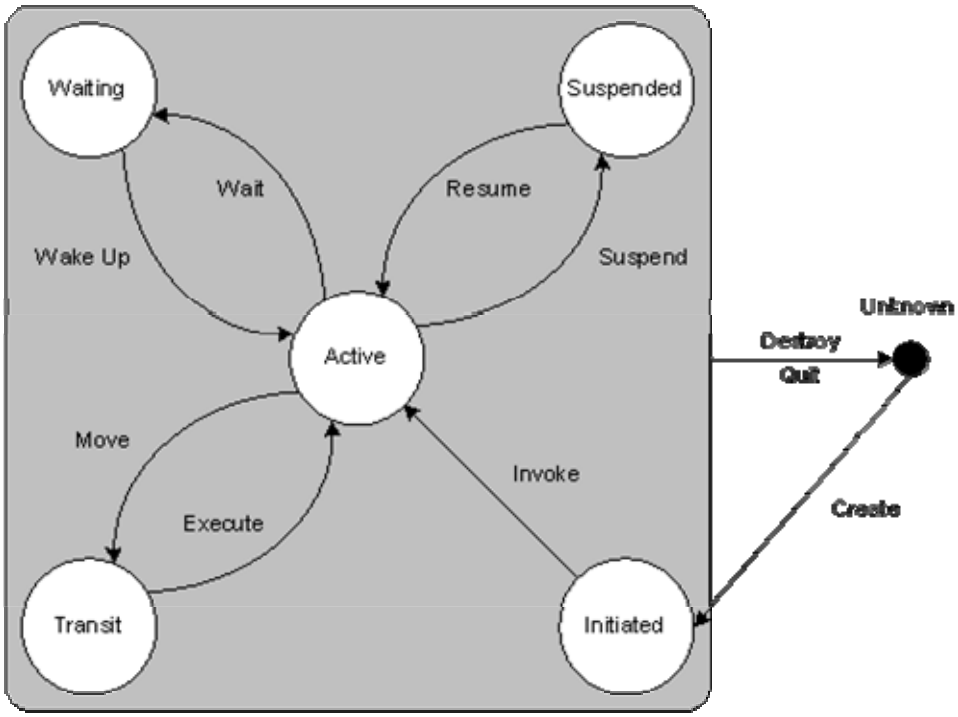
Bližší popis všetkých parametrov nájdete v [4].

JADE neurčuje, ako má vyzerať obsah správ. Je na výbere programátora, aký jazyk si zvolí. Agenti môžu rozumieť viacerým jazykom a nemusia rozumieť žiadnemu jazyku. Existuje viacero jazykov, ktoré sa zameriavajú na obsah správ (CCL - Constraint Choice Language [20], KIF - Knowledge Interchange Format [21], RDF - Resource Description Framework [22]). Ja sa budem zaoberať SL Content Language, ktorý už má svoju FIPA špecifikáciu (nájdeme ju v [5]).

JADE obsahuje viacero pomôcok, ktoré uľahčujú prácu s MAS. Veľmi významná pomôcka implementovaná v JADE je Remote Monitoring Agent, čo je agent, ktorý kontroluje agentné platformy (AP). Každý hlavný kontajner obsahuje okrem AMS a DF aj tohto agenta. Cez GUI (tj. okno) tohto agenta sa veľmi

jednoducho a prehľadne ovláda celá AP. Cez toto GUI môžeme okrem iného zatvárať kontajnery a platformy, taktiež môžeme spúšťať, zabíjať, presúvať agentov.

Programovanie agentov v JADE vychádza z triedy Agent. JADE agent je jednoducho užívateľom definovaná Java trieda, ktorá rozširuje triedu Agent. To znamená dedenie vlastností na splnenie základných interakcií s AP (ako je registrácia, konfigurácia, …) a základné metódy, ktoré môžu byť volané pri implementácii bežných behaviour agenta (ako posielanie/prijímanie správ, používanie protokolov …). Agent môže byť počas života v niekoľkých stavoch, ako je zobrazené na Obrázku 4.3. (obrázok som prevzal z [23])



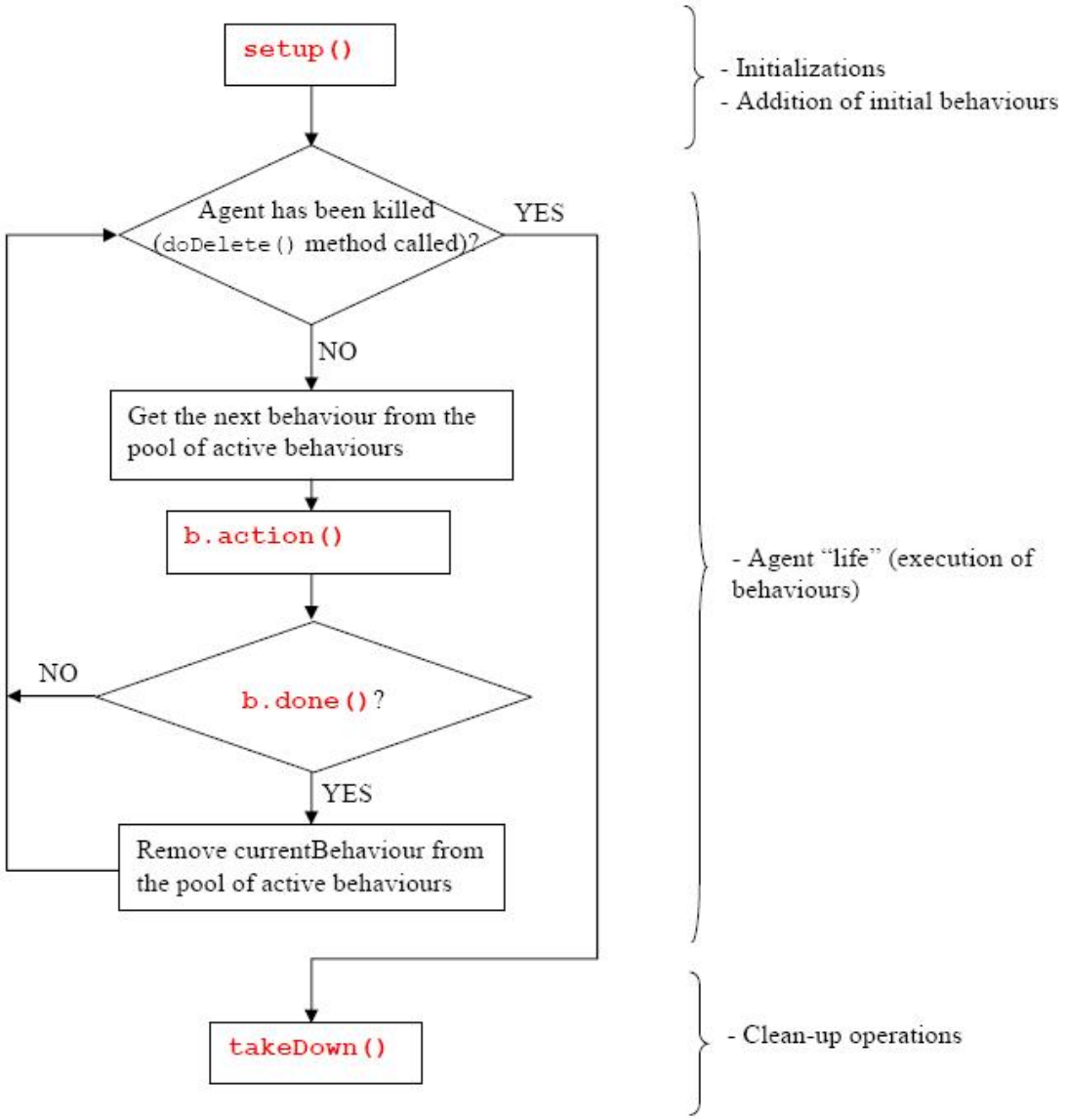
Obrázok 4.3

Stavy :

- INITIATED : Agent je vytvorený, ale zatiaľ ešte nie je registrovaný v AMS, nemá ani meno, ani adresy na komunikovanie
- ACTIVE : Agent je registrovaný v AMS, má meno aj adresu/y na komunikovanie. Iba v tomto stave môže agent vykonávať svoje behaviour
- SUSPENDED : Agent je zastavený. Žiadne jeho behaviour sa nevykonáva

- WAITING : Agent je zablokovaný, čaká na nejakú udalosť (zvyčajne na prijatie správy)
- TRANSIT : Mobilní agenti sú v tomto stave, keď migrujú na inú lokalitu

Každý JADE agent je jedno JAVA vlákno. Keďže JADE agent môže byť naprogramovaný v JAVA jazyku, môže agent pridávať nové JAVA vlákna. Avšak keď budeme uvažovať len o agentovom JAVA vlákne, pomôže nám Obrázok 4.4 (ktorý som našiel a prebral z [17]), na ktorom je znázornený priebeh vykonávania akcií tohto JAVA vlákna.



Obrázok 4.4

(na tomto obrázku je b označenie nejakého agentovho behaviour)

Ako je na Obrázku 4.4 znázornené, po vytvorení agenta sa začne vykonávať jeho metóda setup(). V tejto metóde si agent pridáva svoje behaviour. Ak agent nemá žiadne existujúce behaviour a prešla celá metóda setup(), ostáva agent "živý". Aby sme agenta ukončili, voláme metódu doDelete(), ktorá následne spustí metódu takeDown(). Metódu takeDown() môžeme preprogramovať, aby sme mohli korektne ukončiť agenta (vymazať data, ktoré agent potreboval ...).

5. Kapitola : Protokoly

Ako som spomínal v predchádzajúcej kapitole, JADE bolo vyvinuté pre telekomunikačnú spoločnosť a JADE agenti mali byť pôvodne mobilné telefóny. Aj z týchto dôvodov nemá JADE žiadnu funkcionalitu vyjednávania. Vyjednávanie je jednou z vlastností, ktoré by mal racionálny agent ovládať. Ja som sa rozhodol, že pridám túto dôležitú funkcionalitu do JADE a zároveň chcem ukázať, ako si každý užívateľ JADE, môže prispôsobovať multi-agentný systém podľa svojich požiadavkov. Zameriam sa hlavne na komunikáciu, špeciálne na pridávanie protokolov.

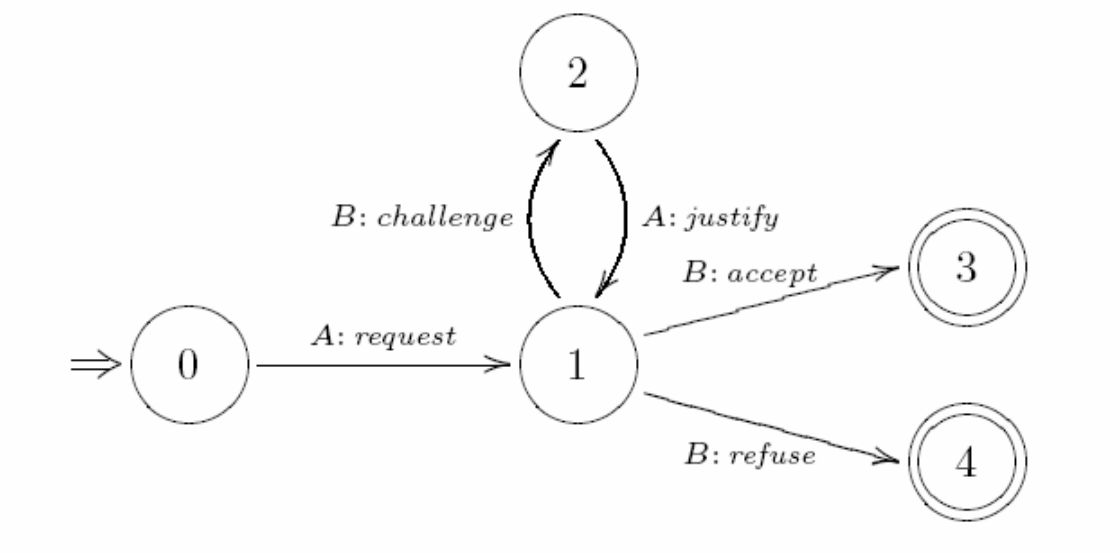
Konverzácia medzi agentami často prebieha podľa určitého vzoru. V týchto prípadoch v každom stave konverzácie očakávame prijatie, alebo poslanie určitých správ. Takýto vzor výmeny správ voláme protokol. V MAS sa väčšinou protokoly predšpecifikujú a agenti len jednoducho sledujú pravidlá daného protokolu. Protokoly sú verejné, teda sú známe všetkým zúčastneným agentom.

5.1. Reprezentácia protokolov

Protokoly môžeme reprezentovať viacerými spôsobmi. Napríklad logický prístup k reprezentácii protokolov môže byť ako množina if – then pravidiel, ktoré špecifikujú správne odpovede na jednotlivé prichádzajúce správy. Taktiež dobrý prístup k reprezentácii protokolov je pomocou konečného automatu, ktorý pozostáva z množiny stavov (vrátane vstupných a konečných stavov), zo vstupnej abecedy a z

prechodovej funkcie, ktorá zobrazuje dvojicu stav a element zo vstupnej abecedy na stav. V tomto prípade sú elementy vstupnej abecedy výkonné parametre správ.

Príklad vyjednávacieho protokolu je znázornený na Obrázku 5.1 (tento obrázok som našiel v [7]).



Obrázok 5.1

Na Obrázku 5.1 je reprezentácia vyjednávacieho protokolu medzi agentami A a B pomocou konečného automatu. Tenot protokol určuje, že po odoslaní žiadosti (request) agentom A, má agent B tri možnosti a to :

- prijať žiadosť (accept) a tým ukončí vyjednávací dialóg
- zamietnuť žiadosť (refuse) a tým ukončí vyjednávací dialóg
- začať vyjednávať (challenge)

Ak sa agent B rozhodne začať vyjednávať, agent A musí taktiež vyjednávať (pomocou justify) a tým vracia agenta B do stavu ako predtým (na Obrázku 5.1 je to stav 1).

Protokol z Obrázka 5.1 môžeme reprezentovať taktiež pomocou if – then pravidiel. Tieto pravidlá určujú správne odpovede na prichadzajúce správy. Napríklad, aby sme vyjadrili, že agent B môže odpovedať na žiadosť agenta A buď pomocou accept, alebo refuse, alebo challenge, použijeme pravidlo :

- $\text{tell(A, B, request, D, T)} \Rightarrow \text{tell(B, A, accept, D, T + 1)} \vee$
  $\text{tell(B, A, refuse, D, T + 1)} \vee$
  $\text{tell(B, A, challenge, D, T + 1)}$

kde tell(X, Y, Performative, D, T) je dialógový posun a znamená : X je agent (iniciátor), ktorý odoslal správu agentovi Y (Y je agent (respondér)), ktorý má odpovedať na prijatú správu od agenta X ($X \neq Y$), Performative je typ správy, D je identifikátor dialógu a T je čas odoslania správy (Pre jednoduchosť predpokladajme, že čas je diskrétna veličina, v každom časovom bode buď A, alebo B odošlú nejakú správu a T + 1 je nasledujúci časový bod po T). Poznamenajme ešte, že premenné, ktoré sú v if – then pravidle len v časti then (napravo od implikácie), sú existenčne kvantifikované a všetky ostatné sú univerzálne kvantifikované cez celé pravidlo.

Ďalej predpokladajme, že začiatok protokolu je spustený nejakou externou udalosťou START. Teda START(X, Y, D, T) je poslaný zo systému agentovi Y (iniciátor) ako dovolenie na začatie dialógu v čase T a identifikátorom dialógu D medzi agentami Y a X. Podobne na ukončenie dialógu použijeme STOP. STOP(X, Y, D, T) pošle agent X systému v čase T, aby ukončil dialóg s identifikátorom D, s agentom Y.

Aby sme mohli protokol z Obrázku 5.1 reprezentovať pomocou if – then pravidiel pre agenta, musíme o ňom uvažovať ako o dvoch subprotokoloch, jeden pre agenta iniciátora a jeden pre agenta respondéra. Takže protokol pre agenta iniciátora je :

$$P(i): \begin{cases} \text{START(X, Y, D, T)} & \Rightarrow \text{tell(Y, X, request, D, T + 1)} \\ \text{tell(X, Y, accept, D, T)} & \Rightarrow \text{STOP(Y, X, D, T + 1)} \\ \text{tell(X, Y, refuse, D, T)} & \Rightarrow \text{STOP(Y, X, D, T + 1)} \\ \text{tell(X, Y, challenge, D, T)} & \Rightarrow \text{tell(Y, X, justify, D, T + 1)} \end{cases}$$

Všimnime si, že agent iniciátor (Y) nemá možnosť výberu (na prijatú správu má určenú odpoveď) a taktiež tento agent posiela signál STOP.

Protokol pre agenta respondéra je :

$$P(r): \begin{cases} \text{tell(X, Y, request, D, T)} & \Rightarrow \text{tell(Y, X, accept, D, T}+1) \vee \\ & \quad \text{tell(Y, X, refuse, D, T}+1) \vee \\ & \quad \text{tell(Y, X, challenge, D, T}+1) \\ \text{tell(X, Y, justify, D, T)} & \Rightarrow \text{tell(Y, X, accept, D, T}+1) \vee \\ & \quad \text{tell(Y, X, refuse, D, T}+1) \vee \\ & \quad \text{tell(Y, X, challenge, D, T}+1) \end{cases}$$

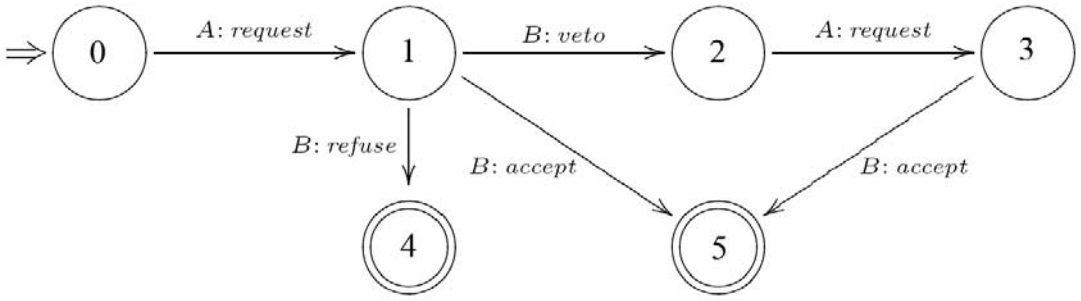
Keď máme nejaký protokol reprezentovaný pomocou if – then pravidiel (teda logická reprezentácia) a na ľavej strane pravidiel je len jeden dialógový posun tell, môžeme pomerne jednoducho skontrolovať, či je protokol dobre skonštruovaný, alebo nie. Protokol musí spĺňať nasledujúce požiadavky (budem ich označovať ako požiadavky dobrej konštrukcie):

- musí existovať aspoň jedno pravidlo protokolu, ktoré má START na ľavej strane (myslím naľavo od implikácie). START nikdy nie na pravej strane (myslím napravo od implikácie) pravidla protokolu
- pre každý dialógový posun (v našom príklade to bol tell(...)), ktorý je na pravej strane pravidla protokolu, existuje pravidlo protokolu, ktoré ho má na ľavej strane. Toto neplatí len pre STOP
- platí : tell(X, Y, S1, T, D) $\wedge$ tell(X, Y, S2, T, D) $\wedge$ S1 $\neq$ S2 $\Rightarrow$ false
- pre každé pravidlo protokolu platí, ak X je iniciátor a Y respondér na ľavej strane pravidla, potom v každom dialógovom posune na pravej strane tohto pravidla bude X respondér a Y iniciátor
- Pre protokol iniciátora platí : dialógové posuny na ľavých stranách všetkých jeho protokolových pravidiel sú navzájom rôzne. To isté platí aj pre protokol respondéra

Na Obrázku 5.1 je príklad protokolu, pri ktorom sa dá jednoznačne určiť, aká správa má nasledovať po prijatí nejakej správy. Avšak takéto nie sú všetky protokoly. Niektoré protokoly určujú nasledujúcu správu podľa n predošlých správ. Protokol z Obrázku 5.1, je špeciálny prípad, kedy n = 1. Ale napríklad na Obrázku 5.2 je protokol,

pri ktorom, aby sme mohli určiť stav 3, potrebujeme zistiť, aké boli až tri posledné správy. Teda v protokole respondéra (na obrázku agent B) bude pravidlo :

tell(A, B, request, D, T - 2) &
tell(B, A, veto, D, T - 1) &
tell(A, B, request, D, T) $\Rightarrow$ tell(B, A, accept, D, T + 1)



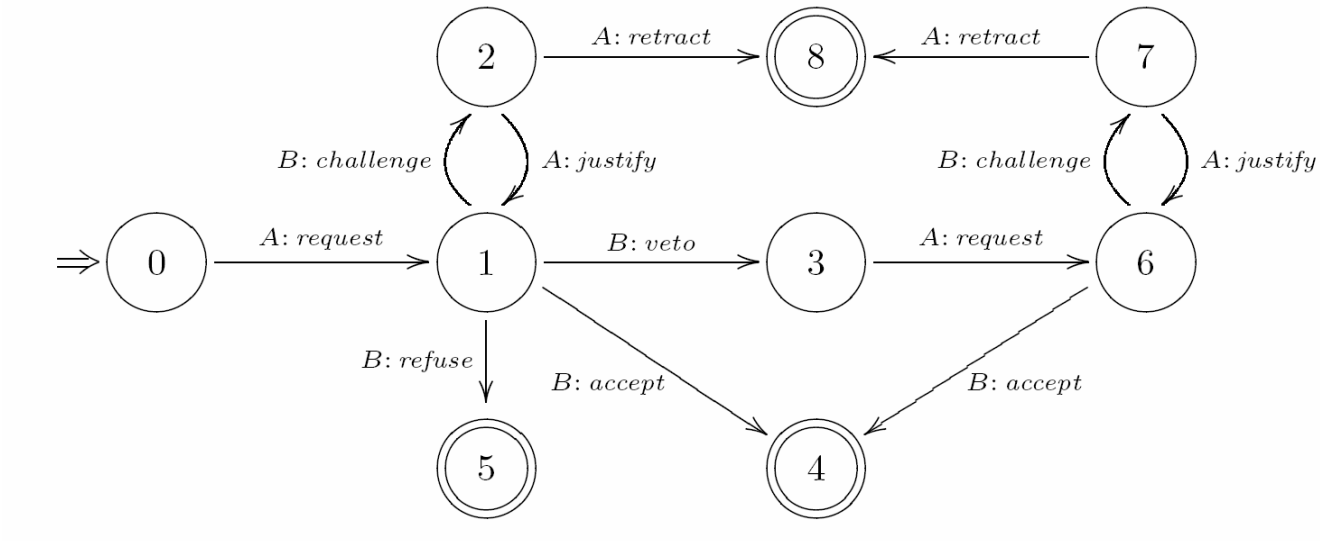
Obrázok 5.2

Už pri týchto typoch protokolov (kedy n > 1) je ťažké určiť požiadavky dobrej konštrukcie. Navyše existujú zložitejšie protokoly, pri ktorých sa nedá určiť n. Na Obrázku 5.3 (protokol a aj obrázok som našiel v [7]) je takýto protokol. V prípade tohto protokolu by sme mali pravidlá respondérovho protokolu pre stav 6 nasledujúce :

- tell(A, B, request, D, T) & tell(B, A, veto, D, T´) & (T´ < T)
  $\Rightarrow$ tell(B, A, accept, D, T + 1) $\vee$ tell(B, A, challenge, D, T + 1)

Avšak napísanie podobného pravidla pre stav 1 je problematické, pretože musíme vyjadriť, že veto nebolo poslané niekedy v minulosti. Jednou z možností, ako sa dá tento problém riešiť, je prijať reprezentáciu stavov do logiky. Môžeme priamo zakódovať konečný automat do dvoch druhov pravidiel :

1. state(N,T) & tell(..., T) $\Rightarrow$ state(N´, T + 1) , toto pravidlo slúži na posun medzi stavmi
2. state(N, T) $\Rightarrow$ tell(..., T + 1) $\vee$ ... $\vee$ tell(..., T + 1), kde tell(..., T + 1) sú možné dialógové posuny zo stavu state(N, T)



Obrázok 5.3

Pre každú konverzáciu medzi dvoma agentami JADE taktiež rozlišuje Iniciátora a Respondéra. Iniciátor začína konverzáciu a respondér začne konverzovať až potom, ako je kontaktovaný iným agentom. JADE poskytuje niekoľko hotových behaviour tried pre iniciátora aj pre respondéra. Tieto triedy odpovedajú niektorým protokolom, ktoré sú definované vo FIPA. Napríklad pre subscription protokol JADE ponúka triedy SubscriptionInitiator a SubscriptionResponder. Všetky iniciátorové behaviour sú ukončené a už sa viac nerozvrhujú hneď, ako dosiahnu konečný stav. Všetky respondérové behaviour sú zasa cyklické a znovu rozvrhnuté po dosiahnutí konečného stavu. Keď chce agent začať komunikovať s iným agentom pomocou nejakého protokolu, jednoducho si pridá behaviour s triedou tohto protokolu a tento behaviour sa rozvrhne tak, ako ostatné jeho behaviour.

Ja som sa zaoberal protokolom, ktorý je znázornený na Obrázku 5.3. Je to protokol, ktorý predstavuje dvojfázové vyjednávanie. Tento protokol je určený pre vyjednávanie práve pre dvoch agentov, označím ich A a B. Začiatok je v stave 0 a koncové stavy sú 4, 5 a 8. Najskôr agent A pošle žiadosť a sme v stave 1. V tomto stave agent B môže vyzvať agenta A k vyjednávaniu, môže zamietnuť alebo súhlasiť

so žiadosťou, alebo môže využiť právo veta. Ak B vyzve agenta A k vyjednávaniu, budeme vo vyjednávacom cykle, pokým buď A nestiahne žiadosť, alebo B neprestane vyzývať. Ak B v stave 1 vetuje žiadosť, A mu pošle novú žiadosť. V stave 6 je to veľmi podobné ako v stave 1, akurát agent B už nemôže vetovať a ani zamietnuť žiadosť. Takže je to trochu risk pre B vetovať žiadosť od A, lebo Agent A pošle novú žiadosť a tú už B nemôže zamietnuť.

Tento protokol som si vybral pretože to je zložitý protokol a mojím cieľom je aj ukázať, ako sa dajú do JADE pridávať nové protokoly. Preto je lepšie ukázať to na zložitejšom príklade, ktorý má v sebe rôzne zložité stavy a potom, keď si bude niekto chcieť pridať do JADE svoj vlastný protokol, bude mu môj príklad oveľa užitočnejší. Zložité sú hlavne stavy 1 a 6 (z Obrázka 5.3). Po obdržaní buď priamo žiadostí alebo žiadostí z vyjednávania (tj. justify) je dôležité určiť, či sme v stave 1, alebo sme v stave 6. Taktiež tento protokol obsahuje cyklus.

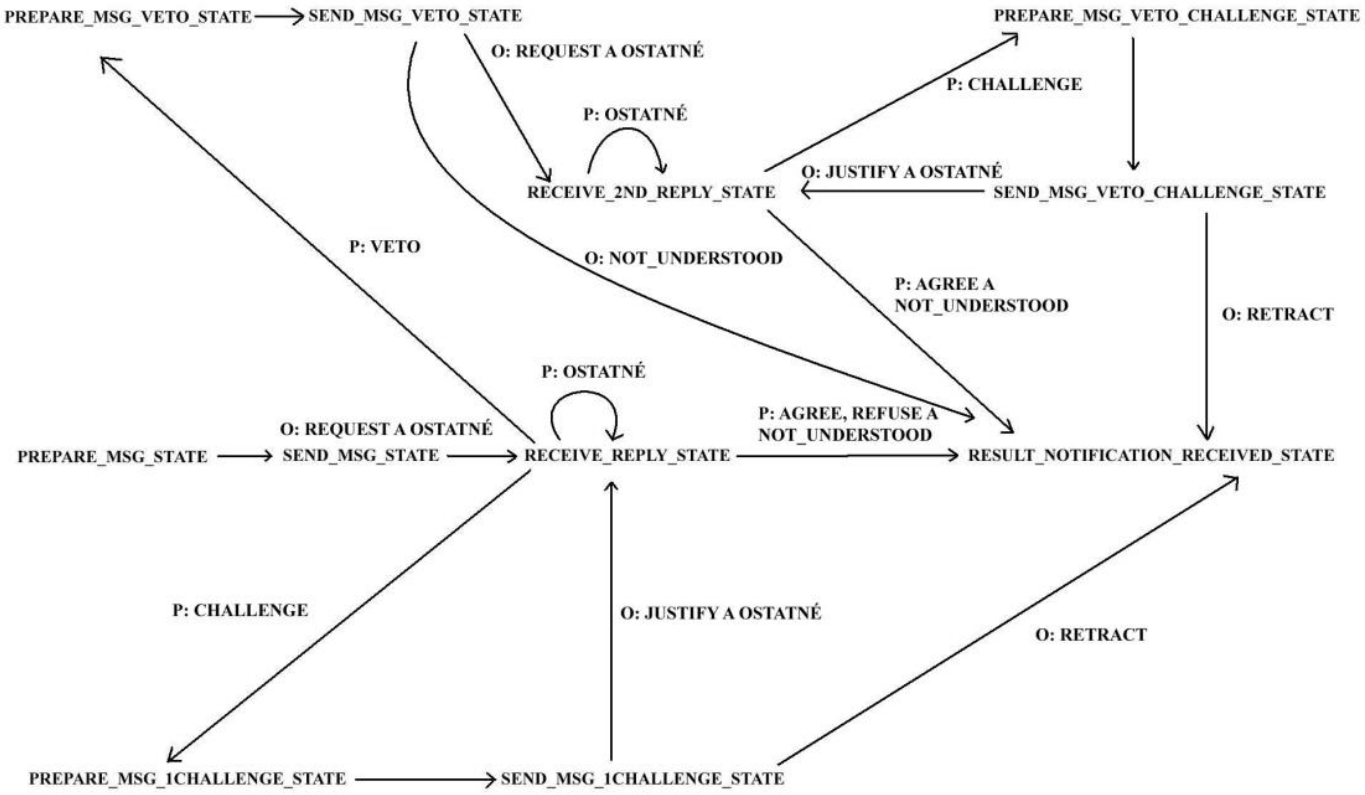
Keby sme chceli tento protokol naprogramovať tak, aby vyhovoval FIPA, nedalo by sa to. FIPA totiž nemá ako výkonný parameter správ Justify, Challenge, Veto, Retract (FIPA nemá ani ACCEPT, ale obsahuje AGREE, čo má rovnaký význam. Odteraz budem používať AGREE namiesto ACCEPT). Keďže JADE je naprogramované tak, aby vyhovovalo FIPA štandardom, ani JADE nepozná názvy týchto parametrov. Tak som ich do JADE doplnil. Taktiež som naprogramoval tento protokol, tak aby ho mohli používať agenti vytvorení v JADE. Nazval som tento protokol Veto, a doplnil som ho medzi protokoly. Triedy, ktoré som vytvoril, sa volajú VetoInitiator a VetoResponder.

5.2 Implementácia Veto protokolu

Veto protokol sú dve triedy, ktoré fungujú ako behaviour. Jeden je pre iniciátora, a to VetoInitiator, a jeden je pre respondéra, a to VetoResponder. Tieto triedy majú určené metódy, ktoré musí agent preprogramovať, aby konverzácia fungovala tak, ako chce agent. Napríklad iniciátor musí preprogramovať metódu prepareChallengeResponse, čo je metóda, ktorá je volaná v stave 2 (podľa obrázku 5.3), kde iniciátor by mal na základe prijatej výzvy určiť, či stiahne svoju žiadosť, alebo nie, a bude žiadať znovu.

V JADE sa pridané behaviour rozvrhujú cyklicky bez priority. Keď príde na rad nejaké behaviour, spustí sa jeho action() metóda a táto prejde až do konca. Keď táto metóda skončí, volá sa metóda done(), ktorá kontroluje, či už agent dosiahol svoj cieľ. Ak áno, behaviour je ukončené a odstránené z radu čakajúcich behaviour, a ak nie, je uložené na koniec tohto radu (pre ilustráciu pozri Obrázok 4.4). Aby sme sa vyhli aktívnemu čakaniu na správu (čo zastaví vykonávanie ostatných agentových behaviour), môžeme zablokovať behaviour pomocou metódy block(). Táto metóda umiestni behaviour do radu blokovaných behaviour hneď po skončení metódy action(). Všetky blokované behaviour sú znova rozvrhnuté, keď je doručená nová správa. Keďže predpokladáme, že agenti budú chcieť vykonávať viaceré behaviour naraz, je dobré, aby žiadne behaviour nezaberalo príliš veľa času svojou zložitosťou, a tým neznemožňovalo vykonávanie ostatných agentových behaviour. Preto by sa mali zložité a výpočtovo náročné behaviour rozdeliť do viacerých krokov. Na tento fakt som bral ohľad aj pri programovaní môjho Veto protokolu. Implementoval som Veto protokol pomocou stavov (podobne ako to je v prípade konečného automatu) tak, aby sa pri každom spustení behaviour vykonávali len operácie určené pre príslušný stav, v ktorom sa práve nachádzame, a posunutie sa do ďalšieho stavu. Potom metóda action() skončí a behaviour sa zaradí na koniec radu čakajúcich behaviour (ak behaviour neskončilo, tj. done() vracia false). Keď príde toto behaviour zasa na rad, vykonávajú sa znova len operácie určené pre aktuálny stav a posunutie sa do ďalšieho stavu. A takto to pôjde ďalej, až kým behaviour neskončí, teda done() vracia true.

VetoInitiator sa pohybuje po stavoch tak, ako je zobrazené na Obrázku 5.4.



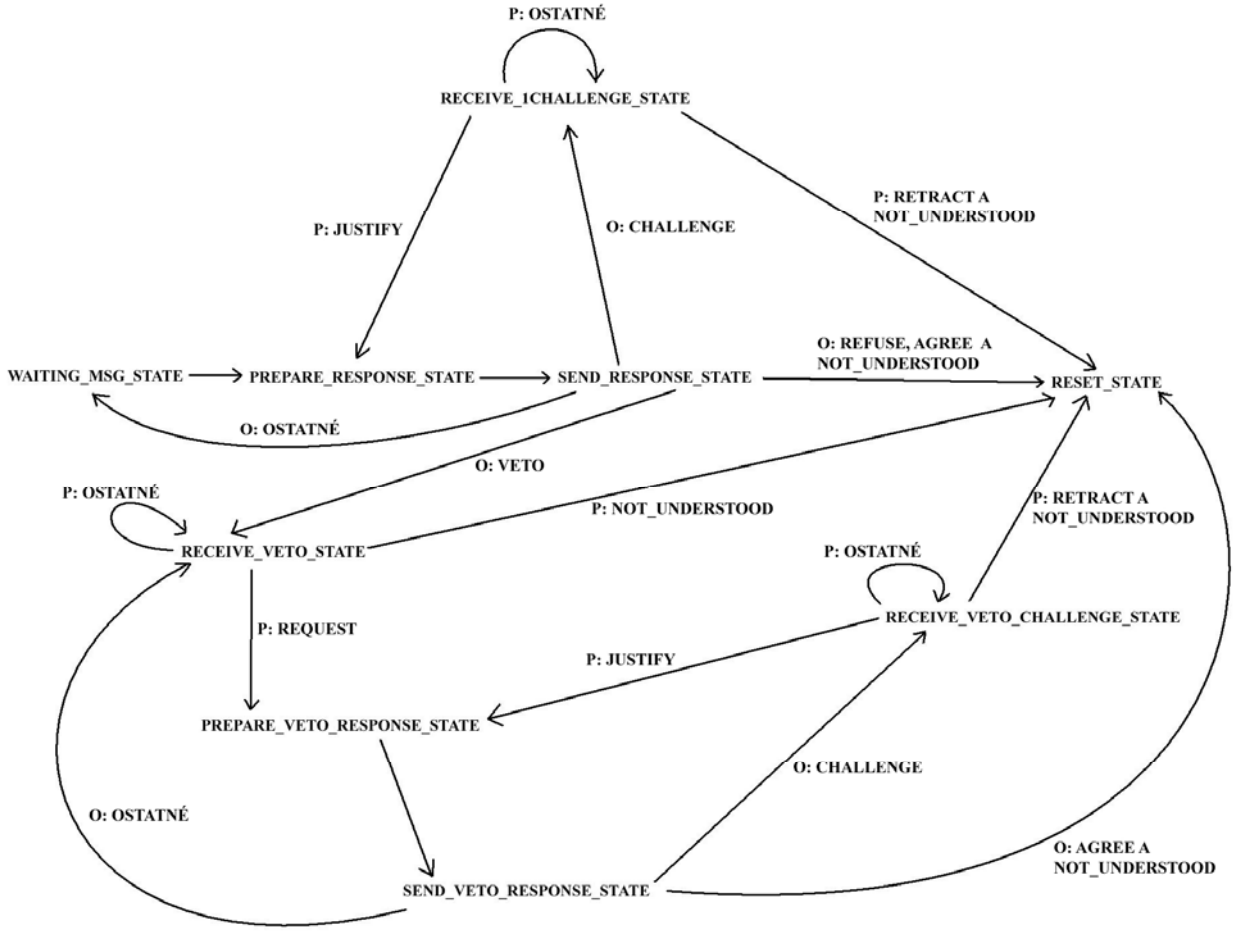
Obrázok 5.4

(na obrázku skratka O: znamená odoslaná správa a skratka P: znamená prijatá správa)

Iniciátor začína v stave PREPARE_MSG_STATE, v ktorom si pripraví správu, ktorú bude posielať ako prvú. Táto správa je daná ako argument pri pridávaní iniciátorovho behaviour. Iniciátor má jeden koncový stav a to RESULT_NOTIFICATION_RECEIVED_STATE, do ktorého sa dostane, ak príjme Agree alebo Refuse, alebo ak odošle Retract. V koncových stavoch sa behaviour považuje za splnené, ukončí sa a vymaže z radu čakajúcich behaviour. V každom stave, ktorý začína na RECEIVE, sa agent zablokuje a čaká na príchod správy (tým nezdržuje ostatné behaviour agenta, zaradí sa do radu blokujúcich behaviour). V každom stave, ktorý sa začína na PREPARE, by si mal agent pripraviť správu, ktorú chce odoslať. Táto pripravená správa sa následne posiela, a tomu zodpovedajú stavy, ktoré začínajú SEND. Keďže agentové behaviour sa vykonáva jedno po druhom (ak

behaviour nedávam do vlastných JAVA vlákien, to však treba explicitne naprogramovať), nie je vhodné, aby jedno spustenie nejakého behaviour trvalo dlho. Preto som implementoval stavy PREPARE a SEND. Keby som ich zlúčil, bolo by jedno spustenie behaviour dlhšie. Určite sa rozdelením týchto stavov nič nezhoršilo, teda môže sa len zlepšiť výkon agenta. Vždy, keď sa spúšťa VetoInitiator behaviour (tj. je prvý v rade čakajúcich behaviour), vykonajú sa akcie jedného stavu a určí sa stav, ktorý sa má vykonávať pri ďalšom spustení.

VetoResponder sa pohybuje po stavoch tak, ako je to znázornené na Obrázku 5.5.



Obrázok 5.5

(na obrázku skratka O: znamená odoslaná správa a skratka P: znamená prijatá správa)

Respondér má začiatočný stav WAITING_MSG_STATE. V tomto stave je zablokovaný a čaká na príchod prvej správy. Taktiež ako v prípade iniciátora aj respondér sa zablokuje v každom stave, ktorý čaká na príchod správy, čiže v stave začínajúcom RECEIVE. Respondér má iba jeden koncový stav, a to RESET_STATE. Stavy začínajúce PREPARE sú podobné ako u iniciátora, takisto aj stavy začínajúce SEND.

V stave RESET_STATE sa narozdiel od iniciátora nepovažuje behaviour za splnené, ale sa len celé behaviour resetuje. Je to preto, aby respondér po dokončení dialógu s jedným agentom mohol komunikovať s týmto agentom aj naďalej (keď budú chcieť odznova vyjednávať podľa VETO protokolu). Toto je typický prístup podľa JADE definície respondérovho behaviour. Avšak môže nastať prípad, kedy agent respondér komunikuje s veľmi veľkým počtom agentov. Po tých, s ktorými už dokomunikoval, ostanú agentovi blokované behaviour. Niekedy týchto behaviour môže byť až príliš veľa. Dá sa to riešiť tak, že ak mám príliš veľa behaviour, môžem tieto behaviour odstrániť pomocou metódy : void jade.core.Agent.removeBehaviour(Behaviour b). Ja však ponúkam ešte inú možnosť. Preprogramovanie metódy done() v triede VetoResponder. Pripomeniem, že metóda done() sa kontroluje na konci prechádzania behaviour, a ak vracia true, behaviour sa ukončí a vymaže, a ak vracia false, behaviour sa zaradí do radu čakajúcich behaviour, a keď príde na rad, znova sa spustí. Pôvodne je metóda done() nastavená tak, aby vracala false, v tomto prípade bude behaviour fungovať tak, že po dosiahnutí RESET_STATE sa nastaví ďalší stav WAITING_MSG_STATE a pri nasledujúcom spustení behaviour sa toto behaviour zablokuje. Agent teda čaká, kým sa mu zasa ozve ten istý agent iniciátor, ktorý bude chcieť komunikovať pomocou VETO protokolu. Tento spôsob má výhodu vtedy, ak agent respondér často komunikuje s nejakou nie príliš veľkou skupinou agentov podľa VETO protokolu. Avšak keď preprogramujem metódu done() tak, aby vracala true, po dosiahnutí RESET_STATE sa behaviour ukončí a vymaže. Tento prístup je vhodný, ak agent respondér komunikuje s ostatnými agentami podľa VETO protokolu len málokedy.

Robustnosť :

VETO protokol pre iniciátora a aj pre respondéra som naprogramoval tak, aby bol robustný voči nepredvídaným situáciam.

Medzi nepredvídané situácie patrí napríklad situácia, keď jeden z agentov pošle správu, ktorá je mimo protokolu. Keď toto nastane, tak sa spustí metóda : void handleOutOfSequence(ACLMessage msg), ktorú môže programátor agenta preprogramovať. Táto metóda potom vlastne bude agentova reakcia na správu mimo protokolu. Avšak, ak agent príjme správu mimo protokolu, neznamená to, že celý dialóg končí. Agent stále očakáva príchod regulárnej správy, teda ostáva vo svojom stave.

Taktiež, agent môže prijať správu, ktorá má správny výkonný parameter (to znamená, že je regulárna voči protokolu), ale obsah tejto správy môže byť nezrozumiteľný. Nezrozuminiteľnosť správy môže vyplývať z viacerých príčin. Jednou z nich je, že agent, ktorý túto správu prijal, nemusí poznať jazyk, v ktorom je obsah správy definovaný. Taktiež môže nastať situácia, že obaja komunikujúci agenti poznajú jazyk, ktorým je písaný obsah správy, avšak tento obsah nedokáže prijímajúci agent spracovať. V prípade, že agent nerozumel správe, môže dať o tom vedieť tak, že vráti správu typu (s výkonným parametrom) NOT_UNDERSTOOD. Keď agent príjme správu typu NOT_UNDERSTOOD, spustí sa metóda (ktorú by mal agent preprogramovať) void handleNotUnderstood(ACLMessage msg) ako reakcia agenta na takýto typ správy. Príjem správy typu NOT_UNDERSTOOD sa považuje za závažný problém. Preto sa po prijatí, alebo odoslaní takejto správy dialóg ukončí (protokol prejde do koncového stavu).

Ďalšou situáciou, ktorá môže nastať je, keď sa správa stratí, alebo sa agent rozhodne neposlať žiadnu správu. V tom prípade by jeden z agentov mohol čakať na správu veľmi dlho. Preto je zvykom, že keď agent odosiela správu, na ktorú očakáva odpoveď, vyplní nepovinný parameter správy ReplyByDate. Tento parameter správy určuje prijímateľovi správy, dokedy má prísť odpoveď na túto správu. Znamená to zároveň, dokedy bude agent čakať na odpoveď. S týmto som počítal aj pri implementácii VETO protokolu. Po odoslaní správy, ktorá ma nastavený parameter ReplyByDate (tento čas musí byť v budúcnosti od času odoslania), si protokol nastaví timeout, do ktorého musí prísť odpoveď. Ak do tohto času odpoveď nepríde, dialóg sa

končí (agent, ktorému vypršal timeout prejde do koncového stavu). Ak agentovi vyprší timeout, spustí sa metóda void handleTimeoutExpired(int state_nmr), ktorú by mal programátor agenta preprogramovať na žiadanú reakciu agenta na takúto udalosť. Zároveň by bolo dobré, keby sa v tejto metóde poslala informačná správa druhému agentovi, aby sa dozvedel o ukončení dialógu. Môže sa stať, že agent nevyplní parameter správy ReplyByDate, aj keď na túto správu čaká odpoveď. Ak agent nevyplní tento parameter, správa sa VETO protokol tak, že agent chce čakať na odpoveď neobmedzene dlho. V tomto prípade sa agentovo behaviour zablokuje až do príchodu tejto odpovede. Je veľmi užitočné, určiť si nejaký maximálny čas, ktorý je agent ochotný čakať na správu, a tento pridávať do parametra (ReplyByDate) každej odoslanej správy. Tým sa môžeme vyhnúť hromadeniu blokovaných agentových behaviour.

Každá konverzácia medzi dvoma agentami môže mať nejaké (unikátne) označenie. Pre tento účel má každá ACL správa parameter ConversationId. Tento parameter nie je povinný. Avšak, ak tento parameter nie je vyplnený, tak môj VETO protokol bude tento parameter vždy vypĺnať nejakým univerzálnym reťazcom (typu String). Programátor agentov sa nemusí starať o tento parameter, slúži len pre lepšie ovládanie protokolu. Napríklad môže nastať situácia, keď dvaja agenti budú chcieť viesť viac dialógov pomocou VETO protokolu zároveň. V tomto prípade by sme bez vyplneného parametra ConversationId, len veľmi ťažko určili, ktorá správa patrí ktorému dialógu.

6. Kapitola : Používanie protokolov

Keď chcú dvaja agenti medzi sebou komunikovať podľa nejakého protokolu, musia obaja poznať tento protokol. Taktiež musia vedieť, aké metódy musia preprogramovať, aby mohli správne reagovať na prijaté správy, a aby mohli posielať chcené správy. Aj v protokole Veto sú takéto metódy.

6.1 Používanie protokolu pre iniciátora

Keď iniciátor chce komunikovať s iným agentom (respondérom) pomocou Veto protokolu, použije triedu VetoInitiator, ktorá má 2 argumenty, a to:

1. triedu agenta, tým sa myslí agent, ktorý si pridáva VetoInitiator ako svoje behaviour. V praxi (ak programujeme v JAVA jazyku) sa za tento argument dosadí “this“.
2. správu, ktorá má byť poslaná ako prvá.

Agent iniciátor by si mal pridať medzi svoje behaviour takú triedu VetoInitiator, ktorá by mala preprogramované nasledujúce metódy:

Metódy, ktoré pripravujú správy na odoslanie:

- **ACLMessage prepareRequest(ACLMessage msg)**: táto metóda je volaná v stave 0 (podľa Obrázka 5.3). Ako argument dostáva správu, ktorá bola argumentom pri pridávaní VetoInitiator behaviour. Táto metóda vracia prvú žiadosť (správa typu REQUEST), ktorú chce iniciátor poslať respondérovi. Ak nie je táto metóda preprogramovaná, vráti správu, ktorá je argumentom tejto metódy. Vrátená správa by mala byť typu REQUEST.
- **ACLMessage prepareChallengeResponse(ACLMessage challenge)**: táto metóda je volaná v stave 2 (podľa Obrázka 5.3). Ako argument dostáva správu CHALLENGE, ktorú naposledy poslal iniciátorovi respondér. Táto metóda by mala vrátiť správu typu JUSTIFY alebo RETRACT, prípadne môže vyhodiť Not_Understood výnimku (to vtedy, keď iniciátor nerozumie správe typu CHALLENGE od respondéra, ktorá je aj parametrom tejto metódy). V takom prípade bude iniciátor posielať správu typu NOT_UNDERSTOOD.
- **ACLMessage prepareVetoResponse(ACLMessage veto)**: táto metóda je volaná v stave 3 (podľa Obrázka 5.3). Argumentom je správa typu VETO od respondéra. Táto metóda vracia správu typu Request. Prípadne môže vyhodiť Not_Understood výnimku a bude sa posielať správa typu NOT_UNDERSTOOD.
- **ACLMessage prepareVetoChallengeResponse(ACLMessage challenge)**: táto metóda je volaná v stave 7 (podľa Obrázka 5.3). Argumentom je správa typu CHALLENGE, ktorú naposledy poslal iniciátorovi respondér. Táto metóda by mala vrátiť správu typu JUSTIFY alebo RETRACT, prípadne môže

vyhodiť Not_Understood výnimku a bude sa posielať správa typu NOT_UNDERSTOOD.

V týchto metódach (začínajúce na prepare) programátor ovplyvňuje správy, ktoré bude agent iniciátor odosielať agentovi respondérovi. Uvažujme nasledujúci prípad : Máme správu, ktorá je argumentom zmeneného VetoInitiator pri pridávaní tohto agentovho behaviour. Ak táto správa je správa, ktorú bude agent iniciátor posielať ako prvú bez toho, aby bolo treba na nej hocičo meniť, alebo pridávať, nie je nutné preprogramovať metódu prepareRequest. Pretože nepreprogramovaná metóda prepareRequest vracia práve správu, ktorú dostáva ako argument pri svojom volaní. Táto správa je zároveň aj správou, ktorá je argumentom pri pridávaní VetoInitiator behaviour (teda keď nezmením metódu prepareRequest, bude sa ako prvá správa posielať správa, ktorá bola argumentom pri pridávaní VetoInitiator behaviour). Avšak je vhodné, ak sa metóda prepareRequest preprogramuje aspoň tak, aby kontrolovala správnosť vstupnej správy.

Ostatné metódy spomenuté vyššie (začínajúce na prepare) je nutné preprogramovať, pretože inak vracajú null a teda agent neposiela žiadnu správu.

Ďalej môže agent preprogramovať metódy, ktoré su volané pri prijatí správy. Sú to:

- **void handleAgree(ACLMessage agree)**: táto metóda je volaná práve vtedy, keď nám respondér pošle zo stavu 1 (podľa Obrázka 5.3) správu typu AGREE. Argumentom je práve táto prijatá správa.
- **void handleVetoAgree(ACLMessage agree)**: táto metóda je volaná práve vtedy, keď nám respondér pošle zo stavu 6 (podľa Obrázka 5.3) správu typu AGREE. Argumentom je práve táto prijatá správa.
- **void handle1Challenge(ACLMessage challenge)**: táto metóda je volaná práve vtedy, keď nám respondér pošle zo stavu 1 (podľa Obrázka 5.3) správu typu CHALLENGE. Argumentom je práve táto prijatá správa.
- **void handleVetoChallenge(ACLMessage challenge)**: táto metóda je volaná práve vtedy, keď nám respondér pošle zo stavu 6 (podľa Obrázka 5.3) správu typu CHALLENGE. Argumentom je práve táto prijatá správa.

- **void handleVeto(ACLMessage veto)**: táto metóda je volaná práve vtedy, keď nám respondér pošle zo stavu 1 (podľa Obrázka 5.3) správu typu VETO. Argumentom je práve táto prijatá správa.
- **void handleRefuse(ACLMessage refuse)**: táto metóda je volaná práve vtedy, keď nám respondér pošle zo stavu 1 (podľa Obrázka 5.3) správu typu REFUSE. Argumentom je práve táto prijatá správa.
- **void handleNotUnderstood(ACLMessage not_understood)**: táto metóda je volaná vždy, keď iniciátor príjme od respondéra správu typu NOT_UNDERSTOOD. Argumentom je práve táto doručená správa.
- **void handleOutOfSequence(ACLMessage msg)**: táto metóda je volaná vždy, keď iniciátor príjme správu od respondéra, ktorá je mimo poradia, vzhľadom k protokolovým pravidlám. Argumentom je práve táto prijatá správa.
- **void handleTimeoutExpired(int state_nmr):** táto metóda je volaná vždy, keď agent prestane čakať na odpoveď na svoju správu, ktorej predtým nastavil parameter ReplyByDate. Agent prestane čakať, pretože vypršal časový limit, v ktorom mala prísť odpoveď. Ako argument má táto metóda číslo stavu, v ktorom je volaná (slúži hlavne pre ladiacu fázu).

Tieto metódy (začínajúce na handle) slúžia hlavne ako agentove reakcie na rôzne udalosti (prijatie rôznych typov správ, vypršanie časového limitu, v ktorom mala prísť odpoveď...). Keď sa niektorá z týchto metód nepreprogramuje, protokolu (dialógu podľa protokolu) to nevadí. Dokonca sa nemusí preprogramovať ani jedna z týchto metód (začínajúcich na handle).

6.2 Používanie protokolu pre respondéra

Aj respondér môže komunikovať s ostatnými agentami podľa Veto protokolu. Pre tieto účely sa používa metóda VetoResponder, ktorá má 2 argumenty:

1. trieda agenta, tým sa myslí agent, ktorý si pridáva VetoResponder ako svoje behaviour.
2. správa, ktorá bola prijatá ako prvá, táto správa zároveň určuje agentovi respondérovi, že agent iniciátor chce s ním komunikovať pomocou VETO protokolu

Agent respondér by si mal pridať medzi svoje behaviour takú triedu VetoResponder, ktorá by mala preprogramované nasledujúce metódy:

Podobne ako u agenta iniciátora používame metódy, ktoré pripravujú správy na odoslanie:

- **ACLMessage prepareResponse(ACLMessage msg)**: táto metóda je volaná v stave 1 (podľa Obrázka 5.3). Ako argument dostáva buď prvú správu typu REQUEST, ktorú posiela iniciátor zo stavu 0 (podľa Obrázka 5.3), alebo správu typu JUSTIFY, ktorú posiela iniciátor zo stavu 2 (podľa Obrázka 5.3). Táto metóda vracia správu, ktorá je typu REFUSE, AGREE, VETO, JUSTIFY, alebo môže vyhodiť Not_Understood výnimku.
- **ACLMessage prepareVetoResponse(ACLMessage request)**: táto metóda je volaná v stave 6 (podľa Obrázka 5.3). Ako argument dostáva buď správu typu REQUEST, ktorú posiela iniciátor zo stavu 3 (podľa Obrázka 5.3), alebo správu typu JUSTIFY, ktorú posiela iniciátor zo stavu 7 (podľa Obrázka 5.3). Táto metóda vracia správu, ktorá je typu AGREE, JUSTIFY, alebo môže vyhodiť Not_Understood výnimku.

Ďalej môže agent preprogramovať metódy, ktoré su volané pri prijatí správy:

- **void handleRequest(ACLMessage request)**: táto metóda je volaná práve vtedy, keď nám iniciátor poslal prvú správu zo stavu 0 (podľa Obrázka 5.3) typu REQUEST. Argumentom je práve táto prijatá správa.
- **void handleRetract(ACLMessage retract)**: táto metóda je volaná práve vtedy, keď nám iniciátor pošle zo stavu 2 (podľa Obrázka 5.3) správu typu RETRACT. Argumentom je práve táto prijatá správa.
- **void handleJustify(ACLMessage justify)**: táto metóda je volaná práve vtedy, keď nám iniciátor pošle zo stavu 2 (podľa Obrázka 5.3) správu typu JUSTIFY. Argumentom je práve táto prijatá správa.
- **void handle2ndRequest(ACLMessage request)**: táto metóda je volaná práve vtedy, keď nám iniciátor pošle zo stavu 3 (podľa Obrázka 5.3) správu typu REQUEST. Argumentom je práve táto prijatá správa.

- **void handleVetoRetract(ACLMessage retract)**: táto metóda je volaná práve vtedy, keď nám iniciátor pošle zo stavu 7 (podľa Obrázka 5.3) správu typu RETRACT. Argumentom je práve táto prijatá správa.
- **void handleVetoJustify(ACLMessage justify)**: táto metóda je volaná práve vtedy, keď nám iniciátor pošle zo stavu 7 (podľa Obrázka 5.3) správu typu JUSTIFY. Argumentom je práve táto prijatá správa.
- **void handleNotUnderstood(ACLMessage not_understood)**: táto metóda je volaná vždy, keď respondér príjme od iniciátora správu typu NOT_UNDERSTOOD. Argumentom je práve táto prijatá správa.
- **void handleOutOfSequence(ACLMessage msg)**: táto metóda je volaná vždy, keď respondér príjme od iniciátora správu, ktorá je mimo poradia vzhľadom k protokolovým pravidlám. Argumentom je práve táto prijatá správa.
- **void handleTimeoutExpired(int state_nmr):** táto metóda je volaná vždy, keď agent prestane čakať na odpoveď na svoju správu, ktorej predtým nastavil parameter ReplyByDate. Agent prestane čakať, pretože vypršal časový limit, v ktorom mala prísť odpoveď. Ako argument má táto metóda číslo stavu, v ktorom je volaná (slúži hlavne pre ladiacu fázu).

Platí, že metódy začínajúce handle, môžu byť preprogramované podľa potrieb agenta. Keď nie sú preprogramované, komunikácii podľa protokolu to nevadí. Sú to taktiež reakcie agenta na rôzne udalosti (prijatie rôznych typov správ, vypršanie časuového limitu, v ktorom mala prísť odpoveď...). Avšak metódy prepareResponse, prepareVetoResponse musia byť preprogramované, pretože inak vracajú null, čo je žiadna správa a komunikácia sa v tomto bode zastaví.

Správanie agenta počas dialógu pomocou VETO protokolu závisí hlavne na preprogramovaných metódach.

7. Kapitola : Príklad používania VETO protokolu

Keď už máme naprogramovaný VETO protokol, skúsme si ukázať, ako s ním pracujú agenti. Iniciátor si musí naprogramovať triedu s názvom napríklad myVetoInitiator (samozrejme ju môže nazvať aj inak, ale v ďaľšom texte

predpokladám tento názov), čo je trieda, ktorá rozširuje mnou vytvorenú triedu VetoInitiator o preprogramované metódy, ktoré som spomínal v kapitole 6.1. Respondér si podobne preprogramuje triedu VetoResponder a nazve ju napríklad myVetoResponder (preprogramuje metódy, ktoré som uvádzal v kapitole 6.2, ďalej v texte budem predpokladať, že si svoju preprogramovanú triedu nazval myVetoResponder, aj keď ju môže nazvať inak). Keď chceme, aby nejaký agent A (iniciátor) začal komunikáciu s druhým agentom B (respondér) pomocou VETO protokolu, mal by si najskôr agent A vytvoriť správu, ktorú chce poslať ako prvú agentovi B (vzhľadom k VETO protokolu by to mala byť správa typu REQUEST). Keď má agent A vytvorenú túto prvú správu, pridá si medzi svoje behaviour (pomocou void jade.core.Agent.addBehaviour(Behaviour b)) myVetoInitiator s 2 argumentmi :

1. samotná trieda agenta A
2. druhý parameter je tá pripravená prvá správa.

Avšak agent A si môže pridať myVetoInitiator len s prvým argumentom, bez druhého argumentu správy. V tomto prípade je ale nutné, aby v triede myVetoInitiator bola preprogramovaná metóda - ACLMessage prepareRequest(ACLMessage msg), ktorá nám bude vracať prvú správu na odoslanie (mala by byť typu REQUEST a predpokladajme, že príjemca správy je B). Keď si agent A pridá toto behaviour, začína jeho komunikácia s agentom B podľa VETO protokolu.

Programátor agenta B nemusí vopred vedieť, s akými agentami bude agent B komunikovať. Preto agent B pridáva svoje myVetoResponder behaviour až potom, ako príjme správu od nejakého iného agenta, v ktorej bude definovaný protokol VETO (to sa dá pomocou vyplnenia parametra protocol, ktorý má každá ACL správa). Teda agent B príjme správu od agenta A (s definovaným VETO protokolom) a pridá si behaviour myVetoResponder s dvoma argumentmi :

1. samotná trieda agenta B
2. práve tá prijatá správa

V tomto prípade musia byť definované oba argumenty. Po tom, ako si aj agent B pridá svoje behaviour myVetoResponder, už prebieha dialóg medzi oboma agentami a ovplyvňovanie správania a reakcií agentov na rôzne situácie, ktoré môžu nastať počas tohto dialógu, sú určené práve pomocou preprogramovaných metód v myVetoInitiator resp. myVetoResponder.

Agent ma taktiež možnosť ovplyvňovať fungovanie VETO protokolu. Napríklad agent B (respondér) sa môže rozhodnúť, či dovolí, aby mohlo prebiehať viacero dialógov podľa VETO protokolu s jedným agentom A zároveň. Predstavme si situáciu, že agent A si pridá zároveň aspoň dve behaviour myVetoInitiator, pričom príjmateľ správ v argumente týchto behaviour bude B (takže bude viacero dialógov medzi agentami A a B podľa VETO protokolu zároveň). Agent B má viacero možností. Napríklad sa môže rozhodnúť, že bude vybavovať dialógy s agentom A postupne, alebo môže tieto dialógy vybavovať zároveň. Teda keď agent B chce, aby prebiehal vždy maximálne jeden dialóg medzi agentami A a B zároveň, preprogramuje metódu done() v myVetoResponder tak, aby vracala false. Keď príjme prvú správu od agenta A, pridá si behaviour myVetoResponder s touto prvou prijatou správou od agenta A (táto správa je typu REQUEST a má nastavený protokol VETO). Je to jediný krát, kedy si agent B pridáva behaviour myVetoResponder pre agenta A. Tým, že toto behaviour nikdy neskončí, bude sa starať o každý nasledujúci dialóg medzi agentami A a B podľa VETO protokolu. Týmto spôsobom sa dá aj obmedziť maximálny počet dialógov medzi agentami A a B podľa VETO protokolu zároveň (keď spustím napríklad len pre päť prvých správ od agenta A behaviour myVetoResponder, bude môcť bežať zároveň len päť dialógov medzi agentami A a B podľa VETO protokolu). Keď agent B chce vybavovať ľubovoľné množstvo takýchto dialógov, môže to urobiť tak, že preprogramuje metódu done() v myVetoResponder tak, aby vracala true a pridáva si behaviour myVetoResponder po každom prijatí správy typu REQUEST s definovaným protokolom VETO a rôznym ConversationId. Tým agent B dosiahne, že pre každý dialóg existuje práve jedno myVetoResponder behaviour, a po skončení dialógu sa príslušné behaviour ukončí a vymaže. Agent A (iniciátor) to má trošku jednoduchšie, keďže sám rozhoduje o začatí dialógu. Keď chce začať zároveň viacero dialógov, pridá si viacero myVetoInitiator behaviour zároveň. Keď chce, aby prebiehali dialógy jeden po druhom, pridá si druhé myVetoInitiator behaviour až po skončení toho prvého myVetoInitiator behaviour.

7.1 Príklad agenta Kupec a agenta Skladnik

Zatiaľ si agenti posielali len správy, ktoré majú štruktúru, ako určuje FIPA ACL špecifikácia. Táto štruktúra správy určuje, ako ma správa vyzerať "zvonku". To znamená, aké má mať parametre a hodnoty parametrov. Pre protokol sú najdôležitejšie parametre odosielateľa, príjemcu, protokolu a výkonný parameter. Parametre odosielateľ a príjemca sú mená agentov (ak sú komunikujúci agenti v rovnakej platforme, stačí iba meno agenta, a ak nie, je potrebné celé AID). Parameter protokolu je názov protokolu. FIPA definovala nejaké protokoly ako napríklad FIPA-QUERY, FIPA-REQUEST, FIPA-PROPOSE, plus môžeme použiť nami definované (doplnené) protokoly (napríklad protokol VETO). Výkonný parameter môže byť jeden z FIPA definovaných parametrov, a to: ACCEPT-PROPOSAL, AGREE, CANCEL, CFP, CONFIRM, DISCONFIRM, FAILURE, INFORM, INFORM-IF, INFORM-REF, NOT-UNDERSTOOD, PROPOSE, QUERY-IF, QUERY-REF, REFUSE, REJECT-PROPOSAL, REQUEST, REQUEST-WHEN, REQUEST-WHENEVER, SUBSCRIBE, PROXY a PROPAGATE, plus to môže byť jeden z mojich doplnených JUSTIFY, CHALLENGE, VETO, RETRACT.

Skúsme sa teraz viac zaoberať obsahom správy. Obsah správy je hodnota obsahového parametra podľa ACL. ACL už ďalej nešpecifikuje, aké hodnoty má mať tento parameter. Na túto špecifikáciu existujú viaceré obsahové jazyky. Medzi takéto jazyky patrí aj SL Content Language, o ktorého špecifikáciu sa postarala taktiež FIPA. FIPA špecifikovala syntax a sémantiku tohto jazyka. SL Content Language sa má používať zároveň s ACL jazykom, kde SL definuje "vnútro" správy.

Uvažujme nasledujúci príklad. Máme skladníka, ktorý obsluhuje tri druhy tovaru, a máme kupca, ktorý potrebuje určitý druh tohto tovaru v určitom množstve. Kupec poprosí skladníka o určitý druh a množstvo tovaru, a skladník mu buď vyhovie, ak ho má na sklade, alebo nevyhovie, ak ho nemá na sklade, alebo ak má nejaké množstvo tohto tovaru na sklade a kupec chce viac, ponúkne kupcovi množstvo, ktoré má k dispozícii. A zároveň má skladník právo vetovať kupcovu žiadosť, to znamená, že bez udania dôvodu môže kupcovi odmietnuť vydať tovar. Avšak, keď raz vetuje kupcovu žiadosť, v ďalšom vyjednávaní už nemôže zamietnuť a ani vetovať žiadnu kupcovu žiadosť. Je vidno, že tento príklad je priamo robený na Veto protokol.

Skladník je respondér a kupec je iniciátor. Na dohovorenie medzi skladníkom a kupcom, ktorí si posielajú správy, je nutné vyplniť obsah týchto správ.

SL jazyk je veľmi expresívny a pre niektoré komunikačné úlohy je zbytočne silný, a teda aj implementačne a výpočtovo náročný. Toto platí aj pre náš príklad s kupcom a skladníkom. Preto FIPA vytvorila podmnožiny SL jazyka. Podľa stúpajúcej úrovne expresivity sú to SL0, SL1 a SL2. Pre náš príklad postačí minimálna podmnožina SL jazyka, teda SL0 (pre priblíženie jazyka SL ako aj SL0 odporúčam [5]).

Tento príklad som naprogramoval aj v JADE. Vytvoril som dve triedy agentov, a to SkladnikAgent2 a KupecAgent2. Najskôr si vytvorím agenta Skladnik z triedy SkladnikAgent2. Potom vytvorím agenta Kupec z triedy KupecAgent2. Kupec bude iniciátor dialógu a Skladnik bude respondér dialógu. Triedy SkladnikAgent2 aj KupecAgent2 som implementoval tak, aby mohli viesť zároveň ľubovoľné množstvo dialógov (či už dialógy medzi rôznymi agentami, ako aj medzi dvoma rovnakými agentami). Zároveň agent Skladnik si pridáva jedno myVetoResponder behaviour pre každý dialóg, a keď dialóg dospeje do koncového stavu, toto behaviour sa ukončí. Taktiež som preprogramoval metódy spomínané v 6. kapitole, a teda agenti fungujú nasledovne.

Kupec najskôr pošle správu typu REQUEST Skladnikovi. Obsah tejto správy bude: "(action (agent-identifier : name Skladnik) (give_me (Co_chcem)) (Kolko_chcem))", kde Co_chcem je náhodne zvolený druh tovaru, Kolko_chcem je náhodne zvolené množstvo tovaru. Agent Skladnik príjme túto správu a zistí, čo a koľko toho chce Kupec. Teraz môžu nastať 4 situácie.

1. Skladnik má aj druh tovaru, ktorý Kupec chce, a má aj množstvo, ktoré chce. V takomto prípade Skladnik dá Kupcovi, čo žiada, a pošle mu správu typu AGREE s obsahom: "(action (agent-identifier : name Skladnik) (give_you (Co)) (Kolko))", kde Co je druh tovaru, ktorý dáva Skladnik Kupcovi, a Kolko je množstvo tohto tovaru.
2. Skladnik nemá na sklade ani jeden kus z toho druhu tovaru, ktorý žiada Kupec. V takomto prípade Skladnik zamietne Kupcovu žiadosť a pošle mu správu typu REFUSE s obsahom: "refuse_request +request", kde +request je správa typu REQUEST od Kupca

3. Skladnik má na sklade druh tovaru, ktorý požaduje Kupec, avšak nemá požadované množstvo. V takomto prípade Skladnik ponúkne Kupcovi také množstvo tovaru, aké má na sklade, a pošle mu správu typu CHALLENGE s obsahom: "(action (agent-identifier : name Skladnik) (may_give_you (Co_dat)) (Kolko_dat))", kde Co_dat je druh tovaru, ktorý žiadal Kupec a Kolko_dat je množtvo tohto tovaru na sklade.
4. Skladnik bez uvedenia dôvodu využije svoje právo veta a vetuje Kupcovu žiadosť. Pošle Kupcovi správu typu VETO s obsahom: "(action (agent-identifier : name " Skladnik") (veto (+requestContent)))", kde +requestContent je obsah správy typu REQUEST, alebo JUSTIFY (z prvého vyjednávacieho cyklu, je to posledná správa zo stavu 2 podľa Obrázka 5.3) od Kupca.

Ak Kupec obdrží správu typu AGREE, je spokojný, že svoju úlohu splnil, a ukončí dialóg. Skladnik vie, že Kupec ukončil dialóg, a tiež ukončí dialóg.
Ak Kupec obdrží správu typu REFUSE, je síce nespokojný, ale už nemôže nič robiť, tak ukončí dialóg. Skladnik vie, že Kupec ukončil dialóg, a tak ukončí aj agent Skladnik svoj dialóg.
Ak Kupec obdrží správu typu CHALLENGE, zistí, čo mu ponúka Skladnik, a rozhodne sa z dvoch možností:

1. Kupec nechce to, čo mu Skladnik ponúka, a preto stiahne svoju žiadosť a pošle Skladnikovi správu typu RETRACT s obsahom: "(action (agent-identifier : name Kupec) (retract (+challenge.getContent)))", kde +challengeContent je obsah spravy typu Challenge, ktorú poslal Kupcovi Skladnik. Kupec neuspel vo svojom snažení a ukončí dialóg, Skladnik vie, že Kupec ukončil dialóg, a tak ukončí aj agent Skladnik svoj dialóg.
2. Kupec súhlasí s tým, čo mu ponúka Skladnik. Pošle Skladnikovi správu typu JUSTIFY s obsahom: "(action (agent-identifier : name Skladnik) (give_me ("Co_chcem")) ("Kolko_chcem"))", kde Co_chcem je druh tovaru, ktorý Kupec chcel, a Kolko_chcem je množstvo tovaru, ktoré mu môže Skladnik dať. Týmto sa dostane do podobnej situácie, ako keď Kupec posielal Skladnikovi

svoju prvú správu typu REQUEST, akurát teraz Kupec posiela správu typu JUSTIFY.

Ak Kupec obdrží správu typu VETO, vie, že Skladnik využil svoje právo vetovať, a preto skúsi poslať Skladnikovi novú žiadosť. Pošle mu správu typu REQUEST s obsahom: "(action (agent-identifier : name Skladnik) (give_me (Co_chcem)) (Kolko_chcem))", kde Co_chcem je náhodne zvolený druh tovaru, Kolko_chcem je náhodne zvolené množstvo tovaru. Agent Skladnik príjme túto správu a zistí, čo a koľko toho chce Kupec. Teraz môžu nastať už len 2 situácie.

1. Skladnik má aj druh tovaru, ktorý kupec chce a má aj požadované množstvo. V takomto prípade Skladnik dá Kupcovi, čo žiada, a pošle mu správu typu AGREE s obsahom: "(action (agent-identifier : name Skladnik) (give_you (Co)) (Kolko))", kde Co je druh tovaru, ktorý dáva Skladnik Kupcovi a Kolko je množstvo tohto tovaru.
2. Skladnik nemá toľko kusov (zo žiadaného druhu tovaru), koľko Kupec žiada. V takomto prípade Skladnik ponúkne Kupcovi také množstvo tovaru, aké má na sklade, a pošle mu správu typu CHALLENGE s obsahom: "(action (agent-identifier : name Skladnik) (may_give_you (Co_dat)) (Kolko_dat))", kde Co_dat je druh tovaru, ktorý žiadal Kupec a Kolko_dat je množstvo tohto tovaru na sklade. Kolko_dat môže byť v tomto prípade aj 0.

Ak Kupec obdrží správu typu AGREE, je spokojný, že svoju úlohu splnil, a ukončí dialóg. Skladnik vie, že Kupec ukončil dialóg, a tak tak ukončí aj agent Skladnik svoj dialóg.

Ak Kupec obdrží správu typu CHALLENGE, zistí, čo mu ponúka Skladnik, a rozhodne sa z dvoch možností:

1. Kupec nechce to, čo mu Skladnik ponúka, alebo mu Skladnik ponúka 0 kusov, a stiahne svoju žiadosť. Pošle Skladnikovi správu typu RETRACT s obsahom: "(action (agent-identifier : name Kupec) (retract (+challenge.getContent)))", kde +challengeContent je obsah správy typu CHALLENGE, ktorú poslal Kupcovi Skladnik. Kupec neuspel vo svojom snažení a ukončí dialóg, Skladnik vie, že Kupec ukončil dialóg, a tak ukončí aj agent Skladnik svoj dialóg.

2. Kupec súhlasí s tým, čo mu ponúka Skladnik. Pošle Skladnikovi správu typu JUSTIFY s obsahom: "(action (agent-identifier : name Skladnik) (give_me ("Co_chcem")) ("Kolko_chcem"))", kde Co_chcem je druh tovaru, ktorý Kupec chcel, a Kolko_chcem je množtvo tovaru, ktoré mu môže Skladnik dať. Týmto sa dostane do podobnej situácie, ako keď Kupec posielal Skladnikovi svoju správu typu REQUEST po tom, ako obdržal VETO, akurát teraz Kupec posiela správu typu JUSTIFY.

Toto by mal byť základný priebeh konverzácie medzi dvoma agentami, ktorí sa držia Veto protokolu. Agent Kupec je naprogramovaný tak, že keď pri jeho vytváraní mu dáme viacero argumentov (argumenty sú mená agentov), začne komunikovať so všetkými týmito agentami súčasne (teda Kupec môže získavať suroviny od viacerých skladníkov súčasne). Agent Skladnik je naprogramovaný tak, že taktiež môže naraz komunikovať s viacerými agentami súčasne. Agentov skladníkov som naprogramoval tak, aby všetci zdieľali jeden sklad. To znamená, že keď napríklad Skladník1 vydá tovar nejakému kupcovi, tak Skladník2 bude mať tohto tovaru o vydané množstvo menej. Tu môže nastať situácia, keď vyjednávací cyklus pôjde viackrát. Predstavme si, že agent Kupec1 (odvodený z triedy KupecAgent2) chce od agenta Skladnik1 (odvodeného z SkladnikAgent2) tovar druhu 1 v množstve 100. Avšak agent Skladnik1 má na sklade len 50 kusov tohto tovaru. Teda pošle agentovi Kupec1 správu typu CHALLENGE, kde mu ponúkne 50 kusov tohto tovaru. Agent Kupec1 s tým súhlasí a pošle agentovi Skladnik1 správu typu JUSTIFY, v ktorej pýta tovar druhu 1 a množstvo 50. Avšak medzitým agent Skladnik2 (odvodený z SkladnikAgent2) už poslal 30 kusov tovaru druhu 1 inému agentovi (a keďže agent Skladnik1 aj Skladnik2 zdielajú spoločný sklad na sklade bude len 20 kusov tovaru druhu 1). Teda potom ako agent Skladnik1 príjme správu typu JUSTIFY od agenta Kupec1, v ktorej pýta 50 kusov tovaru druhu 1, už agent Skladnik1 nemá toto množtvo na sklade a preto znova pošle agentovi Kupec1 správu typu CHALLENGE, kde mu ponúkne 20 kusov tovaru druhu 1. Tento cyklus sa môže opakovať ľubovoľný počet krát.
Taktiež som do príkladu zahrnul možnosť neposlania správy. To znamená, že agent sa v ktoromkoľvek stave môže rozhodnúť neposlať správu.

Keď sa pozrieme na obsah správ, majú podobnú formu. Najskôr je "(action (agent-identifier : name Skladnik) …)", čo značí, že chceme vykonať nejakú akciu, a hneď zadáme, kto ju má vykonať. Za meno vykonávateľa zadávame (NÁZOV_AKCIE ("PARAMETER_AKCIE")) a nakoniec zadáme MNOŽSTVO. Názov akcie by mali poznať obaja účastníci komunikácie.

Pre ukážku a kontrolu tohto príkladu môžeme spustiť JADE s nasledujúcimi argumentmi pre agentov :

- pre základnú komunikáciu medzi 1 agentom Skladnik (odvodený z SkladnikAgent2) a 1 agentom Kupec (odvodený z triedy KupecAgent2) napíšeme ako argument pre spustenie agentov : "Skladnik:examples.protocols.SkladnikAgent2;Kupec:examples.protocols.KupecAgent2(Skladnik)"
- pre komunikáciu medzi piatimi agentami odvodenými zo SkladnikAgent2 a jedným agentom Kupec (odvodený z triedy KupecAgent2) napíšeme ako argument pre spustenie agentov nasledujúci výraz : "Skladnik:examples.protocols.SkladnikAgent2;Skladnik2:examples.protocols.SkladnikAgent2;Skladnik3:examples.protocols.SkladnikAgent2;Skladnik4:examples.protocols.SkladnikAgent2;Skladnik5:examples.protocols.SkladnikAgent2; Kupec:examples.protocols.KupecAgent2(Skladnik,Skladnik2,Skladnik3,Skladnik4,Skladnik5)"
- pre komunikáciu medzi piatimi agentami odvodenými z KupecAgent2 a jedným agentom Skladnik (odvodený z triedy SkladnikAgent2) napíšeme ako argument pre spustenie agentov nasledujúci výraz : "Skladnik:examples.protocols.SkladnikAgent2;Kupec:examples.protocols.KupecAgent2(Skladnik);Kupec2:examples.protocols.KupecAgent2(Skladnik);Kupec3:examples.protocols.KupecAgent2(Skladnik);Kupec4:examples.protocols.KupecAgent2(Skladnik);Kupec5:examples.protocols.KupecAgent2(Skladnik)"
- pre komunikáciu medzi troma agentami odvodenými zo SkladnikAgent2 a troma agentami odvodenými z KupecAgent2 napíšeme ako argument pre spustenie agentov : "Skladnik:examples.protocols.SkladnikAgent2;Skladnik2:examples.protocols.SkladnikAgent2;Skladnik3:examples.protocols.SkladnikAgent2;Kupec:example

s.protocols.KupecAgent2(Skladnik,Skladnik2,Skladnik3);Kupec2:examples.protocols.KupecAgent2(Skladnik,Skladnik2,Skladnik3);Kupec3:examples.protocols.KupecAgent2(Skladnik,Skladnik2,Skladnik3)"

- a ešte pre komunikáciu medzi 1 agentom Skladnik (odvodený z triedy SkladnikAgent2) a 1 agentom Kupec (odvodený z triedy KupecAgent2), pričom agent Kupec bude chcieť viesť zároveň tri dialógy s agentom Skladnik podľa VETO protokolu. Pre tento prípad napíšeme ako argument pre spustenie agentov : "Skladnik:examples.protocols.SkladnikAgent2;Kupec:examples.protocols.KupecAgent2(Skladnik,Skaldnik,Skladnik)"

Tieto vstupné argumenty môžeme ľubovoľne upravovať. Napríklad počet agentov môže byť ľubovoľný a aj agenti odvodení z triedy KupecAgent2 môžu mať ľubovoľný počet argumentov. Na priloženom CD mám aj ukážky takýchto konverzácií.

Nešlo mi o to, aby som vytvoril triedy pre racionálnych agentov, ale išlo mi o to, aby som otestoval a ukázal fungovanie VETO protokolu. Použitie racionálnych agentov by zneprehľadnilo príklad komunikácie medzi agentami a ja poukazujem hlavne na komunikáciu a používanie protokolov a chcem, aby použité príklady boli čo najjednoduchšie a najprehľadnejšie. Zároveň ale platí, že VETO protokol ako aj spôsob komunikácie tak, ako ho ukazujem v mojich príkladoch, je vhodný aj pre racionálnych agentov. Rozdiel je v tom, že v mojich príkladoch je rozhodovanie viacmenej náhodné, alebo predurčené. Zatiaľčo rozhodovanie u racionálnych agentov je komplikovanejšie. Avšak prvky komunikácie medzi agentami sú zhodné aj pre racionálnych agentov (prvky ako vyplnenie parametrov správ, príjmanie a odosielanie správ ...). Racionálni JADE agenti komunikujú tak ako ostatní JADE agenti. Napríklad použitie BDI modelu na ukážku komunikácie podľa VETO protokolu by bolo ako použiť kanón na vrabce.

Prispôsobivosť agentov danému protokolu.

Agent je prispôsobivý danému protokolu, ak jeho behaviour je legálne s ohľadom k danému protokolu. Podľa [7] rozlišujeme 3 úrovne prispôsobenia agenta k danému protokolu:

1) Slabá prispôsobivosť – agent je slabo prispôsobivý k danému protokolu práve vtedy, keď nikdy nepošle takú odpoveď, ktorá nie je správnym následkom (vzhľadom k danému protokolu) v nejakom dialógovom stave.
2) Úplná prispôsobivosť – agent je úplne prispôsobivý k danému protokolu práve vtedy, keď je slabo prispôsobivý k danému protokolu a zároveň na každú správnu správu pošle aspoň jednu odpoveď, ktorá je správna (vzhľadom k danému protokolu).
3) Robustná prispôsobivosť – agent je robustne prispôsobivý k danému protokolu práve vtedy, keď je úplne prispôsobivý a na každú nesprávnu správu odpovie pomocou špeciálnej správy (napr. NOT-UNDERSTOOD).

Každý agent, ktorý chce používať nejaký protokol, musí byť aspoň slabo prispôsobivý k tomuto protokolu, inak nemá vôbec zmysel používať tento protokol. Často chceme, aby bol agent úplne prispôsobivý k danému protokolu. Týmto sa vyhneme tzv. tichej odpovedi a zmätkom okolo straty správ. Avšak niekedy môže byť tichá odpoveď žiadúca. To je napríklad holandská aukcia (v tradičnej holandskej aukcii začne vyvolávač s vysokou vyvolávanou cenou, ktorá je znižovaná, pokiaľ nieje niekto z účastníkov ochotný akceptovat vyvolávačovú cenu. Účastník, ktorý v aukci vyhrál, zaplatí oznámenou cenu). Keby sme vedeli, že všetci agenti sú aspoň slabo prispôsobiví, tak by sme už nepotrebovali definovať robustnú prispôsobivosť. Avšak MAS je otvorený systém a tento predpoklad nemusí platiť.

Čo sa týka agentov odvodených z tried SkladnikAgent2 a KupecAgent2, tak som sa ich snažil naprogramovať tak, aby boli robustne prispôsobiví k Veto protokolu. Avšak po pridaní možnosti, že nepošlú správu, sa stávajú títo agenti slabo prispôsobivý k VETO protokolu.

8. Kapitola : Príklad rozširovania MAS

Výhoda toho, že som implementoval pomerne zložitý vyjednávací VETO protokol je v tom, že keď chcem JADE rozšíriť o ďalší vyjednávací protokol, môžem vychádzať a pomôcť si už pripraveným VETO protokolom. VETO protokol zahŕňa rôzne situácie. Ako napríklad situáciu, keď sa po prijatí správy typu REQUEST

respondér dostane do dvoch rôznych stavov. Navyše prijatie správy typu REQUEST je aj spúšťačom dialógu. Teda respondér príjme správu typu REQUEST s definovaným protokolom VETO. Môžu nastať dve možnosti :

1. prijatá správa je prvá správa typu REQUEST, ktorú posiela iniciátor, aby začal nový dialóg podľa VETO protokolu
2. prijatá správa je odpoveď iniciátora na správu typu VETO od respondéra a teda nezačína nový dialóg podľa VETO protokolu, ale posúva už prebiehajúci dialóg do ďalšieho stavu

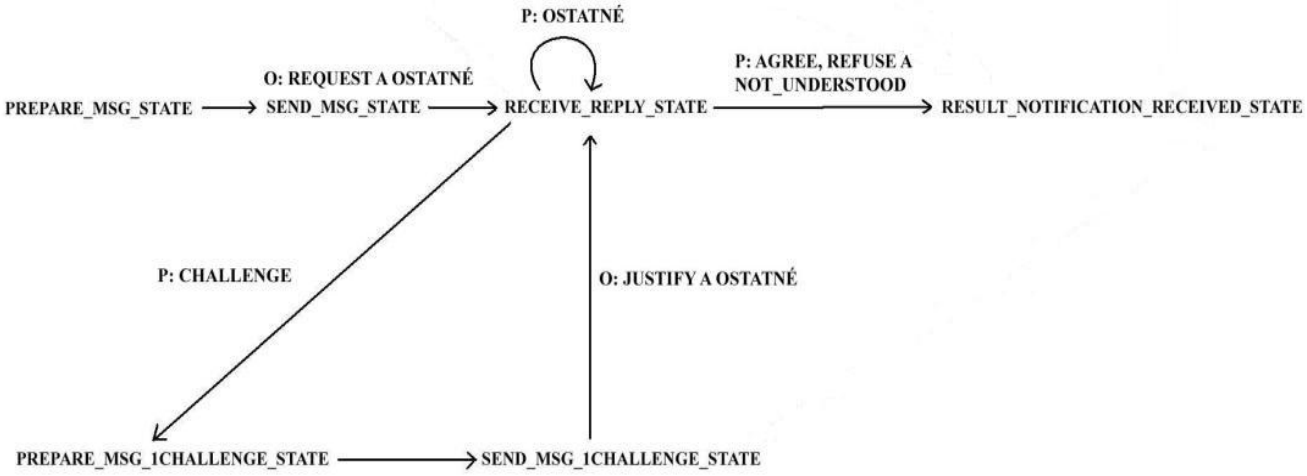
8.1 NEGO protokol :

Skúsme teraz rozšíriť JADE o ďalší vyjednávací protokol. Nech je to protokol z Obrázka 5.1. Je to jednoduchý, ale veľmi používaný vyjednávací protokol. Označím si tento protokol ako NEGO protokol. Podľa NEGO protokolu sa vyjednáva nasledujúcim spôsobom. Agent A (iniciátor) pošle agentovi B (respondér) správu typu REQUEST s definovaným protokolom NEGO a tým začne dialóg medzi A a B podľa NEGO protokolu. Agent B po prijatí tejto správy má tri možnosti :

1. zamietne žiadosť – pošle agentovi A správu typu REFUSE a ukončí dialóg
2. príjme žiadosť – pošle agentovi A správu typu AGREE a ukončí dialóg
3. začne vyjednávať – pošle agentovi A správu typu CHALLENGE

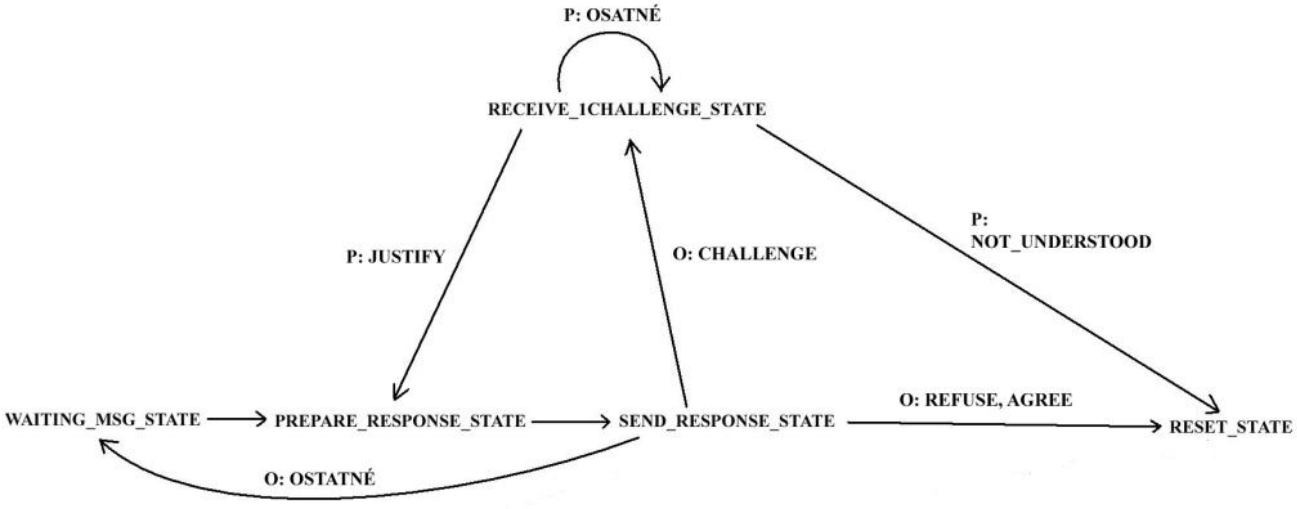
Agent A po prijatí správy typu AGREE, resp. REFUSE ukončí dialóg. Po prijatí správy CHALLENGE pokračuje vo vyjednávaní a posiela správu typu JUSTIFY agentovi B, ktorý je následne v stave, ako bol po obdržaní prvej správy typu REQUEST. NEGO protokol má jeden vstupný stav (stav 0 na Obrázku 5.1) a dva koncové stavy (stavy 3 a 4 na Obrázku 5.1).

Je vidno, že NEGO protokol môžeme implementovať ako VETO protokol s vynechanými stavmi. VETO protokolu vynecháme stavy 3, 6, 7, 8 podľa Obrázka 5.3. Takže po vynechaní týchto stavov, bude protokol pre NEGO iniciátora vyzerať, ako je znázornené na Obrázku 8.1.



Obrázok 8.1

Protokol pre NEGO respondéra je znázornený na Obrázku 8.2.



Obrázok 8.2

Ďalej, agent iniciátor má možnosť preprogramovať nasledujúce metódy :

- **ACLMessage prepareRequest(ACLMessage msg)**
- **ACLMessage prepareChallengeResponse(ACLMessage challenge)**
- **void handleAgree(ACLMessage agree)**

- **void handle1Challenge(ACLMessage challenge)**
- **void handleRefuse(ACLMessage refuse)**
- **void handleNotUnderstood(ACLMessage not_understood)**
- **void handleOutOfSequence(ACLMessage msg)**
- **void handleTimeoutExpired(int state_nmr)**

a agent respondér má možnosť preprogramovať nasledujúce metódy :

- **ACLMessage prepareResponse(ACLMessage msg)**
- **void handleRequest(ACLMessage request)**
- **void handleJustify(ACLMessage justify)**
- **void handleNotUnderstood(ACLMessage not_understood)**.
- **void handleOutOfSequence(ACLMessage msg)**
- **void handleTimeoutExpired(int state_nmr)**

Všetky tieto metódy majú podobný význam ako u VETO protokolu. Až na to, že metóda - ACLMessage prepareChallengeResponse(ACLMessage challenge) nevracia správu typu RETRACT a metóda - ACLMessage prepareResponse(ACLMessage msg) nevracia správu typu VETO.

Ja som doplnil JADE aj o tento protokol. Máme triedu NegoInitiator pre agenta iniciátora, ktorý chce používať NEGO protokol, a triedu NegoResponder pre agenta respondéra, ktorý chce používať NEGO protokol.

Ako ukážku použitia NEGO protokolu som implementoval dve triedy agentov a to : SkladnikAgent3 a KupecAgent3. Tieto triedy sú podobné ako triedy SkladnikAgent2 a KupecAgent2, akurát namiesto VETO protokolu komunikujú pomocou NEGO protokolu.

Na príklade NEGO protokolu vidíme, že keď máme už implementovaný VETO protokol, tak implementovať NEGO protokol už je pomerne jednoduché. V NEGO protokole sme však len vynechávali stavy z VETO protokolu. Skúsim ukázať zložitejší príklad, kde treba do VETO protokolu aj dopĺňať stavy.

8.2 Príklad : Hlavný sklad

Predstavme si nasledujúcu situáciu. Opäť máme agentov skladníkov a agentov kupcov, ktorí pracujú, ako bolo popísané v kapitole 7.1. Teda kupec sa snaží získať od skladníka nejaký druh tovaru v určitom množstve. Predpokladajme, že jeden skladník má na starosti svoj jeden sklad. Kupec aj skladník budú chcieť medzi sebou komunikovať podľa VETO protokolu. Bližšie si všimnime stav 6 podľa Obrázka 5.3. V tomto stave skladník (respondér) príjme žiadosti od kupca (buď priamo správu typu REQUEST, alebo správu typu JUSTIFY z vyjednávacieho cyklu). Skldaník môže na tieto žiadosti odpovedať dvoma spôsobmi :

1. prijať žiadosť – respondér odosiela iniciátorovi správu typu AGREE
2. začať vyjednávať - respondér odosiela iniciátorovi správu typu CHALLENGE

Keď kupec ale pýtal tovar, ktorý skladník nemal na sklade, tak mu skladník posielal správu typu CHALLENGE, kde mu ponúkal nulové množstvo tovaru (v stave 1 podľa Obrázka 5.3, by skladník poslal kupcovi správu typu REFUSE). Všimnime si, že je to niečo iné, ako keby skladník posielal namiesto správy typu CHALLENGE s nulovým množstvom, správu typu REFUSE. Je to, ako keď zákazník príde do obchodu a vypýta si od predavačky nejaký tovar. Je rozdiel, či predavačka povie zákazníkovi, že mu žiadaný tovar nedá, alebo mu povie, že žiadaný tovar nemá. Pretože keď zákazníkovi povie, že tovar mu nedá, zákazník nevie, či predavačka vôbec nejaký tovar má, alebo je len neochotná mu ho vydať.

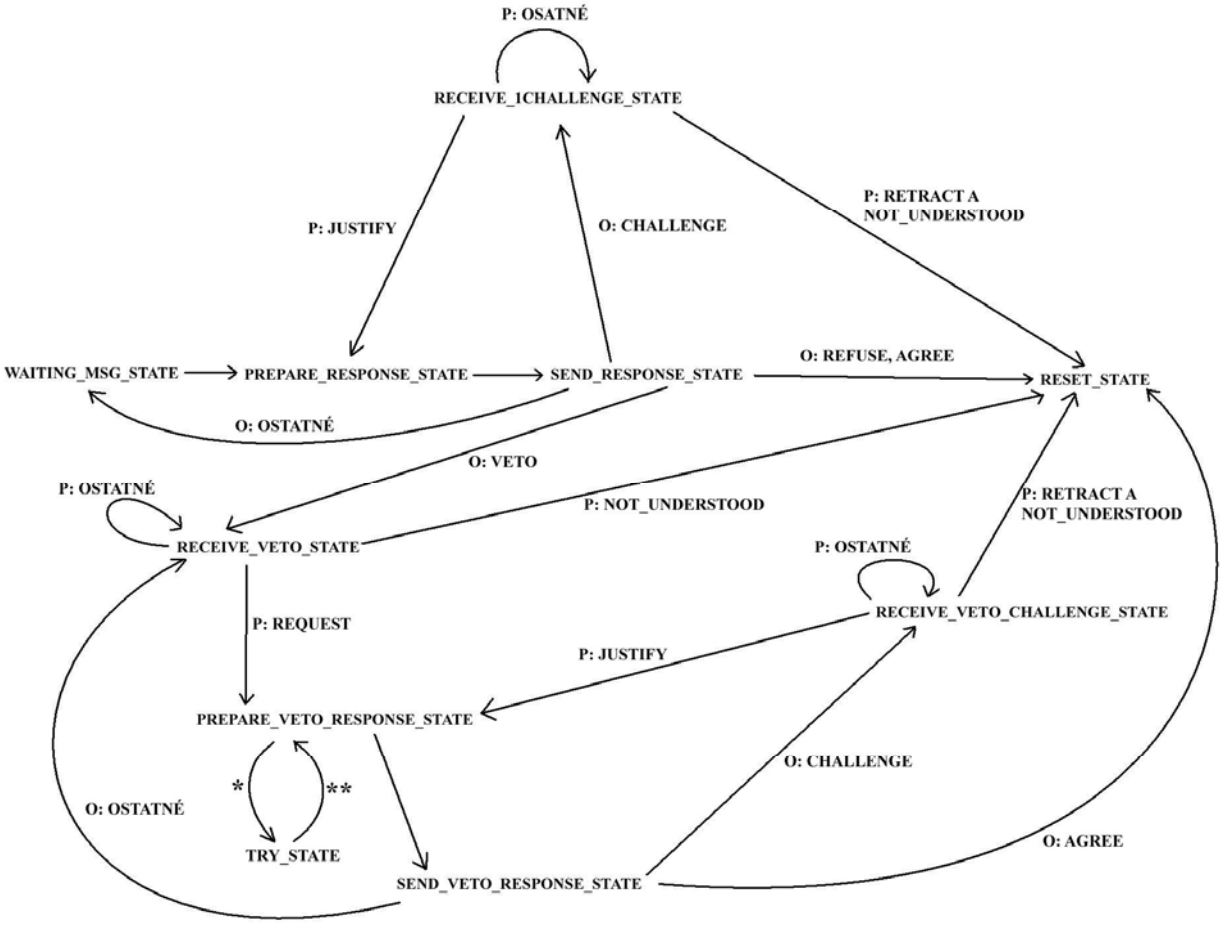
Skúsme tento príklad rozšíriť. Predstavme si, že existuje ešte hlavný sklad. Tento hlavný sklad dodáva tovar ostatným skladom. Keď sme teda v stave 6 podľa Obrázka 5.3 a kupec pýta od skladníka tovar, ktorý skladník nemá na sklade, začne skladník pýtať tento tovar z hlavného skladu, a keď mu dá hlavný sklad tento tovar, ponúkne ho kupcovi. Ak hlavný sklad vydá skaldníkovi požadované množstvo tovaru, skladník odsúhlasí kupcovu žiadosť pomocou správy typu AGREE. Ak hlavný sklad vydá skladníkovi menšie množstvo tovaru, ako skladník požadoval (teda aj ako požadoval kupec), ponúkne skladník tento tovar kupcovi pomocou správy typu CHALLENGE. A až keď hlavný sklad nevydá žiadajúcemu skladníkovi žiadny tovar,

posiela skladník kupcovi správu typu CHALLENGE s ponukou nulového množstva (to znamená, že skladník nemá požadovaný tovar na sklade).

Skladník si žiada od hlavného skladu tovar pomocou komunikácie, ktorá bude prebiehať podľa NEGO protokolu. Viacero skladníkov môže mať spoločný hlavný sklad, ako aj každý skladník môže mať vlastný hlavný sklad.

Všimnime si, že v tomto príklade je kupec rovnaký ako v príklade z kapitoly 7.1, takže na jeho vytvorenie môžem pužiť triedu KupecAgent2. Kupec teda bude aj komunikovať so skladníkmi pomocou VETO protokolu tak, ako v príklade z kapitoly 7.1, teda pre začiatok dialógu si bude pridávať VetoInitiator behaviour.

Avšak skladník potrebuje nejaké zmeny. Preto som vytvoril triedu SkladnikAgent3, z ktorej môže byť skladník odvodený. Majme teda agenta Skladnik, odvodeného z triedy SkladnikAgent3. Tento agent musí vedieť komunikovať ako respondér podľa VETO protokolu (pre komunikáciu s kupcami), ako iniciátor podľa NEGO protokolu (v prípade, že žiada niečo od hlavného skladu) a keďže aj hlavný sklad môže byť odvodený z triedy SkladnikAgent3, musí vedieť komunikovať aj ako respondér podľa NEGO protokolu. Teda trieda SkladnikAgent3 je aj iniciátor aj respondér podľa NEGO protokolu zároveň. Schopnosť komunikácie získava táto trieda agentov vďaka vytvoreniu vlastných behaviour odvodených z : VetoResponder2, NegoResponder a NegoInitiator, v ktorých sú preprogramované metódy, ktoré som spomínal vyššie. Všimnime si, že trieda SkladnikAgent3 používa VetoResponder2 behaviour. Toto behaviour je podobné ako VetoResponder behaviour a pohybuje po stavoch tak, ako je to znázornené na Obrázku 8.3.



Obrázok 8.3

(na obrázku skratka O: znamená odoslaná správa a skratka P: znamená prijatá správa)

Na Obrázku 8.3 je vidno, že oproti VetoResponder nám pribudol len stav TRY_STATE. * pri prechode medzi stavom PREPARE_VETO_RESPONSE_STATE a TRY_STATE, znamená, že sa do stavu TRY_STATE dostaneme, len ak iniciátor, ktorý nám poslal správu typu REQUEST resp. JUSTIFY, žiada tovar, ktorý nieje na sklade. A ** pri opačnom prechode znamenajú, že sa späť do stavu PREPARE_VETO_RESPONSE_STATE dostanem potom, ako skončím konverzáciu podľa protokolu NEGO s hlavným skladom.

Všimnime si, že VetoResponder2 behaviour, tak ako aj VetoResponder behaviour, je legálne voči protokolu VETO. To znamená, že neporušuje pravidlá tohto protokolu.

V praxi to vyzerá tak, že sa v stave PREPARE_VETO_RESPONSE_STATE volá metóda ACLMessage prepareVetoResponse(ACLMessage request), v ktorej agent Skladnik môže zistiť, že požadovaný tovar nemá na sklade (ak to nezistí prebieha dialóg presne ako v príklade z kapitoly 7.1). V tomto prípade sa dialóg podľa VETO protokolu, riadený VetoResponder2 behaviour, presunie do stavu TRY_STATE. V tomto stave sa VetoResponder2 behaviour zablokuje (aby umožnil vykonávanie ostatných agentových behaviour) a čaká na príchod prebúdzacej správy. Agent Skladnik začne komunikovať s hlavným skladom podľa NEGO protokolu. Keď skončí tento dialóg medzi agentom Skladnikom a hlavným skladom, pošle agent Skladnik zo svojho NegoInitiator behaviour sám sebe prebúdzaciu správu, ktorú zachytí vo VetoResponder2 behaviour v stave TRY_STATE a tým odblokuje toto behaviour a presunie sa zo stavu TRY_STATE do stavu PREPARE_VETO_RESPONSE_STATE. Zároveň som implementoval agenta Skladnika tak, aby sa počas jedného dialógu presunul maximálne raz do stavu TRY_STATE, teda ak sa už raz agent Skladnik vrátil zo stavu TRY_STATE do stavu PREPARE_VETO_RESPONSE_STATE, potom sa už do stavu TRY_STATE nedostane (samozrejme je na konkrétnej implementácii, ako bude vyzerať táto regulácia).

Všimnime, že agenti odvodení z triedy SkladnikAgent3 môžu viesť zároveň :

- ľubovolný počet komunikácii podľa protokolu VETO, kde majú rolu respondéra
- ľubovolný počet komunikácii podľa protokolu NEGO, kde majú rolu respondéra
- ľubovolný počet komunikácii podľa protokolu NEGO, kde majú rolu iniciátora

Skúsme vytvoriť zložitejší MAS, ktorý sa skladá z :

- z dvoch hlavných skladov, ktoré reprezentujú agenti : HlavnySklad1 a HlavnySklad2

- zo šiestich skladníkov, ktorých reprezentujú agenti : Skladnik1, Skladnik2, Skladnik3, Skladnik4, Skladnik5, Skladnik6
- zo siedmych kupcov, ktorých reprezentujú agenti : Kupec1, Kupec2, Kupec3, Kupec4, Kupec5, Kupec6, Kupec7

Nech agenti Skladnik1, Skladnik2 a Skladnik3 majú ako hlavný sklad HlavnySklad1 a agenti Skladnik4, Skladnik5, Skladnik6 majú ako hlavný sklad HlavnySklad2. Ďalej nech všetci kupujúci agenti požadujú nejaký tovar v nejakom množstve od každého z našich šiestich skladníkov. Pričom platí, že kupujúci agenti si pýtajú tovar od agentov skladníkov komunikáciou, ktorá prebieha podľa VETO protokolu. Ďalej, ak nejaký agent skladník (napríklad Skladnik1) nemá tovar, ktorý pýta najeký agent kupec (napríklad Kupec1) a agent skladník (Skladnik1) už predtým v dialógu s agentom kupcom (Kupec1) odoslal správu typu VETO, skúsi agent skladník (Skladnik1) získať tento tovar zo svojho hlavného skladu a ponúkne tento tovar kupujúcemu agentovi (Kupec1). Komunikácia medzi agentom skladníkom a jeho hlavným skladom prebieha podľa NEGO protokolu.

Pre spustenie takéhoto MAS, môžeme zadať pri spúšťaní JADE zadáme ako argument nasledujúci výraz :

-gui
HlavnySklad1:examples.protocols.SkladnikAgent3;
HlavnySklad2:examples.protocols.SkladnikAgent3;
Skladnik1:examples.protocols.SkladnikAgent3(HlavnySklad1);
Skladnik2:examples.protocols.SkladnikAgent3(HlavnySklad1);
Skladnik3:examples.protocols.SkladnikAgent3(HlavnySklad1);
Skladnik4:examples.protocols.SkladnikAgent3(HlavnySklad2);
Skladnik5:examples.protocols.SkladnikAgent3(HlavnySklad2);
Skladnik6:examples.protocols.SkladnikAgent3(HlavnySklad2);
Kupec1:examples.protocols.KupecAgent2(Skladnik1,Skladnik2,Skladnik3,Skladnik4,
Skladnik5,Skladnik6,Skladnik7);
Kupec2:examples.protocols.KupecAgent2(Skladnik1,Skladnik2,Skladnik3,Skladnik4,
Skladnik5,Skladnik6,Skladnik7);

Kupec3:examples.protocols.KupecAgent2(Skladnik1,Skladnik2,Skladnik3,Skladnik4,
Skladnik5,Skladnik6,Skladnik7);
Kupec4:examples.protocols.KupecAgent2(Skladnik1,Skladnik2,Skladnik3,Skladnik4,
Skladnik5,Skladnik6,Skladnik7);
Kupec5:examples.protocols.KupecAgent2(Skladnik1,Skladnik2,Skladnik3,Skladnik4,
Skladnik5,Skladnik6,Skladnik7);
Kupec6:examples.protocols.KupecAgent2(Skladnik1,Skladnik2,Skladnik3,Skladnik4,
Skladnik5,Skladnik6,Skladnik7);
Kupec7:examples.protocols.KupecAgent2(Skladnik1,Skladnik2,Skladnik3,Skladnik4,
Skladnik5,Skladnik6,Skladnik7)

Tento výraz je treba zadať bez odriadkovania (ja som ho po každom agentovi odriadkoval, z dôvodu prehľadnosti). "-gui" nám spúšta agenta, ktorý monitoruje celú platformu a umožňuje nám ju ovládať pomocou jednoduchého okna. Ďalej nasledujú agenti, ktorých chcem pri štarte JADE vytvoriť.

Po spustení takéhoto MAS, sa nám začne vypisovať (na štandardný výstup) priebeh komunikácie medzi agentami. Tento výpis pri zložitejších príkladoch býva dosť neprehľadný, pretože jednotlivé behaviour som implementoval ako JAVA vlákna (a aj agenti sú JAVA vlákna). Teda, keď je v zdrojovom kóde volanie výpisu System.out.println(text A); a hneď zaním následuje volanie výpisu System.out.println(text B); nedá sa zaručiť, že na výstupe bude po text A následovať text B. Avšak pri detailnejšom študovaní týchto výpisov sa dá určiť ako prebieha komunikácia medzi agentami. Na priloženom CD sa dá nájsť aj príklad takéhoto výpisu. Tým, že sa agenti rozhodujú náhodne, môže mať každý priebeh komunikácie (dialógu) iný priebeh.

Poznamenám ešte, že môžem spúšťať ľubovolný počet agentov, či už to budú agenti skladníci, alebo kupcovia, ale taktiež aj ostatný JADE agenti. Môžem si napríklad vytvoriť tisíce agentov, ktorí sa budú správať ako agenti skladníci a tisíce agentov, ktorí sa budú správať ako agenti kupcovia. Nešlo mi ale o to, aby som mal čo najväčší počet agentov, ale aby som ukázal, že moja pridaná funkcionalita vyjednávania do prostredia JADE je funkčná a dokáže fungovať aj v zložitých MAS.

9. Kapitola : Inštalácia a spustenie MAS

Aby sme mohli aj prakticky vyskúšať, ako fungujú MAS, na priloženom CD sú uložené programy, ktoré to umožňujú. Testovanie programov som robil v operačnom systéme Windows XP. Avšak JADE by malo fungovať vo všetkých systémoch.

9.1 Inštalácia MAS

Najskôr je treba do počítača nainštalovať JAVA Development Kit (JDK) v prípade, ak ešte nie je na počítači nainštalovaný. Na priloženom CD sa nachádza súbor jdk-6u18-windows-i586.exe, jeho spustením sa spustí inštalácia JDK, ďalej treba pokračovať podľa pokynov a JDK bude úspešne nainštalované.

Teraz sa musia skopírovať priečinky ant a jade z CD na pevný disk PC, napríklad do C:\.

Ďalej treba nastaviť štyri systémové premenné:

1. ANT_HOME : hodnota = C:\ant, ak sme skopírovali priečinok ant na iné miesto na disku, treba zadať cestu k tomuto priečinku
2. JAVA_HOME : hodnota = C:\Program files\Java\jdk 1.6.0_18 alebo cesta, kam sme nainštalovali, alebo máme nainštalovanú JDK
3. PATH : hodnota = %ANT_HOME%/bin
4. CLASSPATH : hodnota = CLASSPATH%;.;c:\jade\lib\jade.jar;c:\jade\lib\commons-codec\commons-codec-1.3.jar;C:\jade\classes, ak sme skopírovali priečinok jade na iné niesto na disku, treba namiesto C:\ zadať cestu k tomuto priečinku.

Vo Windows XP sa systémové premenné nastavujú pomocou : Ovládacie panely -> Systém -> Upresniť -> Premenné prostredie -> Nové

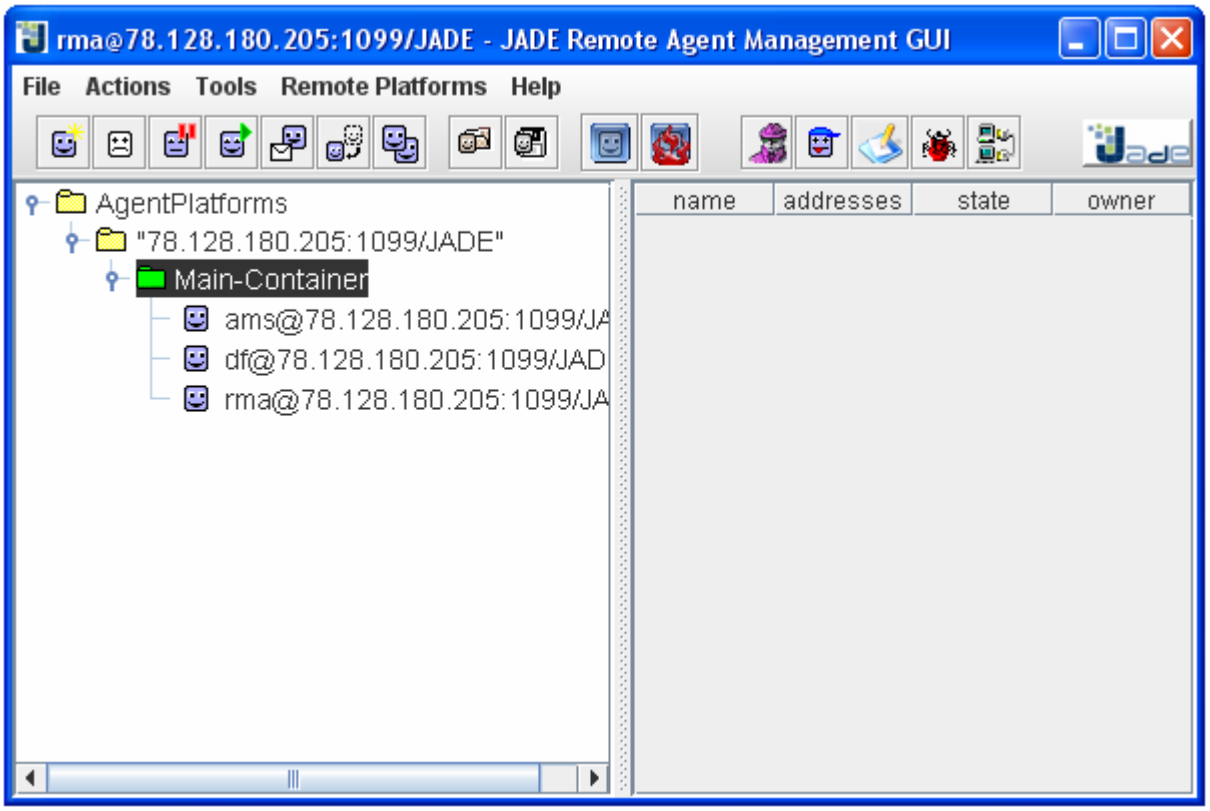
Spustíme príkazový riadok a presunieme sa do priečinku jade. Postupne zadáme do príkazového riadku tri príkazy:

1. ant jade
2. ant lib
3. ant examples

Teraz by už JADE malo byť nainštalované. Ešte pripomeniem, že táto JADE verzia je už doplnená o VETO protokol a NEGO protokol (pre prácu s VETO protokolom slúžia triedy VetoInitiator, VetoRasponder a VetoResponder2 a pre prácu s NEGO protokolom máme NegoInitiator a NegoResponder). Ďalej štruktúra ACL správy už pozná ďalšie výkonné parametre (JUSTIFY, CHALLENGE, RETRACT, VETO) a sú v tejto verzii obsiahnuté aj nejaké príklady na používanie VETO protokolu aj NEGO protokolu. Príklady sú triedy agentov SkaldnikAgent2, SkladnikAgent3, KupecAgent2 a KupecAgent3.

9.2 Spustenie MAS

Na spustenie JADE nám už stačí len jednoducho zadať do príkazového riadku príkaz : java jade.Boot –gui. Týmto príkazom sa nám spustí agentná platforma s hlavným kontajnerom, ktorý obsahuje troch agentov: AMS, DF a RMA. Zároveň sa nám objaví GUI RMA agenta, znázornený na Obrázku 5.1. Odteraz môžeme všetko riadiť cez toto GUI.



Obrázok 5.1

Napríklad na vyskúšanie nášho príkladu so skladníkom a kupcom si musíme najskôr vytvoriť agenta skladníka. Toho vytvoríme nasledujúcim spôsobom. Označíme si kontajner, v ktorom chceme agenta vytvoriť (napríklad Main-Container). Na vrchnej lište vyberieme Actions -> Start New Agent. Objaví sa nám okno ako na Obrázku 5.2.

Obrázok 5.2

Ako Agent Name zadáme meno agenta, napríklad Skladnik. Na zadanie Class Name klikneme na tlačítko "...". Objaví sa nám zoznam dostupných tried agentov. Vyberieme examples.protocols.SkladnikAgent2. Položky Arguments a Owner nevypĺňame. Položka Container by mala byť vyplnená automaticky. Klikneme na OK a vytvorí sa nám agent Skladnik. Na príkazovom riadku by sa mala objaviť hláška od Skladnika: Skladnik: waiting for request... . Týmto nám agent Skladnik dáva vedieť, že čaká na správu typu Request od iného agenta. Tak vytvoríme agenta Kupec, ktorý začne komunikovať so Skladnikom. Tak ako predtým označíme kontajner, v ktorom chceme agenta vytvoriť a dáme Actions -> Start New Agent. Meno agenta môžeme zadať Kupec. Class name dáme examples.protocols.KupecAgent2. Do položky Arguments zadáme argument, čo je meno agenta, s ktorým chceme, aby komunikoval, teda v našom prípade Skladnik. Ak by sa však agent Skladnik nenachádzal v platforme, v ktorej vytvárame Kupca, musíme zadať celé AID agenta (AID má formu meno@meno_platformy). Klikneme na OK. Tým sa vytvorí agent Kupec a okamžite začne komunikovať s agentom Skladnik. Priebeh komunikácie sa nám zobrazuje na príkazovom riadku. Keď nejaký agent píše oznam na výstup, najskôr zadá

svoje meno a ":". Ďalej agenti vypisujú, aký typ správy dostali, aký je obsah... V týchto výpisoch agenti niekedy uvádzajú svoje celé AID.

Avšak pre docielenie toho, aby komunikácia viacerých agentov bola súčasná, môžeme využiť argument pre agentov pri spúšťaní JADE. Pretože inak sa stáva, že kým naklikám ďalšieho agenta, tak ten predošlý už dávno ukončil svoju konverzáciu. Pri spúšťaní JADE pomocou "java jade.Boot –gui", dáme za tento príkaz medzeru a píšeme agentov, ktorých chceme spustiť, v tvare <meno_agenta>:<trieda_agenta>. Trieda agenta musí byť zadaná aj so svojou cestou. Agentov oddelujeme pomocou ";" a nepoužívame žiadne medzery. Napríklad pre spustenie dvoch kupujúcich agentov (odvodených z triedy KupecAgent2) a dvoch skladníkov (odvodených z triedy SkladnikAgent2) zadáme ako argument pre agentov výraz : Skladnik:examples.protocols.SkladnikAgent2;Skladnik2:examples.protocols.SkladnikAgent2;Kupec:examples.protocols.KupecAgent2(Skladnik,Skladnik2);Kupec2:examples.protocols.KupecAgent2(Skladnik,Skladnik2)
Týmto príkazom sa spustí JADE a hneď sa vytvoria naši štyria agenti, ktorí začnú medzi sebou komunikovať zároveň.

10. Kapitola : Záver

Jednou zo základných úloh výpočtovej logiky pri riešení problému je popísanie tohto problému. V MAS je to špeciálne ako popísať agentov. Ja som zvolil na popis agentov logiku LORA. Je to logika, ktorá je veľmi vhodná na popisovanie racionálnych agentov. Ukázal som syntax a sémantiku tejto logiky. Potom ako som ukázal logiku na popis agentov, som sa zameral na komunikáciu medzi agentami. V komunikácii medzi agentami je jednou z najdôležitejších schopností, ktoré by mali agenti mať, schopnosť vyjednávania. Vyjednávanie v MAS prebieha podľa protokolov určených na vyjednávanie. Moja implementácia bola zameraná na prostredie JADE, ktoré, ako som zistil, je určite veľmi dobrým prostredím na implementáciu a testovanie agentov, špeciálne agentov, ktorí vyhovujú FIPA špecifikácii.

To, že sa JADE drží svojich špecifikácií, mu dáva výhodu kompatibility s inými MAS, ktoré sa taktiež držia FIPA špecifikácií. Taktiež, keď nejaký agent z vonkajšieho prostredia bude chcieť komunikovať s JADE agentom, bude presne

vedieť, ako má prebiehať táto komunikácia. Na druhej strane prispôsobenie JADE multi-agentného systému užívateľovým potrebám nie je triviálne. Napríklad, keď som chcel doplniť do ACL jazyka moje nové typy správ, aby som mohol používať Veto protokol, musel som si prejsť zdrojové kódy JADE tried a nájsť triedu a miesto, kam mám tieto nové typy doplniť. Potom som musel znova skompilovať celé JADE aj s touto pozmenenou triedou. Napríklad narozdiel od prostredia AgentBuilder [9], ktoré nám umožňuje zadávať priamo tieto nové typy správ do užívateľského rozhrania. Avšak AgentBuilder sa viac zameriava na to, že vývojár navrhne celý MAS, a taktiež v prípade potreby prepojenia s inými MAS sa o to postará zasa vývojár, takže tento systém nemusí mať pevné špecifikácie. Avšak MAS sú z definície systémy otvorené vonkajšiemu svetu, a preto pri vytváraní MAS by sme mali myslieť aj na vytváranie špecifikácií pre tento systém. Ja si myslím, že zvoliť si prostredie JADE ako svoje vývojárske prostredie a potom ho rozširovať, je dobrá voľba. Totiž už máme špecifikácie pre JADE, a keď doplníme JADE o nejaké prvky, stačí nám tieto prvky doplniť do už existujúcich špecifikácií a nemusíme vytvárať nové špecifikácie.

Avšak JADE neobsahuje žiadne protokoly určené pre vyjednávanie. Preto som uviedol a implementoval nové vyjednávacie protokoly do prostredia JADE. Prvý nový protokol som nazval VETO protokol a je zaujímavý tým, že má v sebe cykly a do 2 rôznych stavov sa môžeme dostať po prijatí rovnakej správy (to sú stavy 1 a 6 podľa Obrázka 5.3). JADE je prostredie, ktoré sa drží FIPA špecifikácií a správy majú štruktúru podľa ACL jazyka. Avšak ACL jazyk nepozná niektoré typy správ, napríklad VETO, RETRACT, JUSTIFY, CHALLENGE, ktoré používam vo svojom protokole. Preto som musel tieto typy do ACL doplniť.

Ukázal som a vysvetlil ako môže vyzerať komunikácia medzi agentami podľa VETO protokolu. Taktiež som impementoval triedy agentov (SkladnikAgent2 a KupecAgent2), ktoré slúžia ako príklady komunikácia agentov podľa VETO protokolu. Pri popise VETO protokolu ako aj pri popise jeho používania som sa snažil ukázať návod, ako pridávať aj ďalšie vyjednávacie protokoly do JADE. Ako príklad toho, som implementoval do JADE ďalší vyjednávací protokol NEGO. Celú funkcionalitu vyjednávania som sa snažil ukázať na zložitejšom MAS (zložitejší preto, lebo niektorí agenti mali viacero rolí – boli iniciátormi aj respondérmi zároveň, a prebiehalo zároveň veľa rôznych dialógov medzi agentami).

Tým sa to ale nekončí. Medzi sebou nemusia komunikovať len agenti, ale medzi sebou môžu komunikovať napríklad spoločenstvá agentov (bližší popis tejto problematiky sa dá nájsť v [10]). Tomuto výskumu sa však pre nedostatok času budem venovať až v mojej ďalšej práci.

Zoznam literatúry

[1] Michael Wooldridge, *An Introduction to Multi Agent Systems*, John Wiley and Sons, Chichester UK,2002

[2] FIPA, http://www.fipa.org/, 2002,

[3] FIPA, FIPA Abstract Architecture Specification , 2002

[4] FIPA, FIPA ACL Message Structure Specification , 2002

[5] FIPA, FIPA SL Content Language Specification , 2002

[6] JADE, http://jade.tilab.com/, 1998

[7] Ulle Endriss, Nicolas Maudet, Fariba Sadri, and Francesca Toni, *Logic-based Agent Communication Protocols*, ACL 2003, LNAI 2922, pp. 91 – 107, 2004

[8] Michael Wooldridge, *Reasoning about Rational Agents*, The MIT Press, London England, 2000

[9] http://www.agentbuilder.com/, 2006

[10] *Organizations have the strategy. They have the data. But can they close those critical gaps to thrive and survive in the information economy?*, Executive Summary, Harvard Business Review, Jún 2010

[11] M. E. Bratman, *Intention, Plans, and Practical Reason,* Harvard University Press: Cambridge, MA, 1987

[12] M. E. Bratman, D. J. Israel, and M. E. Pollack, *Plans and resource-bounded practical reasonin,* Computational Intelligence, 4:349-355, 1988

[13] M. P. Georgeff and A. L. Lansky, *Reactive reasoning and planning,* In Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), pages 677-682, Seattle, WA, 1987

[14] A. S. Rao and M. P. Georgeff. Modelling rational agents within a BDI-architecture, In R. Fikes and E. Sandewall, editors, Proceedings of Knowledge Representation and Reasoning (KR&R-91), pages 473 – 484. Morgan Kaufmann Publishers: San Mateo, CA, April 1991.

[15] Anand S. Rao and Michael P. Georgeff, *Asymmetry Thesis and Side-Effect Problems in Linear-Time and Branching-Time Intention Logics**, Australian Artificial Intelligence Institute, Carlton, Victoria 3053, Australia

[16] J. L. Austin, *How to Do Things With Words,* Oxford University Press: Oxford, England, 1962

[17] JADE, JADE Programming Tutorial, 2010

[18] FIPA, FIPA Agent Management Specification, 2002

[19] http://jade.tilab.com/doc/tutorials/JADEAdmin/jadeArchitecture.html

[20] FIPA, FIPA CCL Content Language Specification , 2002

[21] FIPA, FIPA KIF Content Language Specification, 2002

[22] FIPA, FIPA RDF Content Language Specification , 2002

[23] JADE, JADE Programmer's Guide, 2010

[24] FIPA, FIPA Communicative Act Library Specification, 2002