Vývojová prostředí pro modelování multi-agentních systémů

Development Environment for Multi-Agent Systems Modeling

Bakalářská práce

Jan Šmajcl

Vedoucí bakalářské práce: Ing. Ladislav Beránek, CSc., MBA.

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

2008

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval/-a samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě pedagogickou fakultou elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích dne

Anotace

Tato práce se zabývá problematikou modelování pomocí agentů. Zabývá se srovnáním nejvýznamnějších vývojových prostředí (JADE, MASON, NETLOGO) mezi sebou i v porovnání s OOP. V rámci tohoto srovnání je těmito nástroji implementovaný jednoduchý multiagentní systém. Poté v jednom vybraném prostředí bude vytvořena simulace rozsáhlejšího matematického modelu.

Abstract

The subject of this work is Agent based modeling (ABM). It compares the most significant frameworks such as JADE, MASON and NETLOGO with each other and with Object-oriented programming (OOP). Within this comparison a small multi-agent model is implemented by using these tools. Then there will be a simulation of a larger mathematical model created in one particular framework.

Poděkování

Rád bych poděkoval Ing. Ladislavu Beránekovi, CSc., MBA.

Obsah

1 Úvod

Multiagentní systémy se v současné době stávají jednou z dominantních oblastí výzkumu umělé inteligence. Tato část se postupně odděluje z distribuované umělé inteligence jako samostatná disciplína. Jako taková se opírá o výsledky výzkumu v jiných disciplínách umělé inteligence (tradiční umělá inteligence, decentralizovaná umělá inteligence, distribuovaná inteligence), v oblasti počítačových věd (počítačové vědy, softwarové inženýrství) a o celou řadu dalších disciplín z jiných oblastí (biologie, ekonomie, kognitivní vědy, sociologie, psychologie).

Tato bakalářská práce má za úkol seznámit čtenáře se základními body problematiky modelování pomocí agentů a s tím související problematikou vývojových prostředí pro tvorbu multiagentových simulací. Hlouběji se zaměřuje na prostředí JADE, MASON a NETLOGO. Těmito nástroji vytváříme jednoduchý multiagentní systém a provádíme srovnání. Následně se zabýváme realizací rozsáhlejší simulace matematického modelu.

Práce je dělena na osm kapitol (pokud zahrneme úvod a závěr). Ve druhé kapitole se zabýváme základními body problematiky modelování pomocí agentů. Seznamujeme se se základními pojmy a s jejich vztahem k dané problematice. Ve třetí kapitole si představujeme prostředí pro tvorbu, modelování a simulace. Seznamujeme se s problematikou implementace multiagentových systémů a s některými vývojovými prostředí, která implementaci usnadňují.

V následujících kapitolách se postupně zabýváme třemi vybranými prostředími. Ve čtvrté kapitole je to prostředí JADE (Java Agent DEvelopment Framework). Seznamujeme se s platformou pro agenty, se životním cyklem

agentu, jeho chováním a vzájemnou komunikací. Pomocí tohoto nástroje vytváříme jednoduchou simulaci a rozebíráme specifika implementace. V páté kapitole se do hloubky zabýváme prostředím MASON (Multi-Agent Simulator Of Neighborhoods). Seznamujeme se s jeho architekturou, modelovou a vizualizační vrstvou. Pomocí tohoto nástroje vytváříme jednoduchou simulaci a rozebíráme specifika implementace. V šesté kapitole se do hloubky zabýváme prostředím NETLOGO. Seznamujeme se s realizací modelů v tomto prostředí, s podporou agentů a s vnitřním programovacím jazykem tohoto prostředí. Pomocí tohoto nástroje vytváříme jednoduchou simulaci a rozebíráme specifika implementace.

V sedmé kapitole se soustředíme na realizaci simulace rozsáhlého matematického modelu. Postupně ukazujeme jednotlivé fáze vývoje, kterými je zapotřebí projít při realizaci takové simulace.

Následuje shrnutí celé práce, seznam použité literatury, abecední rejstřík a seznam ilustrací.

2 Problematika modelování pomocí agentů

2.1 Agentní systém

Pojem agentním systém je chápán jako systém skládající se z agentů a prostředí. V následující části se budeme zabývat těmito složkami systému a jejich specifiky.

2.1.1 Agent

Agenta definujeme následovně (převzato z [1]):

Agent je entita zkonstruovaná za účelem kontinuálně a do jisté míry autonomně plnit své cíle v adekvátním prostředí na základě vnímání prostřednictvím senzorů a prováděním akcí prostřednictvím aktuátorů. Agent přitom ovlivňuje podmínky v prostředí tak, aby se přibližoval k plnění cílů.

Agent se vyznačuje následujícími vlastnostmi:

- nezávislý na ostatních agentech
- existuje v prostředí
- s prostředím je propojen pomocí senzorů a aktuátorů

Agenty dělíme dle složitosti vnitřních komponent do několika skupin. Jsou jimi reaktivní, deliberativní, sociální a hybridní architektury.

2.1.2 Reaktivní agent

Reaktivní agent je agent s nejjednodušší architekturou. Jedná se o agent bez plánovacích schopností. Je tvořen pouze sadou paralelních chování, které se aktivují na základě kombinace vnějších vjemů a vnitřního stavu agentu. Agent má pouze určitou formu paměti, kde je uložen jeho aktuální stav.

Nejznámější reaktivní architekturou je architektura subsumpční, kterou popsal roku 1991 protagonista tohoto směru R. Brooks. Agent podle jeho popisu má reagovat pohotově a patřičně na změny prostředí, měl by být robustní (změna prostředí nezpůsobí jeho kolaps) a měl by dokázat v prostředí jednat za účelem dosažení více cílů, které jsou smyslem jeho existence.

2.1.3 Deliberativní agent

Deliberativní agent, někdy nazýván jako uvažující agent, si uchovává symbolickou reprezentaci prostředí a vnitřních stavů, na jejichž základě sestavuje plány pro dosažení svých cílů. Výběr akce u deliberativního agenta probíhá tak, že agent se snaží aplikovat deduktivní pravidla na bázi tvrzení takovým způsobem, aby dosáhl cíle. Cílem je dokázání cílové formule aplikací akčních pravidel. Pro reprezentaci prostředí agent používá modální či temporální logiky.

Nejpoužívanější teorií je teorie BDI (angl. belief, desire, intention). Základními mentálními kategoriemi pro agenta jsou představy o světě (angl. beliefs), tužby nebo též motivace (angl. desire) a záměry směřující k dosažení těchto tužeb (angl. intention). Základy této teorie na filozofické úrovni položil M. Bratman roku 1987 a roku 1991 A. C. Rao a M. P. Georgeff tuto teorii rozšířili o kategorie cílů a plánů a tím položili základy formální teorie BDI deliberativních agentů. Formalizace spočívala ve vymezení mentálních kategorií v jazyce formální logiky pomocí modálních operátorů.

2.1.4 Sociální agent

Sociální agent je takový agent, který rozšiřuje svůj model prostředí, který si vytváří, o modely ostatních agentů. Zejména jsou to adresy, jména a jejich schopnosti, které se používají v případě kooperace vzájemných aktivit.

V případě, že se nejedná o multiagentový systém s centrálním prvkem (agentem), individuální sociální agent musí uchovávat navíc informace o historii předchozích interakcí, jako jsou ceny transakcí, míra kooperativnosti agentů apod. Agenty pro komunikaci využívají vyšší komunikační jazyk.

2.1.5 Hybridní agent

Hybridní agent je takový agent, který obsahuje komponenty pro reaktivitu, deliberativnost i sociální model pro komunikaci na vyšší úrovni. Hybridní architekturu lze dělit na horizontální a vertikální dle způsobu vrstvení. V případě horizontálního vrstvení mají k senzorům a aktuátorům přístup všechny vrstvy agenta. Důležitý je pak řídící mechanismus, který má za úkol správné přidělení zdrojů, aby nedocházelo k narušení racionálního chování agentu. V případě vertikálního vrstvení je se senzory a aktuátory jen jedna vrstva. Data a žádosti proudí z nižších vrstev do vyšších, které je pak delegují na nižší vrstvy pro vykonávání úkolů.

2.1.6 Softwarový agent

V softwarovém inženýrství narůstá potřeba vytvářet systémy, ve kterých se je zapotřebí robustnosti, adaptability, modularizace, autonomie, bezpečnosti a dalších požadavků. Odpovědí na tyto problémy chce být agentově-orientovaného inženýrství.

Nejpoužívanější metodologií tvorby softwaru je dnes objektově-orientované programování. Základním prvkem v tomto případě je objekt, který je charakterizován atributy a metodami. V objektovém programování využíváme dědičnosti a kompozice. Řízení objektu je realizováno vně zapouzdření objektu. Objekty spolu komunikují prostřednictvím posílání zpráv (volání metod).

Potřeba budování decentralizovaných a distribuovaných aplikací vedla k zavedení paradigmatu programování prostřednictvím aktorů. Aktor je autonomní objekt, který kromě zapouzdření atributů a metod zapouzdřuje také své vlákno. Aktory nesdílí své proměnné a proto jsou na sobě nezávislé. Interakce mezi aktory probíhá prostřednictvím posílání zpráv v asynchronním módu (prostřednictvím schránky). Zprávy jsou zpracovávány pomocí FIFO, což může vést k problémům reakce na neaktuální zprávu. I přes určité výhody se modelování pomocí aktorů nerozšířilo mimo akademické a vědecké projekty.

Paradigma agentově-orientovaného softwarového inženýrství se od obou předchozích přístupů liší. Agenty kromě atributů a metod vyžadují možnost definování chování. Agenty nemusejí zapouzdřovat vlákno, ale stále si zachovávají autonomii vůči ostatním agentům. Agenty podobně jako aktory využívají asynchronní komunikaci a to pomocí jazyků založených na teorii komunikačních aktů. Zatímco objekty mohou navzájem volat své metody, u agentů to není samozřejmostí. V objektově-orientovaných aplikacích nehraje prostředí (kromě kompilace nebo interpretace) žádnou roli. Aktor je závislý na aktorovém serverovém systému. V případě agentů sehrává prostředí významnou roli a to nejen pro správu agentů.

2.1.7 Úloha prostředí

V případě agentově orientovaného programování hraje prostředí významnou roli. Pojem prostředí můžeme být použit jednak pro prostředí pro správu agentů a jednak se s ním můžeme setkat v podobě specializovaného agentu.

Prostředí pro správu agentů má za úkol umožnit běh agentů, v případě komunikace zajistit doručení zpráv příslušnému agentu a v případě mobilních agentů má za úkol umožnit migraci takového agentu.

Úloha agentu Prostředí (Environment) je poněkud jiná. Úlohou takového prostředí je udržování reprezentace modelovaného světa. Jednotlivé agenty se mohou dotazovat na tuto podobu, nebo mohou dle určitých pravidel toto prostředí měnit. Například reaktivní agenty mohou pomocí zanechávání určitých značek v prostředí spolu komunikovat a koordinovat svou činnost.

2.2 Interakce mezi agenty

V této podkapitole se budeme zabývat společenstvím agentů. Vysvětlíme si principy koordinace, kooperace a komunikace.

2.2.1 Koordinace

Převzato z [1]:

Koordinace je proces probíhající v multiagentovém společenství, kterým se dosahuje takového propojení jednotlivých komponent v systému, které umožní řešení problému dosažením decentralizace vykonávaných úkolů a někdy i řídícího procesu. Podle toho lze koordinační procesy rozdělit do dvou základních kategorií na procesy s centralizovaným a decentralizovaným řízením.

H. Mintzberg zavádí tři kategorie koordinačních mechanizmů. Jsou jimi vzájemná dohoda, přímý dozor a standardizace. Vzájemná dohoda je koordinační mechanizmus, při kterém není žádný nadřazený prvek (agenty jsou na stejné úrovni) a agenty jí dosahují vzájemnou komunikací. V případě přímého dozoru existuje jeden agent pro centrální řízení procesů. Tento agent

řídí celé společenství a na základě zpětné vazby zasahuje do procesů. V případě standardizace se jedná o koordinaci skupiny agentů pomocí pravidel chování. Mezi koordinační procesy patří reaktivní komunikace a aukce, které si popíšeme blíže.

Reaktivní komunikace je inspirována komunikací některých druhů hmyzu (např. mravenců). Komunikace probíhá prostřednictvím zanechávání stop v prostředí (u mravenců jsou to feromony). Protokol reaktivní komunikace definuje tři základní prvky: značky s časovou platností, funkci (schopnost) vytváření značek a funkci (schopnost) interpretace značek. Prostředím může být nějaká datová struktura nebo speciální agent. Komunikace mezi agenty a prostředím je realizována posíláním zpráv. Výhodou takového modelu koordinace je, že pokud dojde k odstranění nebo přidání agentů, není třeba modifikovat celý systém. Toto je zajímavou vlastností pro tvorbu otevřených systémů.

Dalším základním typem koordinačního mechanizmu je aukce. Základní myšlenkou tohoto mechanizmu je alokace zdrojů na základě ohodnocení nabídkou a poptávkou. Cílem je najít rovnovážnou cenu za dané zdroje. Protokol aukce definuje prvky: zdroje, nabídka, poptávka, aukcionář (nebo faciliátor - nepovinný), pravidla aukce a strategie účastníků. Aukcí existuje několik typů. Například máme anglickou aukci (cena se zvyšuje, dokud je nějaký kupující ochoten koupit), holandskou aukci (cena začíná nadsazená a snižuje se do doby, než ji nějaký kupující přijme), Vickreyova aukce (jednokolová, zdroj je přidělen kupujícímu, který nabídl nejvíce, ale za cenu 2. největší nabídky) nebo oboustrannou aukci (obě strany nabízejí cenu, za jakou jsou ochotni obchodovat, přičemž cílem je uspokojit nejvyšší objem obchodů - používá se například na burzách).

2.2.2 Kooperace

Převzato z [1]:

Kooperace je řízenou formou koordinace s účelovým uspořádáním agentů ve skupině s cílem dosáhnout společného řešení problému nebo konfliktu. Společenství může být řízeno centrálně nebo naprosto decentralizovaně s autonomními prvky řídícími se svou funkcí užitečnosti, která následně ovlivňuje strategie jejich chování. Každý agent má přesně určenou roli, kterou musí plnit a také vztahy s ostatními agenty k tomu, aby bylo dosaženo globálního cíle.

Příklady kooperačních mechanizmů jsou tabulová architektura a kontraktační síť. Tabulová architektura má za úkol zabezpečit koordinování procesů při distribuovaném řešení problémů. Metoda se inspirovala řešením problému skupinou odborníků. Ti sedí v místnosti, dívají se na tabuli, která slouží jako médium pro zaznamenávání mezivýsledků, a podle obsahu tabule přispívají podle své specializace. Proces řešení je paralelní a proto je zapotřebí zapisovatele, který zabezpečuje koordinaci zapisování na tabuli. Tabulová architektura definuje prvky: společné médium (tabule), řídící prvek a prvky doménových poznatků. Řídící prvek je centrálním prvkem architektury.

Kontraktační síť funguje na odlišném principu. Agenty jsou rozděleny do dvou skupin, na řešitelské agenty a na agenty managerské. Řešitelské agenty jsou zdrojově omezené agenty specializované na určitou oblast, zatímco managerský agent je řídícím prvkem v síti, zadává úkoly řešitelským agentům a provádí kontrolu. Řešení problému v kontraktační síti má přesně určený postup a řídí se komunikačním protokolem. Managerský agent specifikuje úkoly, které je třeba řešit a pošle je všem řešitelským agentům. Ty porovnají požadavky se svými schopnostmi a zdroji a v případě možnosti zabezpečení

úkolu odešlou nabídku managerskému agentu. Ten nabídky vyhodnotí a kontrakt přidělí řešitelskému agentu, který učinil nejvýhodnější nabídku.

2.2.3 Komunikace

ACL (Agent communication language) je obvykle nazýván jazyk vyšší úrovně, jenž je určen pro výměnu informací mezi agenty. Tyto jazyky se většinou opírají o teorii řečového aktu, která vychází z analýzy komunikace v přirozeném jazyce. Tato teorie vychází z představy, že komunikace nenese jen informační obsah, ale že komunikující subjekty jejím prostřednictvím něco dělají. Slovesa typu deklaruji nebo požaduji se nazývají performativy (vyžadují, aby se nějaká operace vykonala). Slovesa, která nemohou být použita v takovéto deklarativní podobě nemají charakter řečového aktu.

V teorii řečového aktu lze rozdělit akty na promluvový, nepromluvový a mimopromluvový. V oblasti ACL představuje promluvový akt formulaci elemntární zprávy, nepromluvový akt určuje typ zprávy a mimopromluvový akt zahrnuje požadované změny vnitřních stavů přijímajícího agenta.

Důležité je při komunikaci zabezpečit používání stejné syntaxe (zadefinováním a důsledným dodržováním) a porozumění sémantiky. K tomu, aby agent mohl správně interpretovat obsah zprávy, musí sdílet určitý rámec znalostí - ontologie. Ontologie obsahují množiny specifikací a definic relací určených pro popis jednotlivých problémových oblastí.

Pragmatická stránka komunikace se soustřeďuje především na znalost o tom, s kým komunikovat a jak iniciovat a udržovat vzájemnou komunikaci.

V současné době se rozvijí dva jazykové standarty (KQML a FIPA-ACL), kterými se budeme zabývat dále.

2.2.4 KQML

KQML (Knowledge Query and Manipulation Language) byl navržen jako pokus o jistou formalizaci a standardizaci komunikace a interakcí mezi agenty. Byl vyvinut v rámci projektu Knowledge Sharing Efforts.

Předmětem zájmu jazyka KQML je především podpora pragmatických a částečně též sémantických hledisek komunikace mezi agenty. Jedná se o soubor protokolů, který podporuje aktivity agentů při identifikaci agentů vhodných pro spolupráci, při navazování spojení mezi nimi a při výměně informací mezi nimi. Na syntaktické úrovni není jazyk vázán na použití konkrétního jazyka. Stejně tak na úrovni transportní a komunikační architektury se předpokládá využití libovolných protokolů a služeb.

Základními principy KQML jsou definice malého počtu zpráv realizující předem specifikované komunikační akty (tyto prvky se nazývají performativy) a zavedení speciální třídy agentů, tzv. facilitátorů.

Facilitátor je agent, který poskytuje speciální komunikační služby, jako je například registrace seznamu služeb a seznamu agentů, které tyto služby poskytují. Facilitátor může v podstatě hrát úlohu jistého informačního brokera (zprostředkovatele). Jazyk sice předpokládá existenci facilitátoru, ale nijak jej nespecifikuje. Realizaci ponechává plně na konstruktérovi.

Jazyk KQML tvoří množina performativů. Ta ale není nijak uzavřená, spíše se jedná o jakousi základní knihovnu, ze které konstruktér vychází, některé z nich může vybrat a k těm pak může přidat některé další dle svého uvážení. Performativ určuje formát protokolu a specifikuje typ komunikačního aktu. Často též popisuje očekávaný protokol odpovědi.

Vlastní obsah zprávy je pouze jedním z parametrů performativu a může být vyjádřen v libovolném jazyce. Specifikace pouze vyžaduje možnost určení počátku a konce vlastní zprávy. Nicméně výzkumný projekt KSE zavedl speciální jazyk KIF, který se zpravidla pro tento účel používá.

2.2.5 FIPA-ACL

Sdružení FIPA (založené v roce 1997) navrhlo svůj vlastní jazyk pro komunikaci mezi agenty. Tento jazyk se nazývá FIPA-ACL. Tento jazyk začíná být čím dál více akceptován jako standard.

FIPA-ACL vychází do značné míry z KQML a snaží se odstranit některé jeho nedostatky. Jádrem FIPA-ACL jsou stejně jako u KQML typizované druhy zpráv realizující řečové akty. Nazývají se komunikačními akty (u KQML to byly preformativy). V literatuře se často používá jednoduše pojem zpráva.

FIPA-ACL pracuje s 20 komunikačními akty, které lze rozdělit do pěti skupin: přenos informace, vyžádání informace, vyjednávání, vykonání akce a chybová hlášení. Množinu komunikačních aktů lze chápat jako uzavřenou a pokud definuje konstruktér nový akt, jedná se pouze o akt složený z aktů standardních a s použitím omezeného množství operátorů definovaných ve specifikaci jazyka FIPA-ACL.

Zatímco jazyk KQML předpokládá značnou centralizaci přenosem zpráv prostřednictvím facilitátoru, FIPA umožňuje daleko vetší decentralizaci. Vymezení komunikace na nižší úrovni je přísně odděleno od specifikace ACL. Každý agent má přidělen jednoznační identifikátor GUID. Spolu s dalšími agenty je umístěn v domovské agentové platformě, která mu poskytuje fyzickou infrastrukturu včetně nezbytných služeb jako AMS (agent

management system), DF (directory facilitator) a ACC (agent communication channel). AMS zodpovídá za tvorbu, připojení, rušení, odpojování a registraci agentů v platformě včetně služeb tzv. bílých stránek. DF nabízí seznam agentů s nabídkou jejich aktuálních služeb v rámci komunity, tj. služby tzv. žlutých stránek. Zatímco pro komunikaci v rámci platformy se používá RMI (remote method invocation), pro přenos zpráv mezi platformami se využívá ACC. V tomto FIPA standart převyšuje standard KQML, protože řeší nejenom komunikaci v rámci platformy, ale též se zabývá problematikou, jak zajistit komunikaci mezi agenty různých platforem.

2.3 Využití agentově orientovaného programování

Než se začneme zabývat prostředími pro agentově orientované programování, je zapotřebí se ještě zastavit u otázky využití. Má tato technologie praktické využití? Oblastí nasazení agentově orientovaného programování je celá řada. Zmíníme si dvě nejvýznamnější.

První velkou oblastí je oblast simulací modelů biologických, sociálních či ekonomických jevů. Cílem takové simulace modelu je lepší porozumění dané problematice. Jedná se o modelování odspodu, tj. agenty nedostávají žádné příkazy z hora. Důležitou součástí takového modelu je prostředí (viz. příslušná podkapitola). Interakce mezi agenty probíhají pouze lokálně. Centrem zájmu je chování jednotlivých agentů, které se v čase může vyvíjet. Pojetí času bývá většinou diskrétní, ale nemusí to být pravidlem. Touto oblastí se budeme do hloubky zabývat v dalších částech této práce.

Druhou oblastí, kterou si zde zmíníme, ale nebudeme se jí dále do hloubky zabývat, je tvorba síťových aplikací a jejich využití v prostředí Internetu. Zde se využívá vlastností jako je přímá komunikace mezi agenty

a vyjednávací techniky. Možných nasazení je celá řada, např. v oblasti e-learningu nebo oblasti e-healthcare (zdravotnictví) a hlavně v oblasti e-commerce/e-tradingu. Konkrétním příkladem může být projekt AgentCities, který vznikal mezi lety 2001 až 2003 a který je realizován pomocí JADE.

3 Prostředí pro tvorbu, modelování a simulaci

3.1 Programovací jazyky pro realizaci multiagentních systémů

Tato část je věnována problematice programovacích jazyků, vhodným pro implementaci multiagentních systémů. Podrobně si popíšeme požadavky kladené na takový programovací jazyk učený k implementaci multiagentového systému a s jednotlivými jazyky se seznámíme.

3.1.1 Požadavky kladené na programovací jazyky

Předtím, než se budeme zabývat jednotlivými jazyky vhodnými pro implementaci, musíme specifikovat, které požadavky by takový jazyk měl splňovat.

Prvním požadavkem je **hardwarová nezávislost a nezávislost na operačním systému**. Je zapotřebí, aby v případě, že vytváříme aplikace s distribuovaným zpracováním dat a s decentralizovaným řízením s otevřenou architekturou, programovací jazyk podporoval tvorbu aplikací, které běží na různých hardwarových platformách a jsou podporovány různými operačními systémy.

Dalším požadavkem je **správa paměti a garbage collection** (vymazání a uvolnění již nepoužívané paměti). V některých programovacích jazycích (např. C++) má programátor přímý neomezený přístup k paměti počítače. To má na jedné straně výhodu flexibility, ale na druhé straně to vede k problémům s alokovanou a nevyužívanou pamětí. Proto by se o správu paměti mělo starat prostředí (kompilační nebo běhové).

S pamětí souvisí též problematika **vyloučení směrníkové aritmetiky**. Přesouvání směrníků agentem může ohrožovat server. Proto je nezbytností odstranění směrníkové aritmetiky na úrovni programovacího jazyka alespoň tehdy, nemá-li server procesy pro sledování chráněných oblastí.

Nezbytným požadavkem na bezproblémový běh je **řešení nehod agentů**. Nelze se spoléhat na stoprocentní spolehlivost agentů. V případě, kdy se vyskytne chyba při běhu agenta, je zapotřebí, aby uživatel dostal chybové hlášení a selhání tohoto agenta neovlivnilo běh ostatních agentů. Moderní programovací jazyky tento problém řeší správou výjimek.

Jednou z metod zvýšení bezpečnosti je **interpretace kódu agenta**. Interpretovaný kód je sice pomalejší ve srovnání s kompilovaným, ale u interpretovaných kódů má možnost server kontrolovat funkce a příkazy, která chce agent vykonat. Interpretované kódy mají navíc výhodu větší přenositelnosti mezi různými operačními systémy.

Dynamického vázání umožňuje určování typu objektu až v době jeho vzniku. To umožňuje lepší modularizaci kódu.

S použitím dynamického vázání souvisí i **polymorfní funkce**. Někdy je výhodné využívat proměnné, jejichž typ nedefinoval programátor předem, ale závisí na tom, v jakém kontextu se daná proměnná v kódu objeví.

Objektově-orientované datové struktury můžou najít své místo i v jazyce určeném pro vytváření agentů. Volba objektů a/nebo aktorů závisí na programátorovi a na nárocích na vytvářenou aplikaci.

Výhodou je taktéž **podpora tvorby agentů** na úrovni programovacího jazyka. Tím je myšlena sada předdefinovaných datových struktur, které jsou vhodné k vytváření vlastností typických pro agenty. Tato podpora nebývá

na úrovni samotného programovacího jazyka, ale je součástí vývojových prostředí pro tvorbu multiagentových systémů.

3.1.2 Potenciál jazyků pro agentově-orientované programování

V této části se budeme zabývat programovacími jazyky, které mají potenciál pro tvorbu agentově-orientovaných aplikací.

Java

Nejpoužívanějším jazykem pro agentově-orientované aplikace je programovací jazyk Java. Jazyk má výhody v podobě podpory síťové komunikace, bezpečného kompilačního prostředí a snahy o platformovou nezávislost.

Zdrojový kód programu v jazyce Java je kompilátorem převeden do bajtového kódu, který lze spustit na všech platformách podporujících Javu. Ten lze přenášet pomocí sítě z počítače na počítač. Běhové prostředí (JVM - Java Virtual Machine) na cílovém počítači následně převede bajtový kód do vykonatelné podoby. Tímto způsobem se přenáší sítí například aplety.

Java používá syntaxi podobnou jazykům rodiny C. Java je objektově-orientovaný jazyk. Programátor má ale oproti jazyku C++ (jako objektového zástupce rodiny C) změněný přístup k ovládání některých zdrojů počítače. Java neumožňuje manipulovat se směrníky ani přímý přístup do paměti. Java zabezpečuje automatickou správu paměti, podporu správy vláken a výjimek.

Java umožňuje serializaci objektů a posílání po síti. Tato vlastnost umožňuje vytváření tzv. mobilních agentů. Posílání objektů se provádí pomocí mechanismu RMI (Remote Method Invocation), kterým lze volat metody

vzdálených objektů. Mechanizmem RMI lze přenášet nejen objekty, ale taktéž chování.

Java existuje v několika edicích. Kromě standardního vydání (J2SE) existuje také mikroedice (J2ME), která se specializuje na aplikace pro mobilní zařízení a enterprise edice (J2EE), která je určena pro podnikové aplikace. Na tuto edici Javy se nyní podíváme podrobněji.

Java - J2EE

J2EE je rozšířením standardní edice Javy. J2EE navíc přidává podporu pro tvorbu modulárních a distribuovaných systémů s unifikovaným komunikačním rozhraním. Základními vlastnostmi jsou modularita, otevřená komunikační infrastruktura, kódování zpráv, podpora komunikačních protokolů a podpora registrace a vyhledávání služeb.

Jednotlivé části vytvořené aplikace jsou buď instancí klienta nebo komponentou serveru. Aplikace jsou programovány ve vrstvách od prezentační, přes aplikační logiku až po databázový backend. Všechny tyto vrstvy jsou řízeny serverovým ovládacím prvkem, což ale není v souladu s distribucí funkcí preferovanými v agentním přístupu.

Jednotlivé komponenty v J2EE mohou komunikovat v podobě RMI, RMI-IIOP (rozšíření RMI o komunikaci přes Internet), což je nativní implementace komunikace. Lze však použít i COBRA a další způsoby připojování ke stávajícím systémům. Asynchronní komunikaci zpráv mezi komponentami umožňuje JMS (Java Message Service).

Aplikace vytvořené v jazyce Java standardně komunikují v podobě posílání zpráv. Zprávy lze kódovat i ve formátu XML a Java poskytuje bohaté rozhraní pro posílání zpráv v tomto formátu.

Java podporuje komunikace komponent navzájem a s klienty na mnoha úrovních. Na nejnižší to jsou např. protokoly TCP/IP nebo HTTP, na vyšší úrovni lze využívat služby jako např. JavaMail API. Mimo komunikace distribuovaných objektů prostřednictvím RMI lze využívat i přenos XML zpráv pomocí protokolu SOAP (Simple Object Access Protocol).

Registry komponent jako jsou JNDI nebo JINI jsou spravovány aplikačními servery, které současně poskytují kontejner pro serverové komponenty. UDDI (Universal Directory Definition Interface) poskytuje službu bílých i žlutých stránek ať už pro klienty nebo pro služby poskytující agenty.

Lisp

Lisp je jedním z nejstarších vyšších programovacích jazyků. Jedná se o jazyk určený pro oblast umělé inteligence. Program v jazyce Lisp je množina funkcí, které mají podobu seznamů se jménem a argumenty. Tyto seznamy jsou stromově organizovány, lze je skládat a rozdělovat. Původní verze byla rize funkcionálního charakteru, v současné době jsou k dispozici i verze s podporou objektově-orientovaného programování. Lisp je skriptovací jazyk (interpretovaný), ale v současné době se vyskytují i mutace kompilované. Lisp neumožňuje využívat směrníky a garbage collection provádí systém. Využívá polymorfizmus funkcí a proměnných. Odfiltrování některých funkcí (práce s diskem a pod.) na úrovni zdrojového kódu může ještě zvýšit bezpečnost agentových aplikací.

Python

Na tvorbu agentově-orientovaných aplikací se používá též jazyk Python. Python je skriptovací jazyk (interpretovaný). Od úplného začátku se

jedná o jazyk objektově orientovaný. V tomto jazyce je vše, s čím pracujeme, objektem. Další vlastností jazyku je dynamická změna datových typů. Typ proměnné je určen v okamžiku inicializace nebo přiřazením hodnoty. Pokud následně přiřadíme hodnotu jiného typu, typ proměnné se automaticky změní. Python byl implementován tak, aby jej bylo možno používat téměř na všech dostupných hardwarových platformách a operačních systémech.

Agent0

Agent0 byl prvním pokusem o vytvoření agentově-orientovaného programovacího jazyka. Jednalo se o rozšíření jazyka Lisp. Jazyk byl interpretovaný, měl podporu mentálních kategorií představ o světě, schopností agenta a pravidel závazků. Pokud se schodovala podmínka zprávy s některou z přijatých zpráv a zároveň mentální podmínka s představami agenta, agent provedl nějakou akci.

```
akce ← podmínka zprávy, mentální podmínka.
```

Tcl/Tk

Tcl je skriptovací programovací jazyk, jehož syntax je směsicí csh (c shell) a jazyka C. Používá se jako „glej“ kódu naprogramovaném v jiném programovacím jazyce (např. C, Java). Tcl běží pod operačními systémy Unix, Windows i Mac OS. Podobně jako Lisp se jazyk vyznačuje polymorfizmem proměnných a vyskytují se verze interpretované i s kompilátorem. Ten vytváří bajtový kód, který je překládán do vykonatelné aplikace. Pro tvorbu grafických rozhraní slouží prostředí Tk. Pro tvorbu agentových aplikací se používá verze jazyka s názvem Safe Tcl a Safe Tk. Jazyk Tcl/Tk byl rozšířen o možnost komunikace agentů v jazyce KQML pomocí knihovny TKQML.

Telescript

Telescript vyvinula firma General Magic a jedná se o komerční produkt. Je to objektově-orientovaný interpretovaný programovací jazyk, který je určen pro tvorbu aplikací s mobilními agenty. Ten je tvořen interpretačním prostředím, prostředím pro tvorbu aplikací a samotným programovacím jazykem. Agenty v Telescriptu jsou objekty. Kód agentů lze prostřednictvím API rozhraní zkombinovat s programy napsanými v jiných jazycích.

CLIPS

Pro tvorbu agentů se dají použít i prázdné expertní systémy. Speciálně se to týká pravidlových systémů, které umožňují vytvářet řetězy poznatků v podobě předpoklad důsledek (předpoklad → důsledek). Pravidla mají strukturu vhodnou pro programování modulů reaktivních agentů. Jedním z takových prázdných expertních systémů je i CLIPS. Ten umožňuje využití pravidlové reprezentace modulů v kombinaci s s objektovým modelováním v jazyce COOL. Agent vytvořený v systému CLIPS může reagovat buď na změnu v jeho bázi faktů nebo na přijetí zprávy od jiného agenta.

3.2 Vývojová prostředí pro tvorbu multiagentových systémů

Tato část je věnována vývojovým prostředím pro tvorbu multiagentových systémů. V současné době existuje celá řada komerčních i volně dostupných systémů. Umožňují programování agentů s různou architekturou, multiagentových systémů s podporou meziagentové komunikace, mobilních agentů nebo simulátorů těchto systémů pro podporu modelování umělých nebo reálných systémů. S pěti z nich se seznámíme (JADE, ZEUS, SWARM, MASON a NETLOGO). Poté se podíváme hlouběji

na tři z nich (JADE, MASON a NETLOGO) a na základě sady kritérií je porovnáme.

3.2.1 JADE

JADE (Java Agent DEvelopment Framework) je v Javě vytvořené prostředí pro vývoj a běh peer-to-peer aplikací založených na paradigmatu agentů. JADE vyvinula organizace TILAB. Toto prostředí je kompatibilní se specifikací abstraktní architektury organizace FIPA. Podrobněji se tomuto prostředí budeme zabývat v příslušné kapitole.

3.2.2 ZEUS

ZEUS je vývojové prostředí, implementované v Javě, které bylo vyvinuta na BT. ZEUS vychází podobně jako JADE z architektury FIPA standardu. Zajímavá je jeho podpora ve fázi implementace. Definování ontologií, agentů a jejich chování se provádí pomocí grafického rozhraní. Pro definování pravidel se používá notace shodná s expertním systémem CLIPS. Základní sada pravidel může být rozšířena prostřednictvím v Javě psaných pluginů. Po skončení fáze definování je vygenerován Javový kód agentů, který může být následně ještě doplněn. Po překladu se spouští pomocí vygenerovaných dávkových souborů. Dále se již v této práci tímto prostředím zabývat nebudeme. V případě většího zájmu lze navštívit stránky projektu [17].

3.2.3 SWARM

SWARM je softwarový balík pro milti-agentní simulace komplexních systémů. Vyvinut byl na Santa Fe Institutu. Cílem tohoto prostředí je být užitečným nástrojem pro výzkum v oblasti agentově orientovaného modelování. SWARM je dostupný pro jazyk Objective-C, pro který byl

původně vyvinut a pro jazyk Java, do kterého byl později přenesen. Základem architektury prostředí SWARM je simulace kolekce konkurenčně interagujících agentů. Dále se již v této práci tímto prostředím zabývat nebudeme. V případě většího zájmu lze navštívit stránky projektu [16].

3.2.4 MASON

MASON (Multi-Agent Simulator Of Neighborhoods) je jádro pro simulace s jednoduchými diskrétními událostmi a nástroj pro vizualizaci. MASON je implementován v jazyce Java. Byl vyvinut na George Mason University. MASON obsahuje jak modelové knihovny, tak volitelnou sadu grafických nástrojů pro 2D i 3D. Podrobněji se tomuto prostředí budeme zabývat v příslušné kapitole.

3.2.5 NETLOGO

NETLOGO je multiplatformní multi-agentní programovatelné modelovací prostředí. NETLOGO bylo vyvinuto na Center for Connected Learning. Ač je samotné NETLOGO vytvořeno v jazyce Java, modely se vytváří v jazyce Logo, rozšířeném o podporu agentů. Podrobněji se tomuto prostředí budeme zabývat v příslušné kapitole.

3.2.6 Kritéria pro porovnávání vývojových prostředí

V rámci následujících třech kapitol se postupně do hloubky seznámíme s vývojovými prostředími JADE, MASON a NETLOGO. Nejprve vždy dané prostředí popíšeme a nastíníme, jak se v něm aplikace implementuje a spouští. Poté projdeme postupně seznam porovnávaných vlastností a stručně charakterizujeme, jakou podobu má u daného prostředí. Závěrem v daném prostředí vždy implementujeme srovnávací aplikaci a charakterizujeme

odlišnosti od objektového zpracování a případně výhody daného prostředí při implementaci.

Porovnávané vlastnosti

- Implementační jazyk
- Komunikace mezi agenty
- Možné předměty simulace
- Pojetí času
- Tvorba grafického rozhraní pro aplikaci
- Oddělitelnost modelové a vizualizační vrstvy
- Podpora nastavitelnosti v grafickém rozhraní
- Rychlost provádění simulace
- Debuging

Srovnávací aplikace

Jako srovnávací aplikaci použijeme výukový projekt „taxi-company“ z kapitoly 14 výukových projektů z knihy „Objects First with Java“ autorů David J. Barnes & Michael Kölling. Cílem je hlavně navržení jednoduchého funkčního modelu a vytvoření grafického rozhraní pro zobrazování stavu a polohy agentů (taxi a passenger). V případě, že dané prostředí nabízí nějaké rozšíření bez zdlouhavé implementace, tak toto rozšíření do implementace zahrneme.

4 JADE

JADE (Java Agent DEvelopment Framework) je v Javě vytvořené prostředí pro vývoj a běh peer-to-peer aplikací založených na paradigmatu agentů. Jedná se o middleware, tj. vrstvu mezi operačním systémem a koncovými aplikacemi. JADE je množina rozhraní, která poskytují funkce pro integraci agentů do organizací s využitím J2SE, J2EE i J2ME.

Komunikačním modelem JADE je asynchronní výměna zpráv. Agenty jsou na sobě časově nezávislé. Agenty využívají teorii komunikačních aktů v podobě FIPA ACL standardu.

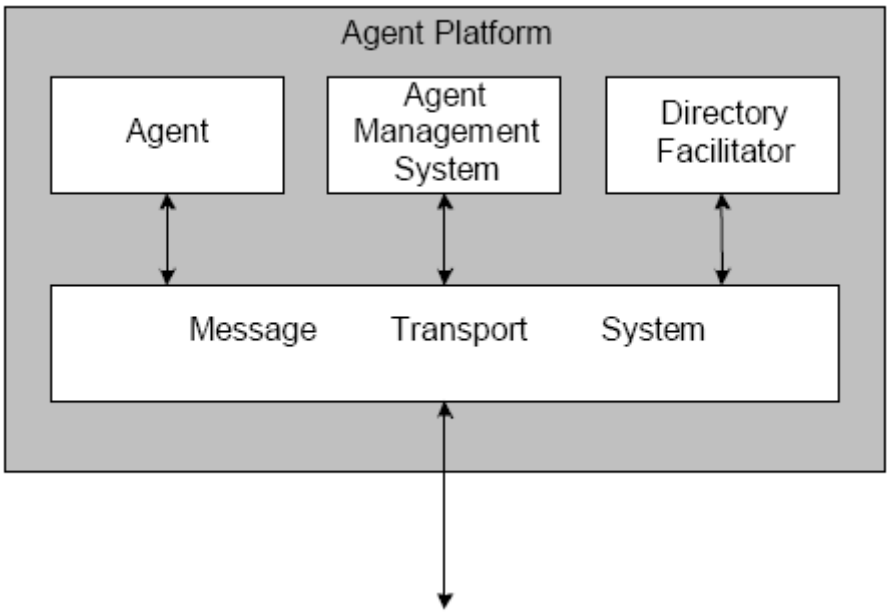
Ač k tomu primárně není JADE určen, my budeme sledovat především jeho využití pro tvorbu simulací.

4.1 Popis prostředí

4.1.1 Platforma pro agenty

Platforma pro agenty tak, jak ji definuje FIPA standard je vidět na ilustraci 1 (převzato z [6]).

Agent Management System (AMS, systém pro správu agentů) je agent, který provádí kontrolu přístupu a užívání platformy. V dané platformě existuje právě jeden AMS. AMS poskytuje službu bílých stránek a služby nezbytné pro životní cyklus agentů, jako je údržba identifikátorů agentů (AID) a jejich stavu. Každý agent se zaregistruje a AMS mu přidělí platné AID.



Ilustrace 1: Referenční architektura FIPA platformy pro agenty

Directory Facilitator (DF, modul žlutých stránek) je agent, který poskytuje službu žlutých stránek. V rámci platformy může být více DF.

Message Transport System, často též nazývaný *Agent Communication Channel* (ACC, agentový komunikační kanál) je softwarová komponenta, která kontroluje výměnu veškerých zpráv v rámci platformy, včetně zpráv pro vzdálené platformy.

JADE je plně v souladu s touto architekturou. Když je JADE spuštěn, okamžitě se vytvoří též AMS a DF a ACC modul je nastaven pro komunikaci pomocí zpráv. Platforma pro agenty může být rozprostřena na několik počítačů.

Každý agent je v podobě vlákna, které zapouzdřuje, řízen kontejnerem pro agenty. Hlavní kontejner je ten kontejner, ve kterém existuje AMS a DF

a kde je vytvořen RMI registr. Ostatní kontejnery jsou na hlavní kontejner napojeny.

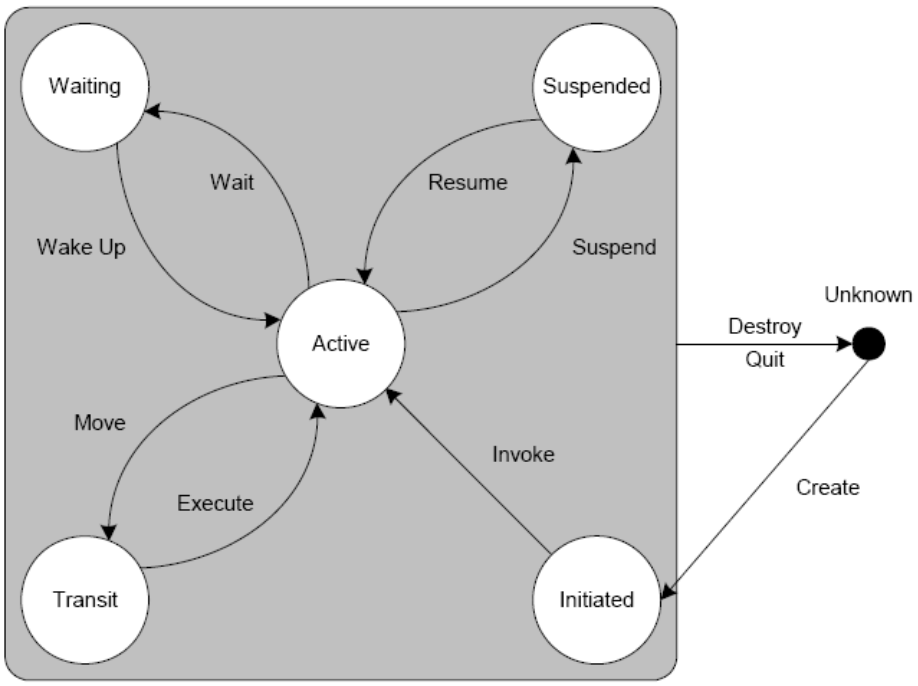
4.1.2 Životní cyklus agentu

Základní třídou pro tvorbu uživatelských agentů je třída *Agent*. Z pohledu programátora se agent vytvoří pomocí dědění od této třídy. Tím dosáhneme i zdědění základních vlastností pro interakci s platformou pro agenty a základní metody pro definování chování agentu.

Každý agent zapouzdřuje vlákno (agent je tedy zároveň aktorem). Funkce agentu by měly být implementovány jako jedno nebo více chování (vysvětlíme si v další podkapitole). Plánovač automaticky plánuje provádění těchto chování.

Agent se může nacházet v jednom z vymezených stavů, které jsou v souladu s Agent Platform Life Cycle FIPA specifikace. Zachyceny jsou na ilustraci 2 (převzato z [6]).

- **INITIATED:** Objekt agentu byl vytvořen, ale ještě nebyl registrován v AMS.
- **ACTIVE**: Objekt agentu je registrován v AMS, má regulární jméno může přistupovat ke všem všem rysům JADE platformy.
- **SUSPENDED**: Objekt agentu je aktuálně pozastaven. Vnitřní vlákno je pozastaveno a neprovádí se žádné chování.
- **WAITING**: Objekt agentu je blokovaný, na něco čeká. Vnitřní vlákno spí na Java monitoru a probudí se, pokud bude splněna nějaká podmínka (typicky pokud přijde nějaká zpráva).



Ilustrace 2: Životní cyklus agentu definovaného FIPA

- **DELETED**: Objekt agentu je definitivně mrtvý. Vnitřní vlákno bylo ukončeno a agent již není registrován v AMS.

- **TRANSIT**: Mobilní agenty se do tohoto stavu dostanou ve chvíli migrace na nové místo. Systém nadále shromažďuje zprávy a ty budou zaslány agentu na jeho nové umístění.

4.1.3 Chování agentu

Agent musí souběžně plnit několik různých úloh a to v závislosti na různých vnějších událostech. Aby bylo řízení agentu efektivní, JADE využívá pro jeden agent pouze jedno vlákno a všechny úlohy, které agent vykonává, se implementují jako objekt třídy *Behaviour* (chování). V rámci

JADE je možno implementovat i vícevláknové agenty, ale není jim poskytována žádná zvláštní podpora, vyjma synchronizace fronty ACL zpráv.

Vývojář, který chce implementovat agentu specifickou úlohu, musí definovat jednu nebo více podtříd třídy *Behaviour*, vytvořit objekt a ten přidat do seznamu prováděných úloh. Plánovač, který je programátorovi skrytý, provádí nepreemptivní plánovací politiku na všechny dostupné chování v rámci fronty. Poté, co chování předá kontrolu zpět, je prováděno chování další. Pokud chování, které se vzdává kontroly, není ještě kompletní, bude naplánováno znovu k provedení v dalším kole. Tuto funkcionalitu zajišťují dvě metody, metoda *action()*, která je metodou provádějící chování touto třídou definované, a metoda *done()*, která zdělí Plánovači (v podobě návratové hodnoty), zda má být chování znovu naplánováno, nebo zda má být definitivně vyřazeno.

4.1.4 Komunikace mezi agenty

Agenty spolu komunikují prostřednictvím posílání ACL zpráv. ACL zpráva je reprezentovaná třídou *ACLMessage*. Třída *ACLMessage* definuje sadu atributů, které jsou schodné s FIPA specifikací.

Pokud chce agent odeslat zprávu, vytvoří nový objet *ACLMessage*, vyplní atributy odpovídajícími hodnotami a zavolá metodu *Agent.send()*. Podobně, pokud chce agent přijmout zprávu, zavolá metodu *receive()*.

JADE taktéž podporuje automatickou tvorbu odpovědí. Dle specifikace FIPA, odpověď musí být formována podle přesných pravidel. V případě, že agent zavolá metodu *createReplay()*, všechny atributy požadované podle specifikace (např. *in-replay-to*) se vyplní automaticky.

4.1.5 Spouštění aplikací vytvořených pomocí JADE

Každá aplikace vytvořená v JADE se skládá z jednoho nebo více agentů. Pokud má aplikace grafické rozhraní, jedná se zpravidla o grafické rozhraní některého z agentů.

Agenty běží prostřednictvím Platformy pro agenty, kterou jsme popsali dříve. Zavádět je lze několika způsoby. Agenty mohou být zavedeny při spouštění platformy administrátorem, prostřednictvím grafického rozhraní *RMA* (Remote Monitoring Agent) nebo prostřednictvím Javového programu (např. jiným agentem). Druhá a třetí varianta je možná pouze v případě, že je platforma spuštěná.

Většinou je používána první a třetí možnost, obzvláště tam, kde JADE neběží nepřetržitě. Nejprve je zaveden JADE a s ním hlavní agent aplikace (často agent představující prostředí) a ten vytváří další agenty, které jsou zapotřebí pro běh komplexní aplikace/systému, ale které již běží nezávisle. Komunikace mezi agenty nadále probíhá pouze pomocí posílání ACL zpráv.

4.2 Porovnání prostředí

4.2.1 Implementační jazyk

Jazykem pro implementaci agentů a chování je jazyk Java. JADE poskytuje sadu tříd, ze kterých programátor vychází (dědí). Jednotlivé agenty se pomocí kompilátoru převedou do bajtového kódu a jsou spouštěny podle postupu popsaného v předchozí části.

4.2.2 Komunikace mezi agenty

Jednotlivé agenty mezi sebou komunikují prostřednictvím posílání ACL zpráv. Komunikace se odehrává v asynchronním módu.

4.2.3 Možné předměty simulace

JADE má své výhody, ale také nevýhody. Výhodou je fakt, že neklade žádné omezení na model (např. simulace v rámci 2D pole). Nevýhodou je právě silná stránka JADE, tj. souběžné provádění jednotlivých agentů.

V simulacích je často kladen důraz na deterministický běh. Je požadováno, aby na základě stejných vstupů (pomineme-li použití generátoru náhodných čísel), jsme získali stejné výstupy. V případě vícevláknové simulace, kde jsou jednotlivé prvky na sobě závislé, toto není možné zaručit. Může se stát (například z důvodu vytížení procesoru), že nějaké ACL zpráva mezi agenty bude odeslána později a v důsledku toho bude ovlivněn průběh celého modelu. Proto je JADE vhodný jen u simulací, kde na tuto vlastnost není kladen důraz, nebo dokonce tam, kde je žádoucí.

4.2.4 Pojetí času

Agenty se provádějí konkurenčně, každý prostřednictvím jednoho vlákna a to kontinuálně. Tato skutečnost ovlivňuje i simulaci jako celek. Pokud tedy chceme zachytit stav simulace v určitých časových krocích, je zapotřebí provádět pravidelně se opakujícího chování, které bude data z modelu shromažďovat a případně zpracovávat. Ani s pomocí takového chování ale nemáme zaručeno opakování ve stejně dlouhých časových úsecích. Jediné, co dokážeme ovlivnit je doba vložení chování do naplánované fronty chování. Přesný čas provedení závisí na plánovači.

4.2.5 Tvorba grafického rozhraní pro aplikaci

JADE nemá žádné doplňkové balíčky pro tvorbu grafického rozhraní. Tvorba grafického rozhraní je plně pod kontrolou programátora a to s využitím balíčků, které jsou součástí knihoven J2SE (Awt, Swing). Grafické rozhraní je zpravidla grafickým rozhraním jednoho z agentů, nejčastěji agentu představující prostředí.

Vláknové pojetí agentů ovlivňuje i proudění dat mezi agentem a jeho grafickým rozhraním. Pokud se data předávají z grafického rozhraní agentu (nebo agentu předáváme povely), děje se tak ve formě přidání nového chování agentu, které data spracuje. Naproti tomu, pokud data předává agent grafickému rozhraní, data se musí zapouzdřit do objektu třídy *Runnable* a ten poté pomocí metody *invokeLate()* nebo metody *invokeAndWait()* zařadit do systémových událostí ke zpracování.

4.2.6 Oddělitelnost modelové a vizualizační vrstvy

Grafické rozhraní není součástí JADE a proto je přirozené, že ani agenty není zapotřebí s grafickým rozhraním vytvářet.

4.2.7 Podpora nastavitelnosti v grafickém rozhraní

JADE nemá žádnou vnitřní podporu grafického rozhraní a tudíž ani podporu nastavování agentů prostřednictvím grafického rozhraní. V případě potřeby může programátor vytvořit grafické rozhraní s podporou nastavování vlastností agentů dle postupu popsaného v části o tvorbě grafického rozhraní (přidáváním dalších chování agentu).

4.2.8 Rychlost provádění simulace

Rychlost provádění simulace je relativně malá. Je způsobena hlavně tím, že každý agent je řízen jedním vláknem. S rostoucím počtem agentů se rychlost značně snižuje. Další věcí, co může průběh simulace ovlivnit, je přítomnost grafického rozhraní u některého z agentů.

4.2.9 Debuging

Vícevláknový charakter JADE může způsobit značné problémy v případě debugingu. Proto JADE obsahuje řadu pomocných agentů s grafickým rozhraním, které mohou ve fázi debugingu pomoci. Za všechny bych zmínil například již dříve popisovaný *RMA*, pomocí kterého můžeme kontrolovat, které agenty máme ve kterém kontejneru, či agent *Sniffer*, který slouží k monitoringu posílání zpráv. Problém, například v případě agentu *Sniffer*, může nastat, pokud bude mít větší množství agentů. V takovém případě bude monitoring značně nákladný a pravděpodobně způsobí více problémů, než užitku. V takovém případě bude nevyhnutelné debugovat pouze s pomocí ladících výpisů do textové konzole.

4.3 Srovnávací aplikace

Implementace simulace pomocí JADE vychází z jeho základních vlastností. Hlavním agentem aplikace je agent reprezentující prostředí (*CityAgent*), který má grafické rozhraní (*CityGUI*). *CityAgent* vytvoří ostatní agenty, které v rámci simulace působí a zavede je do kontejneru platformy.

4.3.1 Třídy projektu

CityAgent je hlavní agent simulace. Zavedením tohoto agentu do platformy se vytvoří grafické rozhraní tohoto agentu (*CityGUI*), dále se vytvoří agent představující společnost provozující taxislužbu (*TaxiCompanyAgent*) a agent představující zdroj pasažérů (*PassengerSourceAgent*). *CityAgent* si udržuje seznam AID jednotlivých agentů spolu s jejich aktuální polohou a stavem. V pravidelných časových úsecích zachytí stav všech agentů do 2D pole znaků a toto pole předá grafickému rozhraní pro vizualizaci.

CityGUI je grafickým rozhraním agentu *CityAgent*. Jeho jediným úkolem je na základě předávaného pole stavů agentů aktualizovat grafickou podobu města s agenty představujícími taxi a pasažéry.

TaxiCompanyAgent je agent představující společnost provozující taxislužbu. Plní několik úloh. Po zavedení agentu do platformy, agent vytvoří agenty představující jednotlivé taxi a zavede je do platformy. Dále se agent registruje u DF (žluté stránky). V průběhu simulace má agent za úkol přijímat objednávky od agentů představující pasažéry a v případě, že jsou volné kapacity (je volný alespoň jeden agent představující taxi), tak obešle jednotlivé agenty představující taxi s požadavkem na zdělení vzdálenosti od agentu představující pasažéra. Na základě odpovědí vybere nejbližší agent a tomu přidělí daný kontrakt.

TaxiAgent je agent představující jedno taxi. Jeho úkolem je na základě žádosti agentu *TaxiCompanyAgent* zdělit vzdálenost od agentu představující pasažéra. V případě, že je agentu *TaxiAgent* přidělen kontrakt v podobě odvozu některého z agentů představujících pasažéra, agent tento úkon vykoná.

PassengerSourceAgent je agent, který má za úkol vytvářet agenty představující pasažéry. V pravidelných intervalech je testováno (na základě generátoru náhodných čísel), zda má být vytvořen nový pasažér, a pokud je podmínka splněna, je vytvořen nový agent představující pasažéra a zaveden do platformy.

PassengerAgent je agent představující pasažéra. Po vytvoření a zavedení do platformy si vyžádá z DF seznam agentů společností provozující taxislužbu a postupně je kontaktuje (v naší podobě simulace je jen jedna), dokud nenalezne společnost, která jej přepraví. V případě, že z celého seznamu agentů nedostane žádnou kladnou odpověď, agent se sám odstraní (pasažér musí hledat alternativní způsob přepravy).

Location je pomocná třída, která představuje polohu v rámci simulovaného města.

Simulation je pomocná třída, která slouží k nastavování vlastností simulace jako celku.

4.3.2 Srovnání

Jednotlivé charakteristické vlastnosti JADE jsme popsali již dříve. Většina se jich projevila i v implementaci srovnávací aplikace.

Jednotliví aktéři simulace (taxi, společnost provozující taxislužbu, pasažéři atd.) jsou reprezentovány pomocí agentů, kteří zapouzdřují vlákno. Každý agent funguje nezávisle na ostatních a s tím souvisí jednotlivé rozdíly v implementaci (v porovnání s klasickým objektovým zpracováním).

Aby bylo možno sledovat simulaci v krocích, byl každému agentu, který jen nereaguje na požadavky ostatních agentů zaveden timer, který

zajišťuje časové prodlevy mezi jednotlivými kroky. To je čas, který je potřebný pro komunikaci s ostatními agenty a pro získání odpovědi. Tato komunikace se odehrává asynchronně prostřednictvím posílání ACL zpráv. Díky zavedení tohoto timeru, který je ale nezbytný pro běh simulací tohoto typu (je jistou formou synchronizace), se průběh simulace značně zpomalil.

Drobným doplňkem, který JADE poskytuje a byl proto využit v rámci srovnávací aplikace je DF. DF poskytuje službu žlutých stránek, což samo o sobě vybízí k jeho využití v této simulaci. Ze skutečnosti víme, že většinou pasažér, který si chce objednat taxi nezná telefonní číslo taxislužby. To se získává ze zlatých stránek (které jsou složeny z bílých a žlutých stránek). Proto se agenty představující společnosti provozující taxislužbu zaregistrují v DF. Agent představující pasažéra získá z DF seznam agentů taxislužby a poté se na některý z nich obrátí.

Další nepříjemnou věcí, kterou je možno při běhu simulace vytvořené pomocí JADE pozorovat, je dočasné zastavení běhu modelu (pravděpodobně způsobeného managementem vláken). Některé (nedá se přesně určit, zda všechny) agenty se zastaví a to bez zjevné chyby (ta by byla ohlášena prostřednictvím konzole a případně by daný agent byl odstraněn). Po chvíli se opět samovolně začnou agenty provádět.

Jak je vidět z těchto pozorování, pokud cíleně nevyhledáváme některé z vlastností JADE, měli by bychom se pro tvorbu simulací matematických modelů tomuto prostředí raději vyhnout.

5 MASON

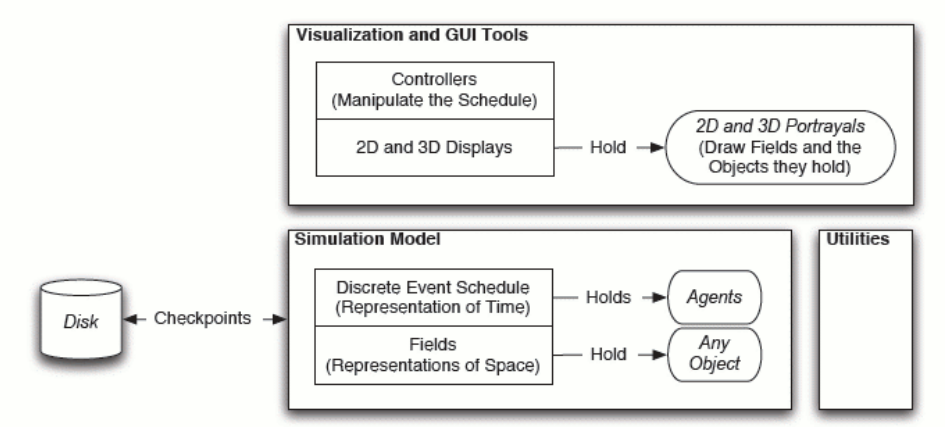
MASON (Multi-Agent Simulator Of Neighborhoods) je jádro pro simulace s jednoduchými diskrétními událostmi a nástroj pro vizualizaci. MASON je implementován v jazyce Java a je navržen tak, aby byl flexibilní pro velkou škálu různých simulací, ale hlavní důraz je kladen na „swarmové“ simulace s velkým množstvím agentů.

5.1 Popis prostředí

5.1.1 Architektura

MASON je vytvořen v modulární, vícevrstevné architektuře, jak je vidět na ilustraci 3 (převzato z [11]). Spodní vrstva je tvořena sadou pomocných datových struktur (utility), které mohou být použity k rozličným účelům. Poté přichází modelová vrstva, která se skládá z krokového plánovače, kvalitního generátoru náhodných čísel a rozmanité sady kolekcí (fileds), které mají za úkol pojmout objekty i s jejich lokací v prostoru.

Vizualizační vrstva umožňuje zobrazování kolekcí (fields) a kontrolu nad simulací. Mezi modelovou a vizualizační částí je jasná dělící čára: sada nástrojů, které umožňují kreslení za běhu a manipulaci s modelem. To nám umožňuje jednat s modelem jako se samostatnou jednotkou. V libovolný okamžik můžeme vytvořit checkpoint (kontrolní bod) stavu modelu, uložit jej na disk, přenést na jinou platformu a tam obnovit běh modelu a k vizualizaci použít i jinou podobu implementace vizualizační vrstvy.



Ilustrace 3: Základní prvky jednotlivých vrstev prostředí MASON

5.1.2 Modelová vrstva

Modelová vrstva je nezávislá na vizualizační vrstvě a může být od ní jednoduše oddělena. Celý model je obsažený v jedné instanci uživatelsky definované podtřídy modelové třídy (*SimState*). Tato instance obsahuje plánovač diskrétních událostí a žádnou nebo více kolekcí (fields).

MASON definuje agent specifickým způsobem a to jako výpočetní jednotku, která může být naplánována k vykonávání nějaké operace a která může manipulovat s prostředím. MASON neplánuje události které mají být prováděny agentem. Raději naplánuje k provádění samotný agent. Pokud je zapotřebí jeden agent naplánovat pro více různých událostí, děje se tak pomocí anonymních obalových (wrapper) tříd.

MASON nepodporuje sub-plánování. Místo toho nabízí sadu nástrojů, které mohou plnit stejnou funkcionalitu. Konkrétně, MASON poskytuje řadu obalů (wrappers), které umožňují seskupování agentů, jejich iterovaní, jejich provádění paralelně v oddělených vláknech atp. Plánovač také umožňuje vytvářet podsekce v rámci jednoho časového kroku.

Kolekce (fields) jsou relacemi libovolných objektů nebo hodnot a jejich lokací v rámci nějakého pomyslného prostoru. Mnoho z těchto kolekcí je jen něco víc, než obal pro jednoduché 2D nebo 3D pole. Jiné nabízejí volné vztahy. Jeden objekt může být zároveň zařazen do více kolekcí (fields). Využití není nijak omezeno a programátor může přidat vlastní kolekce (fields). MASON poskytuje následující kolekce:

- 2D a 3D pole objektů, celých nebo reálných čísel, ohraničené nebo toroidní, s hexagonální, trojúhelníkovou nebo čtvercovou podobou.
- 2D a 3D spoře obsazené objektové mřížky, ohraničené, neohraničené nebo toroidní, s hexagonální, trojúhelníkovou nebo čtvercovou podobou.
- 2D a 3D spoře obsazený spojitý (reálné souřadnice) prostor.
- Orientované sítě (grafy).

5.1.3 Vizualizační vrstva

V případě, že chceme, aby objekty ve vizualizační vrstvě zobrazovaly stav objektů v modelové vrstvě, je zapotřebí použít obal okolo *SimState* v podobě třídy *GUIState*. Jelikož některé objekty ve vizualizační vrstvě potřebují být naplánované, *GUIState* nabízí také vlastní mini-plánovač, který je synchronizovaný s plánovačem modelu. To umožňuje, aby vizualizační vrstva byla naprosto oddělena od modelové vrstvy.

MASON provádí vizualizaci prostřednictvím jednoho nebo více displayů v podobě GUI oken zobrazujících 2D nebo 3D podobu modelových kolekcí. Každý display využívá jeden nebo více zobrazení (portroyal), které

představuje prostředníka, který je zodpovědný za kreslení podoby kolekce a umožňující uživateli provádět inspekci jednotlivých objektů.

MASON nabízí propracovanou grafickou konzoli (*Console*), grafické rozhraní, které usnadňuje uživateli ovládání běhu simulace. Pomocí tlačítek a menu můžeme řídit běh plánovače, ukládat a obnovovat checkpointy, zapínat a vypínat displaye, spouštět různé simulace a používat inspektor. *Console* není podmínkou a může být nahrazena uživatelskou implementací.

Podpora vizualizace grafů není součástí MASON. Předpokládá se, že programátor využije nějaký tradiční produkt jako je JClass Chart nebo PtPlot.

5.1.4 Spouštění aplikací vytvořených pomocí MASON

V této části budeme popisovat způsob spouštění simulace vytvořené v MASON a to prostřednictvím grafické *Console*, tj. simulace s grafickým rozhraním podporovaným přímo MASON. Simulace vytvořené s vlastním rozhraním by se spouštěli v závislosti na tomto rozhraní.

Po spuštění grafického rozhraní *Console* se objeví formulář, který nás vyzývá k výběru simulace. Můžeme buď vybrat nějakou vestavěnou simulaci, nebo do příslušného textboxu vepsat třídu (včetně zařazení v rámci balíčků), která je podtřídou *GUIState*. Poté, když stiskneme volbu „Select“, zobrazí se nám grafická *Console*, ve které je načtena zvolená simulace a otevřou se jednotlivé displaye. Simulace se ovládá prostřednictvím této *Console*.

5.2 Porovnání prostředí

5.2.1 Implementační jazyk

Aplikace vytvářené prostřednictvím MASON jsou plně implementované v jazyce Java. MASON poskytuje sadu tříd, ze kterých programátor vychází (dědí). Simulace se poté spouštějí v závislosti na tom, zda jsou naprogramovány pro standardní *Console*, kterou poskytuje MASON, nebo s vlastním grafickým rozhraním. Ve druhém případě se simulace spouští jako každá jiná aplikace v Javě.

5.2.2 Komunikace mezi agenty

Jednotlivé agenty mezi sebou komunikují, pokud se to dá nazvat komunikací, prostřednictvím volání metod, tj. způsobem používaném v objektovém přístupu.

5.2.3 Možné předměty simulace

Přestože MASON poskytuje celou řadu 2D a 3D kolekcí, samotné simulace tímto faktem nejsou nijak omezeny. MASON využívá diskrétní (krokový) plánovač, čehož většina simulací jen využije. Další výhodou může být i oddělitelnost modelové a vizualizační vrstvy. V případě, že provádíme simulaci pouze na úrovni modelové vrstvy, provádění se značně urychlí.

5.2.4 Pojetí času

MASON využívá diskrétní (krokový) model běhu simulace. Díky tomu nemůže dojít ke znehodnocení dat způsobených managementem nezávislých vláken (jako tomu bylo u JADE). Model lze sice rozdělit do více vláken, ale tato vlákna jsou synchronizovaná vzhledem k času.

5.2.5 Tvorba grafického rozhraní pro aplikaci

MASON nabízí grafické rozhraní v podobě *Console* pro ovládání simulace a displayů, které zobrazují stav simulace. Využití takového rozhraní má své výhody, ale není podmínkou. V případě, že programátor vyžaduje specifické grafické rozhraní, může jej vytvořit plně podle svých představ. Součástí uživatelsky definovaného prostředí musí být i ovládání simulace.

5.2.6 Oddělitelnost modelové a vizualizační vrstvy

Modelová a vizualizační vrstvy jsou plně oddělitelné. Je možno vytvořit i vlastní grafické rozhraní nebo dokonce řídit samotnou simulaci na úrovni textové konzole.

5.2.7 Podpora nastavitelnosti v grafickém rozhraní

MASON podporuje ze standardní grafické *Console* nastavitelnost modelu před spuštěním simulace. V případě, že tuto skutečnost do kódu zavedeme (přidáním jedné metody), automaticky je v rámci *Console* zobrazena záložka *Model*, která zobrazuje nastavitelné proměnné spolu s jejich inicializačními hodnotami. Změněné hodnoty proměnných se projeví jen tehdy, pokud byly nastaveny před spuštěním simulace.

5.2.8 Rychlost provádění simulace

Rychlost provádění simulace je velmi vysoká. Je nejvyšší z porovnávaných prostředí. V případě vynechání vizualizace (například vypnutím displayů) dojde ještě k dalšímu urychlení.

V případě, že máme k dispozici více procesorů či počítačů, je možno simulaci rozdělit do více vláken, která se pak provádějí odděleně. V případě jednoho procesoru by více vláken pouze simulaci brzdilo.

5.2.9 Debuging

MASON neposkytuje žádné speciální nástroje pro debuging. K tomuto účelu se dá využít pouze inspektor, který je součástí grafického rozhraní. Kromě tohoto inspektoru je programátor odkázán na standardní nástroje debugingu jako jsou ladící výpisy.

5.3 Srovnávací aplikace

Implementace simulace pomocí MASON probíhá ve dvou fázích. V první fázi vytvoříme modelovou vrstvu, kde hlavní třídou je třída *City*. Ve druhé fázi vytvoříme vrstvu vizualizační, kde hlavní třídou je třída *CityGUI*.

5.3.1 Třídy projektu

Modelová vrstva

City je třída, která představuje modelovou vrstvu simulace. Tato třída je prostředím (environment), ve kterém se simulace odehrává. Třída *City* poskytuje plánovač celé simulace a řadu kolekcí, ve kterých se udržují agenty a ostatní objekty (např. pasažéři) společně s jejich lokací v rámci města. Třída *City* při inicializaci vytvoří objekt třídy *TaxiCompany* a agent *PassengerSource*.

TaxiCompany je třída, která představuje společnost provozující taxislužbu. Vzhledem k tomu, že *TaxiComplany* nevykonává žádné

naplánované akce, není tato třída považována za agent. *TaxiCompany* ve chvíli inicializace vytvoří určitý počet agentů představující taxi a v průběhu běhu simulace zprostředkovává kontrakty mezi pasažéry a volnými taxi.

Taxi je agent, který představuje jedno taxi. Jeho úkolem je v případě přidělení kontraktu objektem třídy *TaxiCompany* v podobě odvozu některého z pasažérů, tento kontrakt naplnit.

PassengerSource je agent, který má za úkol vytvářet nové pasažéry. V pravidelných intervalech je testováno (na základě generátoru náhodných čísel), zda má být vytvořen nový pasažér, a pokud je podmínka splněna, je vytvořen nový objekt představující pasažéra. V případě, že je volné taxi, passenger je přidán do simulace, v opačném případě zaniká.

Passenger je třída, která představuje jednoho pasažéra. Objekt třídy *Passenger* je pasivním objektem. Jeho úkolem je pouze udržovat stávající lokaci a cílovou lokaci.

Location je pomocná třída, která reprezentuje polohu v rámci simulovaného města.

Vizualizační vrstva

CityGUI je třída, která představuje vizualizační vrstvu. Třída definuje display pro zobrazování stavu simulace a sadu portrayalů (*Net*, *PassengerPortrayal*, *FreeTaxiPortrayal*, *FullTaxiPortrayal*), které určují grafickou podobu jednotlivých zobrazovaných objektů a agentů.

5.3.2 Srovnání

Jak již bylo popsáno, implementace modelů se v MASON provádí ve dvou vrstvách, ve vrstvě modelové a vrstvě vizualizační. V případě implementace srovnávací aplikace tomu nebylo jinak.

Jednotlivý aktéři simulace (taxi, společnost provozující taxislužbu, pasažéři atd.) jsou realizováni pomocí agentů (v podobě definované MASON) nebo objektů a to v závislosti na tom, zda vykonávají pravidelně plánovanou činnost nebo ne.

Na rozdíl od JADE je celý model prováděn pomocí jednoho vlákna. Tato vlastnost chod modelu zrychluje a odstraňuje problémy se synchronizací událostí. Rychlost je možno snížit v rámci vizualizační vrstvy.

MASON přidává, oproti klasickému zpracování, podporu na úrovni vizualizační vrstvy v podobě nastavitelnosti simulace a inspektoru zobrazovaných objektů a agentů. Inspektor je možno používat automaticky. Nastavení aplikace je rozděleno do dvou částí. Nastavení průběhu simulace (rychlost, inicializace generátoru náhodných čísel apod.) prostřednictvím záložky *Console* je automatická, nastavení parametrů modelu prostřednictvím záložky *Model* je zapotřebí implementovat. Tato implementace ale není nijak složitá, jedná se jen o přidání jedné metody pro samotné povolení a, pokud jsme tak již neučinili, přidání přístupových metod k jednotlivým nastavovaným proměnným (get, set).

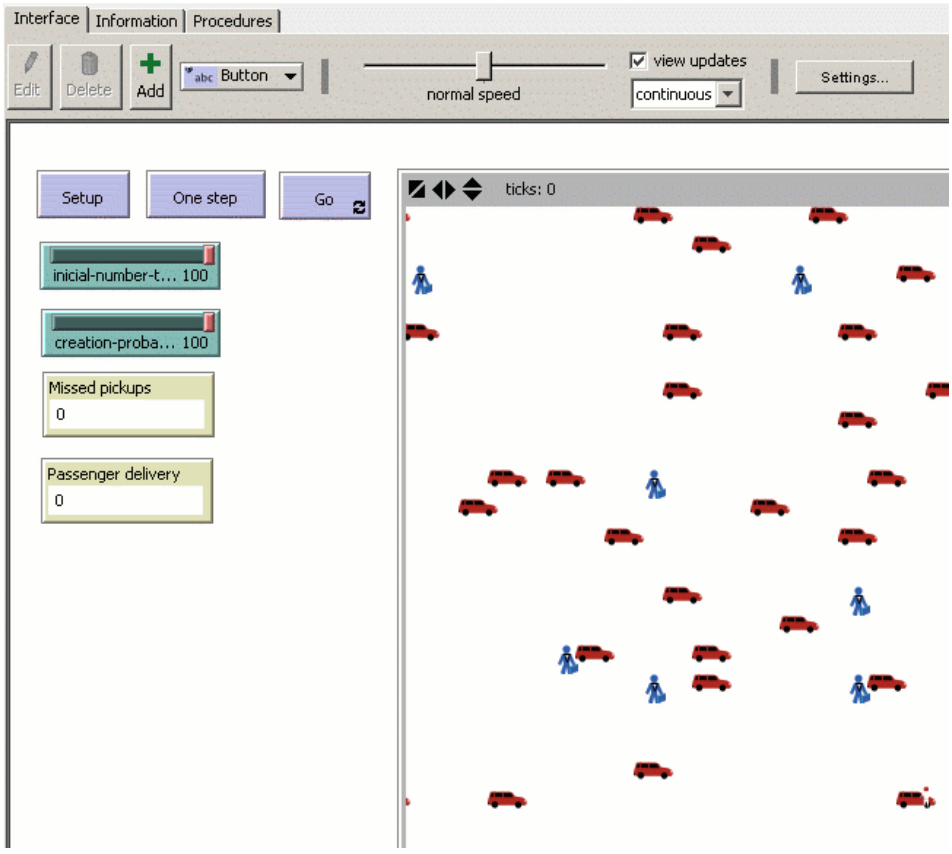
Jak je vidět, MASON má oproti klasickému objektovému zpracování (bez doplňkových tříd) podporu jak na úrovni modelové, tak na úrovni vizualizační. Navíc, ani v případě, že nám nevyhovuje podoba vizualizační

vrstvy, bychom neměli na MASON zanevřít. Modelová vrstva se dá použít jako podklad pod libovolné grafické rozhraní.

V porovnání s JADE je MASON pro simulace výhodnější hlavně z důvodu rychlosti a nepotřebnosti dodatečné synchronizace za běhu. JADE má jistě řadu výhod v oblastech, kde je vyžadována nezávislost agentů, ale ty nejsou předmětem tohoto srovnání.

6 NETLOGO

NETLOGO je multiplatformní multi-agentní programovatelné modelovací prostředí. NETLOGO bylo vyvinuto na Center for Connected Learning. Ač je samotné NETLOGO vytvořeno v jazyce Java, modely se vytváří v jazyce Logo, rozšířeném o podporu agentů.



Ilustrace 4: Podoba modelu v rámci NETLOGO

6.1 Popis prostředí

6.1.1 Simulace v NETLOGO

Simulační model s v NETLOGO skládá z 2D světa obdélníkového tvaru, v rámci kterého mezi sebou interagují agenty. Dále je součástí modelu sada tlačítek, přepínačů, monitorů nebo grafů, které slouží k ovládání nebo sledování doplňkových údajů o průběhu samotné simulace. Samotná simulace je řízena pomocí programu v dialektu jazyka Logo, který byl rozšířen o podporu agentů.

Příklad, jak může model vypadat, vidíme na ilustraci 4.

6.1.2 Agenty

NETLOGO používá čtyři druhy agentů: želvy (turtles), políčka (patches), spojení (links) a pozorovatel (observer). Agenty typu želva jsou agenty, které se pohybují v rámci světa, který je dvoudimenzionální a rozdělené pomocí mřížky na agenty typu políčko. Každým agentem typu políčko je čtvercový kus „země“, přes který se mohou pohybovat agenty typu želva. Spojení jsou agenty, které spojují dva agenty typu želva. Agent pozorovatel nemá žádné umístění.

Po načtení modelu v rámci prostředí NETLOGO, svět neobsahuje žádné agenty typu želva. Vytvořit je můžeme buď prostřednictvím agentu pozorovatele, nebo prostřednictvím agentů typu políčko. Když jsou již vytvořeny agenty typu želva, i ty mohou vytvářet další agenty téhož typu.

Agent pozorovatel a agenty typu políčko se vytvářejí automaticky. Zatímco agent pozorovatel je právě jeden, počet agentů typu políčko je určen

velikostí světa. Svět je obdélníkového vzhledu, ale v základním nastavení má vlastnost toroidu.

Agenty typu spojení mezi jednotlivými agenty typu želva mohou být jednosměrné nebo obousměrné a slouží k tvorbě grafů.

Poloha jednotlivých agentů v rámci světa je určena různě. Zatímco agenty typu políčko mají souřadnice celočíselné a nemohou se měnit, agenty typu želva mají souřadnice reálné a mohou se měnit dle potřeby (v rámci omezení velikostí světa). Poloha agentů typu spojení není určena souřadnicemi, ale dvěma agenty typu želva, mezi kterými je spojení utvořeno. Jak již bylo řečeno výše, agent pozorovatel nemá umístění.

6.1.3 Programovací jazyk

Ovládání agentů NETLOGO provádí pomocí dialektu jazyka Logo. Jedná se o procedurální jazyk, ale procedury jsou prováděny prostřednictvím jednoho ze čtyř typů agentů.

Pokud není procedura volána jinak, je prováděna prostřednictvím agentu pozorovatele (observer). Pokud chceme, aby danou proceduru provedl jeden nebo více agentů, použijeme příkaz *ask* (př. *ask patches* ...). Tím se z univerzální procedury stává metoda objektu (chování agentu).

V případě většiny simulací bychom nevystačili jen s jedním druhem pohyblivých agentů (typu želva). Proto NETLOGO zavádí vlastnost *breed* (rasa). Díky tomu můžeme procedury provádět s agenty jedné konkrétní rasy nebo s agenty typu želva jako celkem.

Další nastavitelnou vlastností je vzhled agentů typu želva (a nejen jich). Výchozí vzhled v podobě šipky by byl pro účely modelování sotva dostačující.

Proto NETLOGO zavádí vlastnost shape (podoba), která určuje podobu daného agentu. Vybírat si můžeme z předdefinovaných vzhledů, nebo si pomocí vestavěného grafického editoru vytvořit vlastní. Bohužel ale není zavedena podpora načtení externích obrázků (pravděpodobně z důvodu, že jednotlivé obrázky jsou specificky definovanými vektorovými grafikami).

6.1.4 Ovládací prvky a procedury jazyka

NETLOGO nabízí podporu pro sadu ovládacích a monitorovacích prvků. Tato podpora má podobu přímého napojení na programovací jazyk.

Například mějme nějaké tlačítko u kterého definujeme, že při stisknutí se zavolá konkrétní procedura. Bez potřeby dalšího programování je tato procedura automaticky zavolána pokaždé, když je dané tlačítko stisknuto.

Podobně fungují například monitory a proměnné. Ve chvíli, kdy vytvoříme nějaký monitor (myšlena zobrazovací komponenta) a zvolíme, jakou proměnnou bude zobrazovat, tento monitor zobrazuje hodnotu proměnné, aniž bychom museli programově nějak nápis na monitoru měnit.

6.1.5 Spouštění aplikací vytvořených pomocí NETLOGO

Hotové modely je možno spouštět dvěma způsoby. První možností je spouštění prostřednictvím samotného prostředí NETLOGO, ve kterém model vytváříme. Druhou možností je spouštění modelu v podobě Apletu.

6.2 Porovnání prostředí

6.2.1 Implementační jazyk

Ač je samotné prostředí NETLOGO naprogramované v jazyce Java, pro tvorbu modelů se využívá dialekt jazyku Logo, který byl rozšířen

o podporu agentů. Pro běh modelu je zapotřebí buď samotné prostředí NETLOGO, nebo přenosná verze pro zobrazení, která se využívá v případě, kdy chceme model distribuovat v podobě Apletu. Ani v tom případě není model vyexportovaný do Javové podoby (jako by tomu bylo například v případě ZEUS). Samotnou prováděcí část Apletu tvoří zmenšená verze prostředí a s tou je distribuovaný soubor s uloženým modelem.

6.2.2 Komunikace mezi agenty

Při provádění procedur není nijak omezeno nakládání s jinými agenty. Agent může zjišťovat stav jiných agentů, požádat jej o provedení některé procedury (*ask*) nebo s jiným agentem přímo nakládat. Do jisté míry se tento postup dá přirovnat k objektovému přístupu (s výjimkou provádění univerzálních procedur jinými agenty).

6.2.3 Možné předměty simulace

Oblast, kterou jsme schopni simulovat, je do jisté míry omezena podobou 2D světa. Na druhou stranu je možno využít svět pouze jako místo, kde agenty existují ale nepohybují se, a samotný průběh modelu odehrávat pouze v rámci programu v podobě procedur.

6.2.4 Pojetí času

Čas je podobně jako v prostředí MASON chápán krokově. Z pravidla bývá reprezentován opakovaným voláním jedné procedury, která provádí události daného kroku. Navíc v určitých případech můžeme agenty požádat o provedení některé procedury souběžně (*ask-concurrent*).

6.2.5 Tvorba grafického rozhraní pro aplikaci

Grafické rozhraní je neoddělitelnou součástí simulace. NETLOGO má podporu jak nastavení vzhledu jednotlivých agentů (v podobě knihovny obrázků), tak podporu různých tlačítek a monitorů. Do jisté míry je to ale i omezením, protože (vyjma nakreslení vlastních obrázků agentů) není možno přidat cokoliv, co není přímo podporováno.

6.2.6 Oddělitelnost modelové a vizualizační vrstvy

Modelovou a vizualizační vrstvu není možno žádným způsobem oddělit, protože vzhled světa je součástí modelu.

6.2.7 Podpora nastavitelnosti v grafickém rozhraní

NETLOGO nabízí celou řadu tlačítek a monitorů. Například nabízí *slider*y pro nastavování celočíselných hodnot proměnných nebo *switch*e pro nastavení logických hodnot proměnných.

6.2.8 Rychlost provádění simulace

V rámci NETLOGO je možno zvolit rychlost provádění simulace. Je možno zvolit rychlost od té nejpomalejší, kde model téměř stojí, až po tu nejrychlejší, kde se model z důvodu urychlení téměř vůbec nepřekresluje. Co do rychlosti může NETLOGO dosáhnout téměř rychlosti dosažitelné pomocí MASON.

6.2.9 Debuging

NETLOGO nenabízí žádnou podporu určenou pro debuging. Na druhou stranu, pro debuging se dá využít řada vestavěných funkcí. Je možno testovat funkčnost jednotlivých procedur prostřednictvím „příkazové řádky“, je možno

provádět simulaci po krocích a je možno provádět specifickým způsobem ladící výpisy. Ty se nedají nechat vypisovat někam do textové konzole. K tomuto účelu můžeme použít nápis (*label*), který se dá zobrazit nad každým agentem, nebo můžeme použít nějaký monitor, který ve fázi, kdy už nebude zapotřebí, odstraníme. Další možností debugingu je použití inspektoru jednotlivých agentů.

6.3 Srovnávací aplikace

Implementace simulace v NETLOGO probíhá v grafickém rozhraní tohoto prostředí, prostřednictvím designeru, ve kterém přidáváme jednotlivé ovládací a zobrazovací prvky a ke kterým vytvoříme procedury v dialektu jazyka Logo.

6.3.1 Rasy (*breed*) agentů projektu

Taxi je první ze dvou ras použitých při tvorbě modelu. Úkolem jednotlivých taxi je odvážet pasažéry, kteří je o to požádají. Pasažér může o odvoz požádat pouze volné taxi.

Passenger je druhou rasou, která je v rámci projektu použita. Passenger představuje pasažéra, který je přepravován prostřednictvím taxi. Přestože se během svého působení v rámci simulace pasažér a teoreticky by se dal reprezentovat pomocí agentu typu políčko (nepohybují se), zvolena byla variantra rasy agentu. Je to z důvodu, že se v jeden okamžik na jednom políčku může nacházet více pasažérů.

6.3.2 Srovnání

Simulace v rámci prostředí NETLOGO se vytváří přímo v tomto prostředí a spouštět se dají pouze jeho prostřednictvím. Při vytváření naší

srovnávací aplikace jsme tedy museli postupovat stejně. Jednotlivé aktéry jsme nadefinovali jako agenty typu želva s grafickým vzhledem omezeným tímto prostředím.

Podobně jako v prostředí MASON, průběh simulace je velice rychlý, čímž se JADE stává nejpomalejším prostředím pro naše účely (simulace matematických modelů).

Vhledem k specifické podobě jazyku, který NETLOGO využívá, není porovnání s objektovým přístupem příliš snadné. NETLOGO využívá jazyk, o kterém by se dalo říci, že je kombinací mezi procedurálním a objektovým. Jedná se o jazyk, kde konkrétní procedury (metody) nejsou zapouzdřeny v rámci jednotlivých objektů (agentů), ale je možno je použít pro libovolný objekt, pokud jsou dostatečně obecné, nebo pro určitou skupinu, pokud tomu tak není.

Při tvorbě srovnávací aplikace jsme plně využili prvky, které NETLOGO nabízí. Proto nebylo nijak obtížné přidání dvou monitorů, které zobrazují počet přepravených a počet zmeškaných pasažérů.

V případě, že si vystačíme s omezeními, které NETLOGO má, je implementace modelu v tomto prostředí nejlehčí. V porovnání s prostředím MASON je pouze o něco pomalejší a proto, pokud nevytváříme opravdu rozsáhlou simulaci u které bude výpočet velmi náročný, zvolíme NETLOGO. V opačném případě (nákladné výpočty) zvolíme prostředí MASON.

7 Simulace rozsáhlého matematického modelu

V rámci této kapitoly se budeme zabývat realizací simulace rozsáhlého matematického modelu. Postupně projdeme jednotlivé fáze vývoje od zadání přes analýzu a design až po běh simulace a zhodnocení. Nedílnou součástí bude i výběr prostředí pro implementaci. Našim hlavním úkolem bude zmapování tohoto procesu a seznámení se se specifiky agentově-orientovaného designu.

7.1 Zadání

První fází celého procesu vývoje jakékoliv aplikace je fáze zadání úlohy. V této fázi zadavatel (někdy jím můžeme být i my sami jako v případě této práce), zadá nějaký úkol, který chce zpracovat (úlohu, kterou chce modelovat). Naším úkolem je získat co nejpřesnější údaje o problematice, aby později nedocházelo k tomu, že budeme muset celou aplikaci upravovat nebo dokonce vytvářet znovu od začátku.

Předmětem simulace bude laboratorní pokus „Rozdělení potravy v laboratorních koloniích mravence Monomorius pharaonis L.“ provedený na Vysoké škole zemědělské v Praze v roce 1990 (popis experimentu viz. [4]).

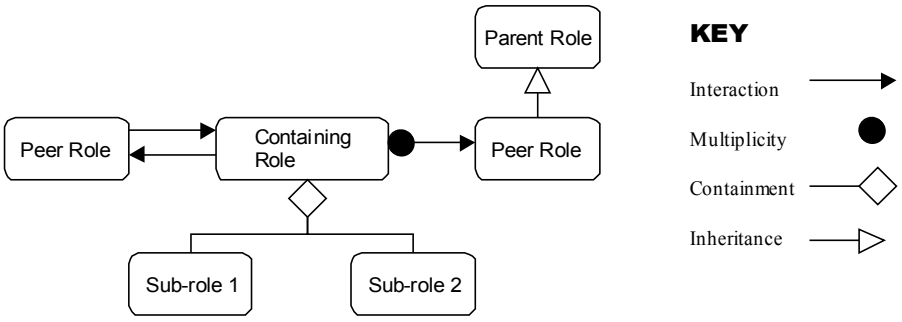
V rámci tohoto experimentu bylo sledováno krmení laboratorní kolonie Monomorium pharaonis L. se zaměřením na rozdělení potravy mezi jednotlivé kasty v hnízdě. V rámci experimentu byly použity dva druhy potravy, med a vařený vaječný žloutek. Smyslem tohoto experimentu bylo zkoumání využití potravy jako způsobu boje s tímto obtížným synantropním škůdcem.

7.2 Analýza problematiky

Další fází, kdy již máme zadání úlohy, je fáze analýzy problematiky. V této části máme za úkol porozumět problematice daného zadání a výběr vhodného modelu.

V této fázi se používá celá řada technik. My si popíšeme a budeme používat techniku modelování prostřednictvím rolí. Tato technika se na aplikaci dívá jako na scénář, který je třeba odehrát. Ve fázi analýzy máme za úkol identifikovat konkrétní scénáře a v nich obsažené role, které budeme v následující fázi přidělovat jednotlivým agentům. Na ilustraci 5 (převzato z [18]) vidíme notaci používanou v rámci UML diagramů typicky používaných v modelování prostřednictvím rolí.

UML diagram modelu rolí vychází z notace UML diagramu tříd. Podstatným rozdílem je pouze předmět jejich zájmu. Zatímco UML diagram tříd zachycuje statické vztahy mezi třídami, UML diagram modelu rolí zachycuje dynamiku interakcí mezi rolemi.



Ilustrace 5: Typicky používaná notace UML diagramů

Dalším krokem analýzy je vytvoření tzv. UML diagramu kolaborací. Tento diagram se soustředí na to, jak jednotlivé role mezi sebou navzájem působí.

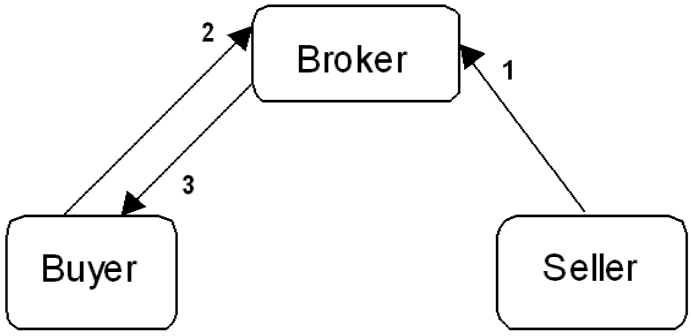
Jediným rozdílem mezi diagramem rolí a diagramem vzájemného působení je skutečnost, že na diagramu vzájemného působení jsou zachyceny jednotlivé interakce s pomocí čísel, které jsou odkazem na popis jednotlivých interakcí.

Následně se jednotlivé role popíší. K tomu slouží speciální tabulky, které zachycují vlastnosti jako název role, vztah role k ostatním, popis, zodpovědnost role, spolupracovníky a další.

Do hloubky popsaných modelů existuje celá řada (obdoba návrhových vzorů v objektovém programování) a proto většinou není potřeba vytvářet modely nové. V případě potřeby se existující modely pouze upraví.

V rámci zadání (a ani v rámci samotného pokusu) nebyl přesně popsán redistribuční proces mezi jednotlivé kasty v rámci kolonie pozorovaných mravenců. Nezbývá nám tedy, než použít některý již existující model, případně jej upravit a poté porovnat, do jaké míry daný model odpovídá skutečnosti.

Použijeme mírně upravený existující model. Bude jím upravený model oboustranné aukce, který bude sloužit k redistribuci potravy. Budeme předpokládat, že ochota mravence přijímat potravu je závislá na určité vnitřní funkci (v našem případě lineární) představující potřebu mravence přijmout potravu. Dále předpokládáme, že každá dělnice s určitou pravděpodobností obstará díl potravy, který je určen k redistribuci (nebo k vlastní spotřebě, pokud je potřeba dělnice vyšší než potřeba ostatních mravenců).



Ilustrace 6: UML diagram kolaborací námi upraveného modelu aukce

Model aukce definuje následující role: nabízející (Seller), poptávající (Buyer) a zprostředkovatel (Broker). Jednotlivým agentům přidělíme role v následující části.

Na ilustraci 6 vidíme UML diagram kolaborací námi upraveného modelu aukce. Naše podoba modelu užívá pouze tři interakce. Jsou jimi nabídka potravy (1), která je současně symbolickým předáním potravy, poptávka potravy (2) a přidělení potravy (3).

7.3 Návrh jednotlivých agentů

V této fázi se vývojář zaměřuje na identifikaci jednotlivých agentů a přidělení rolí, které vyplynuly z fáze analýzy, těmto agentům.

Zodpovědnost jednotlivých agentů je určena ve dvou úrovních. Jednak je to na sociální úrovni, kde je popsána úloha agentu vůči ostatním agentům v podobě komunikace a pak je to na doménové úrovni, kde je popsána úloha agentu z hlediska chování agentu.

Naše úloha, jak vyplývá ze zadání, definuje jako hlavní typ agentu agent představující mravence. Vzhledem k tomu, že budeme sledovat redistribuci potravy v mezi jednotlivé kasty, bude zapotřebí i jednotlivé agenty rozlišit. Proto si zavedeme následující tři druhy agentů představující mravence: dělnice, královny a plod.

Agent dělnice bude mít, oproti ostatním dvěma druhům, úlohu poněkud rozšířenou (bude navíc obstarávat potravu). Z tohoto hlediska bude hrát v našem modelu jak roli nabízející strany, tak roli poptávky. Ostatní druhy agentů mravenců budou hrát pouze roli strany poptávající.

V rámci modelu aukce je zapotřebí ještě zprostředkovatele. Tím bude v našem případě agent prostředí (environment). Ten bude mít za úkol celou vyhodnocovací proceduru. Vyžádá si od jednotlivých agentů jejich nabídky (představující nabízené množství) a jejich poptávky (v podobě velikosti potřeby). Následně rozhodne o uspokojení agentů s největší potřebou, nebo v případě převisu nabídky o uskladnění přebytků.

Jednotlivé zodpovědnosti dělnice na sociální a doménové úrovni můžeme vidět v následujících tabulkách.

Dělnice - sociální úroveň	
Role	**Zodpovědnost**
Nabídka	Odeslat nabídku v podobě množství zprostředkujícímu agentu
Poptávka	Odeslat poptávku v podobě úrovně potřeby zprostředkujícímu agentu

Tabulka 1: Sociální úroveň agentu dělnice

Dělnice - doménová úroveň	
Role	**Zodpovědnost**
Nabídka	Obstarat potravu pro mraveniště
Poptávka	Zpracovat přidělenou potravu
Poptávka	Evidovat úroveň potřeby potravy
Poptávka	Evidovat množství spotřebované potravy

Tabulka 2: Doménová úroveň agentu dělnice

7.4 Výběr prostředí

S touto fází se v případě vývoje pomocí objektové metodologie většinou nesetkáme. Konkrétní programovací jazyk pro implementaci máme určený zpravidla již před zadáním zakázky. Tak tomu může být i v případě agentově-orientovaného programování, ale nemusí to být pravidlem.

Jak jsme si ukázali v předchozích kapitolách, vývojová prostředí pro modelování multi-agentních systémů mají velice rozdílné vlastnosti a ne vždy je možno jedním prostředím implementovat libovolný model. Proto je často na základě doplňkových specifikací, jako je požadovaná rychlost nebo podoba grafického rozhraní, zapotřebí vybrat konkrétní prostředí pro implementaci.

Pro naši simulaci se pokusíme zajistit co největší rychlost provádění (aby bylo možno pozorovat model z hlediska dlouhého období), vlastní grafické rozhraní (z důvodu vizualizace prostřednictvím grafu) a možnost ukládání naměřených dat do Xml souboru.

Vybírat budeme ze tří námi popisovaných prostředí: JADE, MASON a NETLOGO. Z důvodu ukládání dat do zvláštního souboru (Xml) jsme nuceni vyřadit NETLOGO. To umožňuje ukládat pouze aktuální podobu modelu a navíc v přesně definované podobě. Dalším omezením pro NETLOGO je fakt, že toto prostředí je určeno pro simulace v rámci 2D světa.

Ve výběru tedy zůstávají prostředí JADE a MASON. V případě prostředí MASON by bylo zapotřebí nahradit vizualizační vrstvu vlastní implementací. Podobně tak ani JADE nenabízí předdefinované grafické rozhraní, takže toto kritérium není rozhodující. Rozhodujícím bude tedy faktor času. Průběh simulace, pokud bychom použili JADE, by se několikanásobně (řádově i stonásobky) zpomalil. To si můžeme dovolit v případě krátkého představení modelu, ale je to nemyslitelné v případě dlouhotrvající simulace (jako tomu bude v našem případě). Proto zvolíme prostředí MASON.

7.5 Realizace

Nyní následuje fáze samotné implementace agentů a ostatních tříd. V této fázi dostává aplikace již hmatatelnou podobu a je možno pozorovat první výsledky.

V předchozí části jsme zvolili prostředí MASON a s tím souvisí i samotná realizace modelu. Ta bude probíhat ve dvou fázích. První fází bude implementace modelové vrstvy a ve druhé fázi implementujeme vrstvu vizualizační.

Na úrovni modelové vrstvy máme definované agenty *Ant* (abstraktní třída představující agent mravence), *Worker* (představující dělnici), *Queen* (představující královnu), *Child* (představující plod) a *Environment* (představující prostředí). Tyto agenty vyplynuly z předchozích fází vývoje.

Se samotnými agenty si nevystačíme a budeme muset zavést řadu pomocných tříd, které budou sloužit pro konfiguraci modelu a shromažďování dat. Pro konfiguraci agentů zavedeme třídu *Configuration*. Aby byla daná třída dostatečně univerzální, použijeme metodu klíčů a k nim příslušných hodnot. Klíče jsou vymezeny výčtovým typem *ConfigValues.* Předmět simulace je vymezen výčtem *SimulationSubject*.

Pro statistické účely bude zapotřebí zavést několik dalších tříd. Než tak ale učiníme, zavedeme ještě třídu *Storage*, která bude shromažďovat veškerou potravu, kterou agenty představující dělnice obstarají, ale nebude redistribuována a spotřebována (bude uložena na odkládacích hromádkách v mraveništi). Pro ukládání statistických dat slouží třída *Statistic*, která agreguje data jednotlivých kroků simulace v podobě kolekce objektů třídy *StepStatistic*. Tato třída využívá pro ukládání dat podobný systém, jako třída konfigurační a pro vymezení klíčů využívá výčtový typ *StatisticValues*.

Hlavní třídou vizualizační vrstvy je třída *EnvironmentGUI*, která představuje podtřídu třídy *GUIState*. Vzhledem k potřebě vlastního designu grafického rozhraní, je zapotřebí grafickou konzoli přepracovat. Touto novou grafickou konzolí pouze pro účely této simulace je třída *MainGUI*, která nabízí veškeré potřebné ovládání modelu na grafické úrovni. Další důležitou funkcí grafického rozhraní je průběžné překreslování grafu, který zobrazuje stav modelu. K vytvoření grafu použijeme knihovnu PtPlot. Poslední důležitou třídou grafického rozhraní je třída *ConfigForm*, která slouží pro editaci nastavení modelu.

7.6 Nastavení modelu a jeho zhodnocení

Nedílnou součástí implementace matematického modelu je jeho nastavení. Model zpravidla umožňuje nastavitelnost různých parametrů agentů či simulace jako celku a úkolem vývojáře je určité nastavení, které by odpovídalo co nejvíce skutečnosti.

V případě, že modelujeme děje, které jsou zmapované na úrovni agentů (jednotlivých aktérů), máme úlohu usnadněnou. Stačí nám vhodně zvolit měřítko a nastavit model podle něj. V případě, že máme údaje pouze o makro-úrovni, tj. představu, jak by se měl chovat model jako celek a nemáme údaje o jednotlivých složkách, nezbývá, než nastavení v první fázi aproximovat vhodnou funkcí a následně model doladit mírnými úpravami jednotlivých hodnot, aby odpovídal skutečnosti, resp. aby se odlišoval jen do stanovené úrovně. Problém může nastat v případě chybně zvoleného modelu, protože v takovém případě se nemusíme skutečnosti přiblížit nikdy.

Součástí této fáze je samozřejmě zhodnocení věrohodnosti námi zvoleného modelu a jeho odchýlení od reality.

V případě námi zvolené simulace nemáme žádná data na úrovni jednotlivých agentů (mravenců). Máme k dispozici pouze data souhrnná z celého experimentu. Proto nám nezbývá, než nastavení vhodným způsobem aproximovat a poté doladit.

Prvním krokem při takovémto postupu je volba měřítka. Budeme k oběma experimentům přistupovat jednotlivě a to nejen z důvodu odlišnosti povahy těchto experimentů, ale i z důvodu rozdílného naměřeného množství spotřebované potravy. V obou případech budeme množství potravy měřit

aktivitou fosforu v cpm. Námi naměřená data budou představovat množství udávané v této podobě.

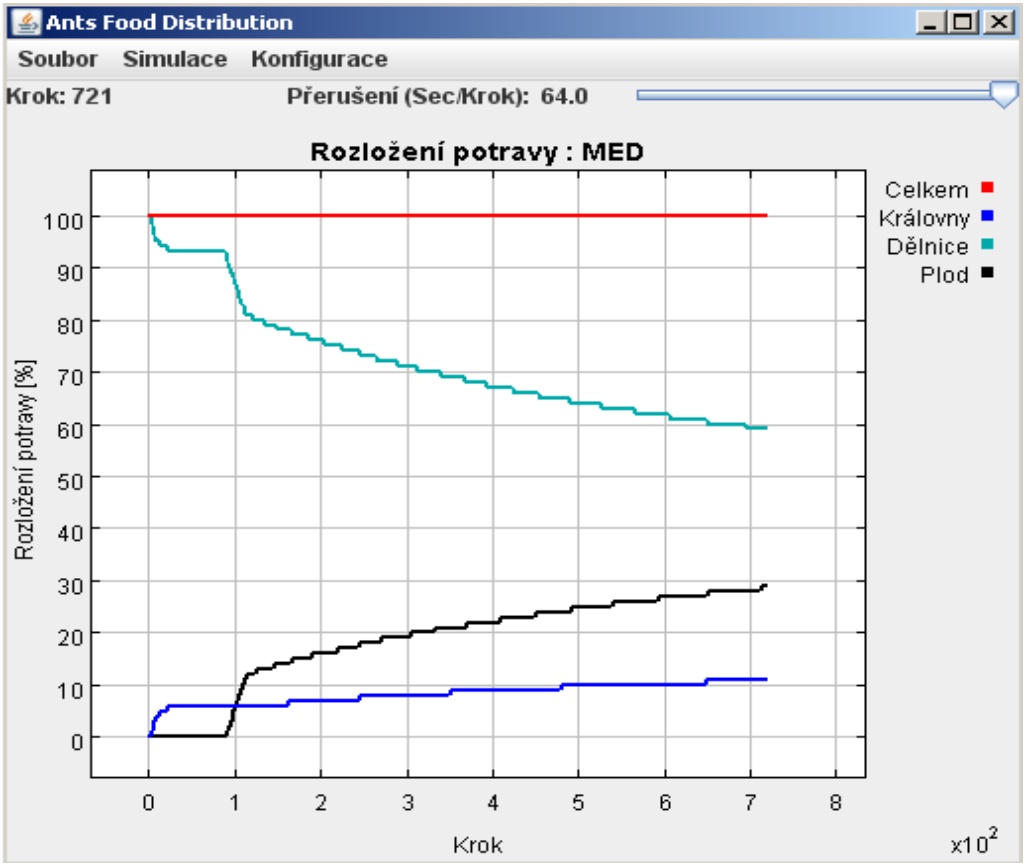
Na základě tohoto požadavku a na základě množství, které jsou schopny dělnice přepravit do mraveniště (předpokládáme, že dělnice v jednom kroku přepraví 1cpm potravy) jsme stanovili měřítko pro experiment s medem 30 kroků představující jednu hodinu. Tímto předpokladem nijak nechceme tvrdit, jakou má dělnice nosnost. Ve skutečnosti to pro náš experiment není důležité. Tímto nastavením pouze určujeme, kolik přibližně přepraví dělnice daného druhu potravy do mraveniště. Jak bude vidět dále, v případě vařeného vaječného žloutku je to několikanásobně více.

Aproximaci provádíme přímkou. Z dlouhodobého hlediska naše podoba modelu aukce vykazuje lineární chování, což je způsobeno lineární povahou nabídky a poptávky. Sklon přímky je určen nárůstem potřeby a jejím uspokojením v případě přijmutí potravy jednotlivými mravenci. Model vykazuje nelineární chování pouze ve chvíli, kdy vstupují do procesu poptávání potravy mravenci, kteří do té doby byly opomíjeni (z důvodu značně nižší potřeby potravy, než je uspokojovaná hladina). Nepatrný vliv na nelineárnost průběhu simulace má taktéž faktor náhody, který byl do modelu zaveden, ale z dlouhodobého hlediska má vliv zanedbatelný.

Po nastavení vypočtených hodnot je bylo zapotřebí dále upravovat, což bylo způsobeno především zmíněním faktorem nelineárnosti v určitých bodech. Naměřené hodnoty z modelu jsou shrnuty v tabulce 3. Údaje v tabulce jsou souhrnné za jednotlivé kasty a v cpm. Za znakem ± je uvedena směrodatná odchylka naměřených hodnot. Naměřená data jsou z deseti různých měření.

Příjem potravy medu			
čas	**dělnice**	**královny**	**plod**
3 hodiny	6362±107	465±9	40±133
24 hodin	9466±14	1797±5	4625±13

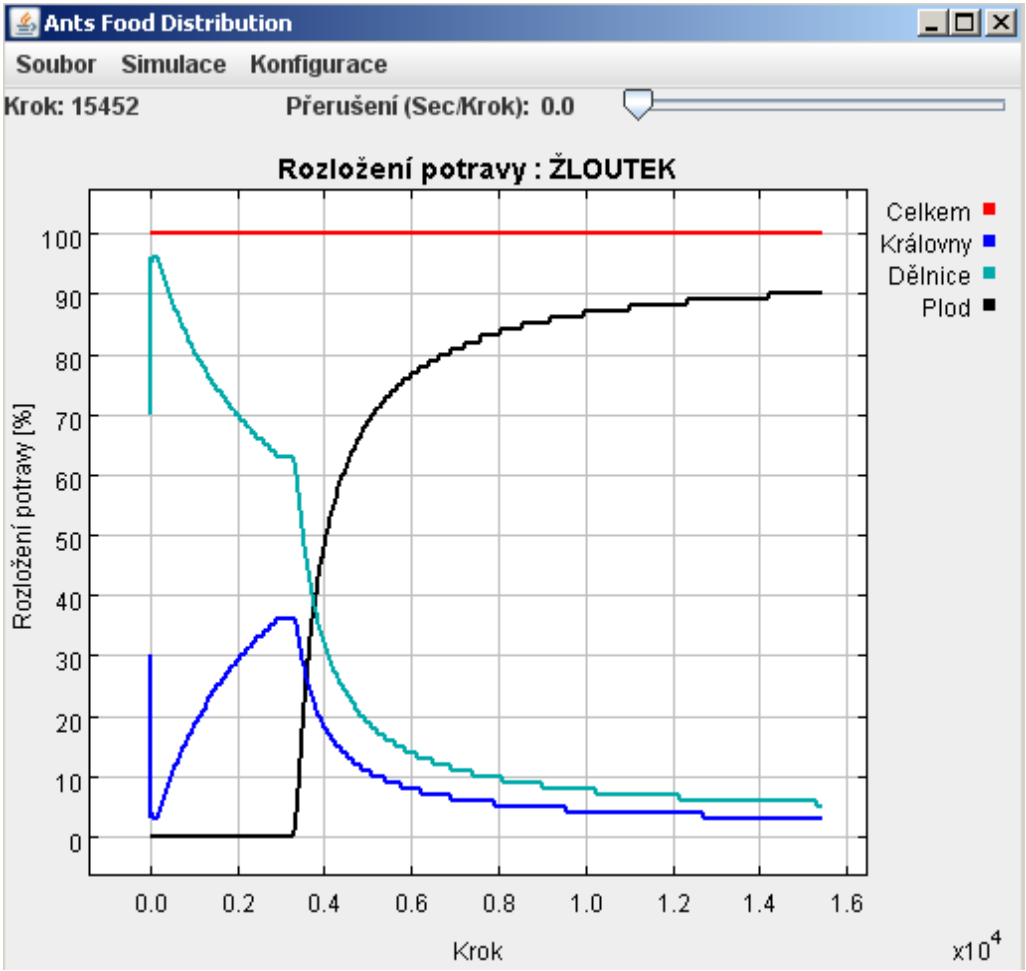
Tabulka 3: Naměřená data z modelu



Ilustrace 7: Graf simulace experimentu s medem

Průběh simulace v podobě grafu je možno vidět na ilustraci 7. Vizualizovaná data jsou za období 721 kroků, tj. přibližně 24 hodiny.

Pro toto nastavení (nastavené jako výchozí pro experiment s medem) náš model odpovídá skutečnosti přibližně z 90% (porovnáváno se střední naměřenou hodnotou naměřených dat v rámci podkladového experimentu).



Ilustrace 8: Graf simulace experimentu s vařeným vaječným žloutkem

Podobným způsobem byly nastavovány výchozí hodnoty i pro pokus s vařeným vaječným žloutkem. V tomto pokusu byl jako časový faktor zvoleno 300 kroků pro jednu hodinu. Je to způsobeno množstvím potravy, kterou jsou schopny dělnice v takové podobě přepravit a jednotliví mravenci přijmout.

Dalším problémem při nastavování jednotlivých hodnot se ukázalo chování společenství v prvních hodinách. V případě reálného experimentu v tomto období nepřijímali mravenci téměř žádnou potravu, zatímco v pozdějším období měli průměrný příjem potravy za hodinu velmi vysoký. Tento rozpor bylo možno řešit dvěma způsoby. Při aproximaci přímkou brát v úvahu všechny tři období, které se ale zdaleka nepodobají přímce, nebo aproximovat pouze období poslední. V našem případě jsme se rozhodli pro druhou variantu. Tím se ale stal model pro počáteční období simulace (několik hodin) naprosto nepoužitelným.

Průběh simulace v podobě grafu je možno vidět na ilustraci 8. Vizualizovaná data jsou za období 15452 kroků, tj. přibližně 51,5 hodiny. Na ilustraci je i vidět, jak se model chová v počátečním období (přibližně 6000 kroků) nerealisticky.

Příjem potravy vařeného vaječného žloutku			
čas	**dělnice**	**královny**	**plod**
24 hodin	20749±42	12163±5	149333±725
48 hodin	29630±43	17538±5	428440±1345

Tabulka 4: Naměřená data z modelu

7.7 Diskuse k provedené simulaci

Jak bylo popsáno v předchozí části, model, který jsme zvolili, může, až na určitá období, popisovat výsledky reálného experimentu s mravenci. Bohužel máme k dispozici pouze data z jednoho takového experimentu a proto nemůžeme tvrdit, že se jedná o model skutečně popisující realitu. Abychom mohli udělat takový závěr, bylo by zapotřebí data naměřit ve více nezávislých experimentech s různými počty mravenců v jednotlivých kastách a poté provést porovnání. Tato část již přesahuje téma této práce.

8 Závěr

V této práci jsme se seznámili s problematikou agentově-orientovaného programování. Zabývali jsme se vývojovými prostředími pro tvorbu, modelování a simulaci multi-agentních systémů. Podrobněji jsme se zaměřili na prostředí JADE, MASON a NETLOGO a provedli srovnání těchto prostředí. V poslední části jsme na realizaci simulace rozsáhlého matematického modelu postupně ukázali jednotlivé fáze, kterými programátor prochází při implementaci takového modelu.

Na jednotlivých prostředích jsme si ukázali nejenom rozdílnosti prostředí, ale také rozdíly v přístupu k problematice modelování pomocí agentů. Ukázali jsme si, že standardizace může mít své výhody (plná podpora agentů), ale i nevýhody (časový faktor). Naproti tomu prostředí, které se multi-agentními systémy spíše jen simulují sice mohou implementaci usnadnit, ale oblast jejich použití je značně omezena.

Naším hlavním předmětem srovnání bylo využití těchto prostředí pro tvorbu simulace matematických modelů. Prostředí MASON se ukázalo jako nejvýhodnější hned v několika sledovaných faktorech. Hlavními přednostmi tohoto prostředí jsou rychlost prováděné simulace, podpora nastavitelnosti agentů v případě standardní grafické vrstvy a oddělitelnost grafické a vizualizační vrstvy. Prostředí JADE ve srovnání příliš neuspělo a to především z důvodu, že se autoři tohoto prostředí zaměřili na plnou podporu standardů a rychlost prováděného modelu byla až na druhém místě. NETLOGO se svou rychlostí blíží výkonu prostředí MASON, ale díky nemožnosti oddělit vizualizační vrstvu od vrstvy modelové a omezenosti světa (2D) je využití tohoto prostředí značně omezeno.

Na realizaci rozsáhlého matematického modelu jsme si ukázali jednotlivé fáze, kterými programátor prochází v případě implementace takového modelu a na úskalí s tím souvisejícími. Do značné míry je postup podobný jako v případě objektově-orientovaného programování, ale v některých fázích se postup liší.

První fází je zadání úlohy. V této fázi nám zadavatel zadává nějakou úlohu a my se snažíme získat pokud možno co nejpřesnější informace. Další fází je fáze analýzy problematiky, kdy se snažíme zadání plně porozumět a vybrat vhodný model. Pokud máme vybraný vhodný model, přecházíme do fáze návrhu jednotlivých agentů. V této fázi identifikujeme jednotlivé agenty a přidělujeme těmto agentům role, které vyplynuly ze zvoleného modelu v předchozí fázi. Ve chvíli, kdy již máme přehled o agentech, které budou v rámci modelu účinkovat, je zapotřebí zvolit prostředí pro implementaci. Při této volbě vycházíme zpravidla z požadavků zadavatele. Následuje fáze samotné realizace, kdy příslušný model implementujeme ve zvoleném prostředí. Před samotným během simulace je ještě zapotřebí model nastavit. V případě, že známe chování jednotlivých agentů, máme úlohu jednodušší. V přídě, že známe pouze chování modelu jako celku, musíme nějakým způsobem chování jednotlivých agentů odhadnout.

Předmětem námi realizované simulace rozsáhlého matematického modelu byla redistribuce potravy v rámci mraveniště mezi jednotlivé kasty mravenců. Testovali jsme, zda lze pospat chování mravenců pomocí modelu oboustranné aukce. Jako rozhodovací faktor jednotlivých mravenců jsme použili potřebu přijímání potravy v podobě vnitřní lineární funkce. Nastavení modelu bylo vztaženo k datům naměřeným v [4]. Ukázalo se, že až na počáteční období (několik hodin) v případě experimentu s vařeným

vaječným žloutkem, náš model dokáže z 90% popsat chování těchto mravenců (porovnáváno se střední naměřenou hodnotou naměřených dat v rámci podkladového experimentu). Vzhledem k omezenému vzorku dat však nemůžeme říci, zda náš model skutečně popisuje realitu. K tomu by bylo zapotřebí dalších reálných pokusů na různých populacích mravenců (s různým složením). Tento experiment však přesahuje rozsah i zaměření této práce.

Reference

[1] KUBÍK, Aleš. *Inteligentní agenty: Tvorba aplikačního softwaru na bázi multiagentových systémů*. Praha: Computer press, 2004. 280s.

[2] MAŘÍK, Vladimír, et al. *Umělá inteligence (3)*. Praha: ACADEMIA, 2001. 328s.

[3] HEROUT, Pavel. *Java a XML*. České Budějovice: Kopp, 2007. 316 s.

[4] FROUZ, J., VLK, F. Rozdělení potravy v laboratorních koloniích mravence Monomorius pharaonis l.. In *Sborník Vysoké školy zemědělské v Praze*. Praha : [s.n.], 1990. s. 143-150.

[5] *Java Agent DEvelopment Framework* [online]. 28.3.2008 [cit. 2008-03-28]. Dostupný z WWW: <http://jade.tilab.com/>.

[6] BELLIFEMINE, Fabio, et al. *JADE PROGRAMMER'S GUIDE* [online]. 18.6.2007 [cit. 2008-03-28]. Dostupný z WWW: <http://jade.tilab.com/dl.php?file=JADE-doc-3.5.zip>.

[7] BELLIFEMINE, Fabio, et al. *JADE ADMINISTRATOR'S GUIDE* [online]. 18.6.2007 [cit. 2008-03-28]. Dostupný z WWW: <http://jade.tilab.com/dl.php?file=JADE-doc-3.5.zip>.

[8] GIOBANNI, Caire. *JADE PROGRAMMING FOR BEGINNERS* [online]. 4.12.2003 [cit. 2008-03-28]. Dostupný z WWW: <http://jade.tilab.com/dl.php?file=JADE-doc-3.5.zip>.

[9] *JADE API Documentation* [online]. 21.6.2007 [cit. 2008-03-28]. Dostupný z WWW: <http://jade.tilab.com/dl.php?file=JADE-doc-3.5.zip>.

[10] *Multi-Agent Simulator Of Neighborhoods* [online]. 28.3.2008 [cit. 2008-03-28]. Dostupný z WWW: <http://www.cs.gmu.edu/~eclab/projects/mason/>.

[11] LUKE, Sean, et al. *MASON: A New Multi-Agent Simulation Toolkit* [online]. 2004 [cit. 2008-03-28]. Dostupný z WWW: <http://www.cs.gmu.edu/~eclab/projects/mason/publications/SwarmFest04.pdf>.

[12] LUKE, Sean, et al. *MASON: A Java Multi-Agent Simulation Library* [online]. 2003 [cit. 2008-03-28]. Dostupný z WWW: <http://www.cs.gmu.edu/~eclab/projects/mason/publications/Agent2003.pdf>.

[13] *MASON Documentation and Tutorials* [online]. 28.3.2008 [cit. 2008-03-28]. Dostupný z WWW: <http://www.cs.gmu.edu/~eclab/projects/mason/docs/#docs>.

[14] *NetLogo* [online]. 28.3.2008 [cit. 2008-03-28]. Dostupný z WWW: <http://ccl.northwestern.edu/netlogo/>.

[15] *NetLogo User Manual* [online]. 5.12.2007 [cit. 2008-03-28]. Dostupný z WWW: <http://ccl.northwestern.edu/netlogo/docs/>.

[16] *Swarm* [online]. 6.3.2007 [cit. 2008-03-28]. Dostupný z WWW: <http://www.swarm.org/wiki/Swarm_main_page>.

[17] *Zeus* [online]. 28.3.2008 [cit. 2008-03-28]. Dostupný z WWW: <http://labs.bt.com/projects/agents/zeus/>.

[18] COLLIS, Jaron, NDUMU, Divine. *The Role Modelling Guide* [online]. 1999 [cit. 2008-03-28]. Dostupný z WWW: <http://sourceforge.net/project/showfiles.php?group_id=6593>.

[19] COLLIS, Jaron, NDUMU, Divine. *The Application Realisation Guide* [online]. 1999 [cit. 2008-03-28]. Dostupný z WWW: <http://sourceforge.net/project/showfiles.php?group_id=6593>.

[20] COLLIS, Jaron. *Case Study 1: FruitMarket* [online]. 1999 [cit. 2008-03-28]. Dostupný z WWW: <http://sourceforge.net/project/showfiles.php?group_id=6593>.

[21] COLLIS, Jaron, NDUMU, Divine. *ZEUS Technical Manual* [online]. 1999 [cit. 2008-03-28]. Dostupný z WWW: <http://sourceforge.net/project/showfiles.php?group_id=6593>.

[22] ZBOŘIL, František ml. *Úvod do agentních a multiagentních systémů* [online]. 2006 [cit. 2008-03-28]. Dostupný z WWW: <http://www.fit.vutbr.cz/~zborilf/study/AGS/AGS01_Uvod.pdf>.

[23] ZBOŘIL, František Ing., Ph.D.. *Agentní a multiagentní systémy: Studijní opora* [online]. 2006 [cit. 2008-03-28]. Dostupný z WWW: <http://perchta.fit.vutbr.cz/vyuka-ags/uploads/3/AGS_opora_m1_ESF.pdf>.

[24] BARNES, David J., KÖLLING, Michael. *Objects First with Java: Book Projects* [online]. 28.3.2008 [cit. 2008-03-28]. Dostupný z WWW: <http://www.bluej.org/objects-first/resources/projects.zip>.

[25] MATTHEW, Neil, STONES, Richard. *Linux Programujeme profesionálně*. Praha: Computer press, 2001. 1080s.

Seznam ilustrací

Seznam tabulek

Výčet příloh - obsah CD

- Text bakalářské práce
  - TextPrace.pdf
- Realizovaná srovnávací aplikace
  - JADE.rar
  - MASON.rar
  - NETLOGO.rar
- Simulace rozsáhlého matematického modelu
  - AntsFoodDistribution.rar
- Instalace prostředí
  - JADE-all-3.5.zip
  - mason.zip
  - NetLogo4.0.2Installer
  - ptplot5.6.zip
  - classpath.bat
  - zeus-2-0-e.zip
  - Swarm-2.2-java.zip