VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MODELOVÁNÍ UVAŽUJÍCÍCH AGENTŮ PETRIHO SÍTĚMI

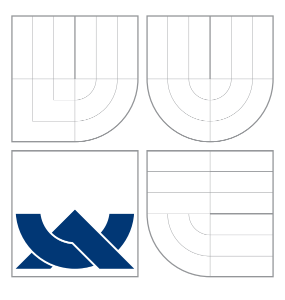
DISERTAČNÍ PRÁCE

PHD THESIS

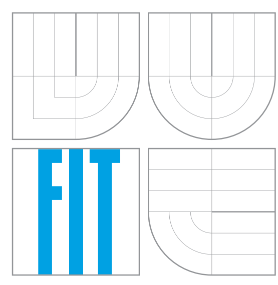
AUTOR PRÁCE — Ing. ZDENĚK MAZAL

AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MODELOVÁNÍ UVAŽUJÍCÍCH AGENTŮ PETRIHO SÍTĚMI

MODELLING DELIBERATIVE AGENTS BY PETRI NETS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE — Ing. ZDENĚK MAZAL
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. FRANTIŠEK V. ZBOŘIL, CSc.
SUPERVISOR

BRNO 2008

Abstrakt

Multiagentním systémem nazýváme systém, ve kterém se nachází několik autonomních prvků, nazývaných agenti, které spolu vzájemně interagují. Agentní a multiagentní systémy představují relativně nový perspektivní vědní obor s řadou aplikací v oblasti distribuovaných softwarových systémů, zejména v prostředí Internetu. V rámci této problematiky je řešena celá řada otázek. Jednou z nich je hledání vhodné vnitřní architektury agentů a její modelování.

Dlouholeté zkušenosti prokázaly, že Petriho sítě představují vhodný nástroj pro modelování paralelních a distribuovaných systémů. Jako takové by měly být vhodné pro modelování agentních a multiagentních systémů, které jsou inherentně paralelní. V současné době však existuje velmi málo systémů, které by umožňovaly modelovat celý multiagentní systém pomocí Petriho sítí – jejich použití je obvykle omezeno pouze na některé podproblémy, jako je verifikace interakčních protokolů nebo synchronizace plánů.

Tato disertační práce se zabývá možnostmi modelování multiagentních systémů pomocí formalismu objektově orientovaných Petriho sítí (OOPN). Oproti standardním vysokoúrovňovým Petriho sítím představují OOPN mnohem dynamičtější nástroj, což je pro modelování agentních systémů velmi důležité. Hlavním přínosem práce je návrh architektury frameworku PNagent, který umožňuje vývoj, testování a běh softwarových agentů v konzistentním grafickém prostředí PNtalk, založeném na formalismu OOPN. Framework je vhodný pro prototypování a experimenty jak s konkrétními multiagentními aplikacemi, tak se samotnou architekturou agentů. Zároveň, díky formální povaze použitých nástrojů, poskytuje dobrý základ pro vysokoúrovňovou verifikaci vlastností multiagentního systému. Návrh architektury byl ověřen prototypovou implementací frameworku, včetně ukázkové aplikace z oblasti řízení mobilních robotů. Součástí práce je také přínos k obecné metodice modelování pomocí formalismu OOPN a nástroje PNtalk.

Klíčová slova

modelování, uvažující agent, BDI architektura, OOPN, PNagent

Abstract

Multi-agent system is a system which consists of several autonomous elements, called agents, which interact with each other. Agent and multi-agent systems present a relatively new and prospective discipline with plenty of applications in distributed systems, especially on the Internet. As a part of it, many problems are being solved. One of them is that of suitable inner agent architecture and its modelling.

Long time experiences have proven that Petri nets are a valuable tool for modelling concurrent and distributed systems. As such, they should be suitable for modelling agent and multi-agent systems, as these are inherently concurrent. Nevertheless, there are not many systems that allow modelling of the whole multi-agent system in Petri nets. The use of Petri nets is usually limited to sub-problems, such as verification of interaction protocols or plan synchronisation.

This Ph.D. thesis deals with the possibilities of modelling multi-agent systems using the formalism of Object Oriented Petri Nets (OOPN). Compared to standard high-level Petri nets, OOPN present a much more dynamic tool, which is essential for modelling agent systems. The main outcome of this work is the design of PNagent framework, which allows for development, testing and running software agents in a consistent graphical environment PNtalk, based on the OOPN formalism. The framework is suitable for prototyping and experiments with both multi-agent applications and the particular agent architecture itself. At the same time, thanks to the formal nature of its underlying paradigm, it provides means for verification of multi-agent system's properties. The design has been verified by a prototype implementation, including a demonstration application from mobile robot control field. An important part of the work is also a contribution to general methodology of modelling using the OOPN formalism and PNtalk tool.

Keywords

Modelling, Deliberative Agent, BDI Architecture, OOPN, PNagent

Modelování uvažujících agentů Petriho sítěmi

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením doc. Ing. Františka V. Zbořila CSc.

. . . .
Zdeněk Mazal
17. září 2008

Poděkování

V průběhu svého studia jsem byl podporován následujícími granty: GAČR GD102/05/H050, CEZ MŠMT MSM0021630528 a FRVŠ MŠMT FR2286/2007/G1. Dále bych chtěl poděkovat svému školiteli doc. Ing. Františku V. Zbořilovi, CSc. za jeho podporu při studiu a svým spolupracovníkům z Ústavu inteligentních systémů, zejména Ing. Vladimíru Janouškovi, Ph.D., Ing. Radkovi Kočímu, Ph.D. a Ing. Františku Zbořilovi, Ph.D. za konzultace a cenné rady. Můj dík patří také rodině, Šárce a všem ostatním, kteří mě po dobu studia jakkoliv podporovali.

© Zdeněk Mazal, 2008.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **4**
1.1 Motivace, cíle disertační práce . . . 5
1.2 Struktura práce . . . 5

2 Objektově orientované Petriho sítě **7**
2.1 Vysokoúrovňové Petriho sítě . . . 8
2.2 Neformální specifikace OOPN . . . 9
2.2.1 Základní prvky sítí v OOPN . . . 9
2.2.2 Struktura neprimitivních objektů . . . 10
2.2.3 Dynamické chování OOPN . . . 11
2.3 Jazyk a systém PNtalk . . . 12
2.4 Navržené rozšíření jazyka PNtalk . . . 13
2.4.1 Řešení v barvených Petriho sítích . . . 14
2.4.2 Řešení v jazyku PNtalk . . . 15
2.4.3 Převod negativních predikátů na existující jazykové struktury . . . 17
2.4.4 Negativní predikáty a logické programování . . . 19

3 Agenti a multiagentní systémy **21**
3.1 Základní pojmy . . . 22
3.1.1 Inteligentní agent . . . 22
3.1.2 Multiagentní systém . . . 22
3.2 Architektury agentů . . . 22
3.2.1 Reaktivní agent . . . 23
3.2.2 Uvažující agent . . . 25
3.2.3 Hybridní agent . . . 27
3.3 Prostředí a interakce v multiagentních systémech . . . 28
3.4 Komunikace v multiagentních systémech . . . 29
3.5 Standardizace agentních technologií – organizace FIPA . . . 30

4 Architektura BDI **32**
4.1 Základní principy a komponenty architektury BDI . . . 32
4.1.1 Představy . . . 32
4.1.2 Touhy . . . 33
4.1.3 Záměry . . . 33
4.1.4 Plány . . . 34
4.2 Formalizace BDI . . . 34
4.2.1 Modální logiky . . . 35
4.2.2 Temporální logiky . . . 35

4.2.3 BDI CTL logika . . . 36
4.3 Realizace architektury BDI . . . 37
4.3.1 Abstraktní interpret BDI agenta . . . 37
4.3.2 IRMA . . . 37
4.3.3 PRS . . . 38
4.3.4 Systémy vycházející z PRS . . . 40
4.3.5 JADEX . . . 41
4.3.6 Další implementace BDI . . . 43
4.4 Formalizace systémů založených na architektuře BDI . . . 43
4.4.1 AgentSpeak(L) . . . 43
4.4.2 Formální specifikace dMars a 3APL . . . 44

5 Modelování BDI agentů objektově orientovanými Petriho sítěmi 45
5.1 Požadavky, cíle, východiska . . . 45
5.2 Přehled frameworku PNagent . . . 46
5.3 Reprezentace báze představ . . . 48
5.3.1 Ukázkový systém – Cleaner world . . . 48
5.3.2 Tradiční způsob reprezentace báze představ . . . 48
5.3.3 Reprezentace báze představ pomocí prostředků OOPN . . . 49
5.4 Plány . . . 51
5.4.1 Přidávání specifické funkčnosti instance plánu . . . 52
5.4.2 Životní cyklus instance plánu . . . 53
5.4.3 Priority plánů . . . 54
5.5 Události . . . 54
5.5.1 Unifikace událostí, spouštění plánů . . . 55
5.5.2 Cíle . . . 57
5.6 Záměry . . . 57
5.7 Interpret . . . 57
5.7.1 Zpracování událostí . . . 58
5.7.2 Interpretace záměrů . . . 59
5.8 Prostředí a meziagentní komunikace . . . 60
5.8.1 Platforma jako zdroj podnětů agenta . . . 60
5.8.2 Služby poskytované platformou . . . 61
5.9 Příklad aplikace frameworku PNagent . . . 62
5.9.1 Neformální zadání úlohy . . . 62
5.9.2 Mobilní roboti . . . 63
5.9.3 Struktura modelu . . . 64
5.9.4 Předzpracování dat a události . . . 65
5.9.5 Báze představ a plány . . . 65
5.9.6 Zhodnocení . . . 70
5.10 Jiné přístupy k modelování multiagentních systémů Petriho sítěmi . . . 72

6 Závěr 74
6.1 Zvolený přístup k řešení . . . 74
6.2 Dosažené výsledky . . . 74
6.3 Možnosti dalšího výzkumu . . . 75

Literatura 75

Přehled publikací autora **82**

Přílohy **83**

A Syntax jazyka PNtalk **84**

B Model Cleaner world **86**

C Model Blocks world **92**

D Návod pro spuštění modelů na přiloženém CD **97**

Kapitola 1

Úvod

Multiagentním systémem nazýváme systém, ve kterém se nachází několik autonomních prvků – nazývaných *agenti* – které spolu interagují. *Autonomií* máme na mysli schopnost dlouhodobě plnit cíle zadané uživatelem bez jeho dohledu. *Interakce* představuje komplexní sociální aktivity, jako kooperace, vyjednávání, apod.

Multiagentní systémy představují relativně nový vědní obor – první práce v této oblasti se datují do osmdesátých let minulého století, do širokého povědomí se multiagentní systémy dostaly v polovině let devadesátých, zejména v důsledku prudkého rozvoje Internetu, který pro ně představuje zřejmě nejdůležitější aplikační doménu. Multiagentní systémy jsou obor velmi interdisciplinární – nejblíže mají k umělé inteligenci, vychází však i z dalších přírodních i společenských věd, jako jsou softwarové inženýrství, robotika, logika, ekonomie, ekologie, sociologie, filozofie a další.

Abstrakce autonomní entity – agenta – nachází uplatnění v mnoha aplikačních oblastech. Jako příklad jmenujme komplexní softwarové systémy, kde jednotlivé komponenty jsou natolik nezávislé a složité, že z hlediska návrhu je efektivní považovat je za zcela samostatné jednotky, které spolu komunikují ve velmi malé míře pomocí vysokoúrovňového jazyka, například jen v případě nestandardních situací [SBD+00]. Dalšími oblastmi jsou týmová spolupráce robotů (např. dobře známý robotický fotbal [Kit98]), telekomunikační aplikace (mobilní agenti, kteří migrují po síti za daty, což vede k minimalizaci zatížení komunikační infrastruktury) [AM04], software s vysokou mírou rekonfigurovatelnosti a přizpůsobivosti (zakázková výroba [Kou01], řídící systémy letiště [GR96]), distribuované aplikace na Internetu (např. dolování z dat) [KHS97], senzorové sítě [LTO03], sémantický web (agenti vyhledávající pro uživatele komplexní informace) [HM03], simulace socioekonomických procesů [MP01] a mnoho dalších.

Výzkum a vývoj v oblasti multiagentních systémů se v současné době soustřeďuje do několika oblastí. Velká pozornost je věnována nasazování reálných multiagentních aplikací v průmyslu, s čímž úzce souvisí standardizace (o tu se stará zejména organizace FIPA[1]). Tato standardizace se zabývá zejména možnostmi spolupráce mezi heterogenními agenty – tedy jazyky pro komunikaci, službami pro podporu běhu a migraci agentů apod. Současně však stále pokračuje výzkum vhodných vnitřních modelů agenta – a to jednak na rovině teoretické, kde je klíčovou otázkou jaké architektury jsou vhodné pro modelování rozhodovacích procesů agenta a jejich formální popis, ale samozřejmě také praktičtěji zaměřený výzkum na realizaci těchto principů a jejich implementaci pomocí vhodných prostředků.

[1] http://www.fipa.org

1.1 Motivace, cíle disertační práce

Předložená disertační práce se zabývá možnostmi modelování jedné z populárních architektur agenta (architektury BDI) pomocí formalismu *objektově orientovaných Petriho sítí*. Petriho sítě jsou považovány za dobrý nástroj pro modelování paralelismu. Mezi jejich výhody patří snadná pochopitelnost a vizuální reprezentace. Přitom jsou na rozdíl od většiny programovacích jazyků založeny na formálním aparátu, který umožňuje sítě automaticky analyzovat a ověřovat jejich vlastnosti. Multiagentní systémy jsou inherentně paralelní – tento paralelismus pramení ze dvou hlavních zdrojů. Prvním je paralelismus v rámci celého systému, tedy interakce mezi jednotlivými agenty, kdy tyto autonomní jednotky nezávisle na sobě přetváří prostředí ve kterém se nachází. Druhý zdroj paralelismu nacházíme uvnitř agenta, jelikož většina netriviálních agentů obsahuje celou řadu senzorů a aktuátorů, proto pro svůj efektivní chod musí vykonávat celou řadu činností zároveň.

Zdá se tedy, že Petriho sítě by měly být vhodným prostředkem pro tvorbu a modelování agentních systémů. Skutečně, byla publikována celá řada prací na toto téma, například [MW97], [WH02] či [YC06]. Tyto studie však byly většinou realizovány pomocí *barvených Petriho sítí*, případně některé jejich objektové nadstavby. Srovnání s těmito přístupy bude provedeno později, hlavní rozdíl však spočívá v samé povaze použitého formalismu. Zatímco barvené Petriho sítě zůstávají ve své podstatě statické (v modelu mohou pouze vznikat či zanikat značky), objektově orientované Petriho sítě umožňují daleko větší dynamiku modelu. Tato dynamika je pro modelování a tvorbu multiagentních systémů velmi důležitá, domnívám se dokonce, že klíčová. To umožňuje používat při modelování agentů Petriho sítěmi zcela nové přístupy, které se více blíží tradičnímu programování, při zachování výhod Petriho sítí – grafické reprezentace, formálně definované sémantiky, možností vysokoúrovňové verifikace modelu atd.

Nástroj *PNtalk*, který představuje prostředí pro tvorbu modelů a jejich simulaci založený na formalismu objektově orientovaných Petriho sítí se stále nachází ve fázi prototypu. Jedním z cílů předložené práce tak bylo zhodnocení použitelnosti systému PNtalk pro modelování inteligentních systémů obecně, včetně spolupráce s autory na vývoji metodiky modelování v tomto systému. Hlavní cíle předložené práce tak můžeme shrnout do dvou bodů:

- Navrhnout framework pro modelování BDI agentů pomocí objektově orientovaných Petriho sítí, který umožní modelování a vývoj multiagentních aplikací, zároveň však bude otevřený tak, aby umožňoval experimenty se samotnou architekturou BDI.
- Zhodnotit možnosti modelování inteligentních systémů pomocí nástroje PNtalk, doplnit metodiku modelování v tomto systému.

1.2 Struktura práce

Práce se skládá z celkem šesti kapitol. Úlohou první kapitoly bylo stručně uvést čtenáře do problematiky, vysvětlit motivaci a vymezit cíle této disertační práce.

Druhá kapitola se věnuje objektově orientovaným Petriho sítím. Nejprve jsou stručně prezentovány souvislosti, tedy klasické P/T Petriho sítě a barvené Petriho sítě. Následuje neformální popis struktury i dynamiky objektově orientovaných Petriho sítí a představení jazyka a nástroje PNtalk. Zbytek kapitoly se věnuje rozšíření jazyka PNtalk o koncept negativních predikátů, který představuje příspěvek k metodice modelování pomocí tohoto

nástroje a jeden z přínosů předložené práce. Cílem druhé kapitoly je tedy představit nástroje a metody použité v této práci.

Třetí kapitola se věnuje agentům a multiagentním systémům obecně. Nejprve jsou vymezeny základní pojmy, dále je provedeno rozdělení agentů podle jejich vnitřní architektury. Následuje diskuze vybraných částí problematiky multiagentních systémů, které souvisí s předloženou prací – zejména otázky prostředí agenta, komunikace a standardizace multiagentních systémů. Cílem této kapitoly je vymezit používané pojmy a uvést prezentovanou práci do kontextu.

Čtvrtá kapitola se podrobněji zabývá architekturou BDI z různých pohledů – filozofického, formálního a implementačního, společně s přehledem nejznámějších projektů v této oblasti. Cílem této kapitoly je dát teoretická východiska pro prezentovanou práci.

Pátá kapitola představuje vlastní framework a představuje tak hlavní přínos předložené práce. Nejprve jsou diskutovány základní principy fungování frameworku, dále jsou podrobně představeny jeho jednotlivé aspekty – představy, události, plány, záměry, atd. Poté je představena ukázka nasazení frameworku v reálné aplikaci – řízení skupiny mobilních robotů. Na závěr je provedeno porovnání frameworku s ostatními přístupy v dostupné literatuře.

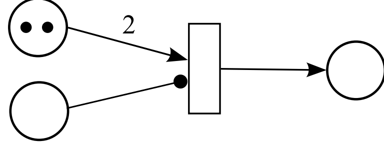
V závěrečné šesté kapitole jsou shrnuty výsledky disertační práce, diskutovány možnosti dalšího výzkumu a provedeno zhodnocení splnění cílů.

Kapitola 2

Objektově orientované Petriho sítě

Petriho sítě představují třídu matematických modelů používaných zejména pro modelování paralelismu, kauzality a nedeterminismu v diskrétních distribuovaných systémech. Nazývají se podle německého matematika C. A. Petriho, který ve své doktorské disertační práci v roce 1962 zavedl nové koncepty popisu závislosti mezi podmínkami a událostmi v modelovaném systému. Koncepčně vycházejí z paralelně pracujících a komunikujících konečných automatů. Kromě formálního popisu mají taktéž grafickou reprezentaci ve formě orientovaného bipartitního grafu, díky které je modelování systémů jejich prostřednictvím intuitivní. Tyto grafy se skládají z *míst* a *přechodů* spojených *orientovanými hranami*. Místa mohou obsahovat libovolný počet *značek*. Stav systému je vyjádřen rozložením značek v místech sítě, toto rozložení se nazývá *značení*.

Klasické Petriho sítě, nazývané též *P/T Petriho sítě* (Place/Transition Petri nets), patří k velmi rozšířeným formalismům a jsou poměrně dobře popsány v řadě publikací, například v [Češ94]. P/T Petriho sítě je možné celkem snadno doplnit o *inhibiční hrany* (hrany, které vyžadují, aby v místě bylo méně značek, než je ohodnocení této hrany). Příklad grafického znázornění takové sítě je na obrázku 2.1 (inhibiční hrana je znázorněna jako hrana, která není ukončena šipkou, ale malým kroužkem). Takovéto sítě již dosahují výpočetní síly Turingova stroje [DA92], takže teoreticky jsou schopny modelovat libovolný algoritmus. Z praktického hlediska však mají jedno zásadní omezení – neposkytují žádné prostředky pro strukturování modelu. Při modelování rozsáhlých systémů se tak výsledný model stává velmi nepřehledným, případně zachycuje pouze vztahy mezi základními komponentami systému. Pro překonání tohoto omezení byla navržena celá řada rozšíření, která bývají souhrnně označována jako *vysokoúrovňové Petriho sítě*, HLPN (High-Level Petri nets). Zřejmě nejvíce používanou variantou HLPN jsou *Barvené Petriho sítě*, CPN (Coloured Petri nets).



Obrázek 2.1: Příklad P/T Petriho sítě doplněné o inhibiční hranu (znázorněna ukončením hrany malým kroužkem místo šipky)

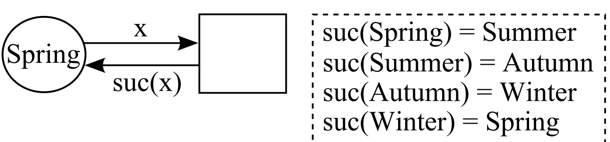
2.1 Vysokoúrovňové Petriho sítě

Barvené Petriho sítě [Jen97] zavedené prof. Jensenem představují velmi významné rozšíření původních P/T Petriho sítí. Zatímco v P/T Petriho sítích jsou značky nerozlišitelné, v CPN může každá značka nést hodnotu. Barva značky pak udává její typ. Přechody mohou mít podmínky proveditelnosti, kontrolující atributy těchto značek. Kromě grafického vyjádření sítě je model doplněn o *formální inskripci*, která obsahuje:

- definice typů značek v modelu;
- počáteční značení jednotlivých míst;
- hranové výrazy určující jaké značky jsou odebírány či přidávány z míst, které jsou spojeny danou hranou s přechodem;
- strážní podmínky příslušející daným přechodům, jež omezují jejich proveditelnost v závislosti na atributech značek;
- procedury, opět spojené s jednotlivými přechody, které umožňují provádět výpočty se značkami, včetně vedlejších efektů.

Příklad grafické reprezentace jednoduché barvené Petriho sítě ukazuje obrázek 2.2.

Výsledný modelovací jazyk spojuje výhody klasické Petriho sítě a vysokoúrovňového programovacího jazyka. Pro práci s CPN autoři již od počátku vyvíjejí modelovací nástroje, takže veškerá teorie může být ihned aplikována v praxi. Původní nástroj *Design CPN* dosáhl poměrně velkého rozšíření, v současné době je nahrazen novějším softwarem *CPN Tools*. Tyto nástroje se skládají z několika komponent – grafického editoru sítí, který umožňuje intuitivní tvorbu modelů; interaktivního simulátoru běhu sítě a konečně sady nástrojů pro ověřování vlastností modelů.



Obrázek 2.2: Příklad barvené Petriho sítě

Formalismus barvených Petriho sítí umožňuje modelovat výrazně komplexnější systémy než P/T Petriho sítě. Je to dáno zejména možností některé části modelovaného systému vyjádřit pomocí operací se značkami. Hierarchické barvené Petriho sítě (HCPN) dále rozšiřují CPN o podporou pro statické strukturování sítě pomocí mechanismu stránek. Jistým omezením však zůstává celková statičnost sítě – v síti mohou vznikat a zanikat značky, není však možné měnit strukturu sítě (přidávat či ubírat místa a přechody) během jejího provádění.

Tuto nevýhodu se snažili odstranit různí autoři spojením konceptů Petriho sítí a technik objektově orientovaného modelování. Jedním z těchto přístupů jsou *objektově orientované Petriho sítě*, OOPN (Object oriented Petri nets), které zavedl V. Janoušek ve své doktorské disertační práci [Jan98].

OOPN spojují výhody obou použitých paradigmat – Petriho sítě umožňují formálně popsat vlastnosti modelovaného systému, zatímco objektová orientace přináší přirozené

možnosti strukturování modelu a možnost dynamického vytváření instancí sítí. Inspirací pro objektový model OOPN byl jazyk Smalltalk, což ve stručnosti znamená, že:

- Každý objekt je instancí nějaké třídy.
- Veškeré výpočty v rámci inskripčního jazyka se provádějí zasíláním zpráv mezi objekty.
- Proměnné obsahují reference na objekty.
- Třída definuje chování svých instancí jako množinu metod, které specifikují reakci na zaslání zprávy.
- Třídy jsou definovány inkrementálně, tedy každá třída je podtřídou nějaké existující třídy (používá se jednoduchá dědičnost).

Tento základní model byl obohacen o prvek souběžnosti – objekty jsou chápány jako aktivní servery, které poskytují služby ostatním objektům. Vlastní chování objektu, stejně jako poskytované služby jsou popsány pomocí Petriho sítí.

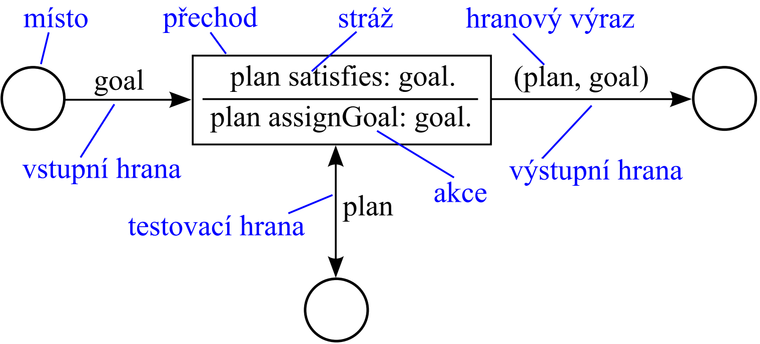
2.2 Neformální specifikace OOPN

V této části budou stručně a neformálně popsány základy formalismu objektově orientovaných Petriho sítí. Nejprve se budu zabývat strukturou OOPN, tedy jaké prostředky OOPN poskytují pro popis modelů, a poté dynamikou OOPN, tedy jak jsou tyto modely interpretovány. Smyslem této části je seznámit čtenáře s formalismem OOPN tak, aby rozuměl popisu a modelům prezentovaným v dalších částech práce. Formální definice všech pojmů diskutovaných dále je možné nalézt v [Jan98].

2.2.1 Základní prvky sítí v OOPN

Sítě se v OOPN skládají z míst a přechodů, které jsou spojeny orientovanými hranami. Příklad grafického vyjádření jednoduché sítě s popisem jednotlivých prvků ukazuje obrázek 2.3.

Místa mají stejný význam jako v běžných Petriho sítích, zachycují tedy parciální stav modelovaného systému. Každé místo může mít počáteční značení (multimnožinu značek).



Obrázek 2.3: Základní prvky sítě OOPN

Přechody vyjadřují změny v systému, každý přechod má dvě části – *stráž* a *akci*. Stráž omezuje proveditelnost přechodu, skládá se z konjunkce booleovských výrazů (booleovským výrazem se rozumí zaslání zprávy objektu jejímž výsledkem je booleovský návratový typ). Akce umožňuje provádět výpočty se značkami, tj. zasílat zprávy objektům s možností přiřadit výsledek do proměnné. Je možné provést více akcí v jednom přechodu, tento přechod se pak chápe jako sekvence přechodů s vloženými místy tak, aby mezi těmito přechody došlo k přenosu příslušných značek a aby takto upravená síť odpovídala definici Petriho sítě.

Hrany reprezentují vstupní a výstupní podmínky přechodů. Vstupní podmínka je graficky reprezentovaná šipkou směřující od místa k přechodu, výstupní podmínka šipkou od přechodu k místu. Dále jsou zavedeny testovací hrany, jež jsou graficky reprezentované oboustrannými šipkami. Hrany jsou ohodnoceny hranovými výrazy, což jsou opět multimnožiny značek, přičemž některé značky mohou být nahrazeny volnými proměnnými. Tyto proměnné se pak navazují na konkrétní značky při testování proveditelnosti přechodu.

Značky představují reference na objekty nebo jejich n-tice, tyto objekty mohou být buď tzv. *statické* (primitivní) nebo *neprimitivní*. Statické objekty jsou chápány jako konstanty, jejichž jména lze ztotožnit s obsahem (hodnotou) objektu. Jedná se například o čísla, znaky, řetězce, symboly, apod. Neprimitivní objekty jsou popsány pomocí Petriho sítí a jejich strukturou se zabývá následující podkapitola.

2.2.2 Struktura neprimitivních objektů

Neprimitivní objekty jsou definovány svojí třídou. Třída obsahuje:

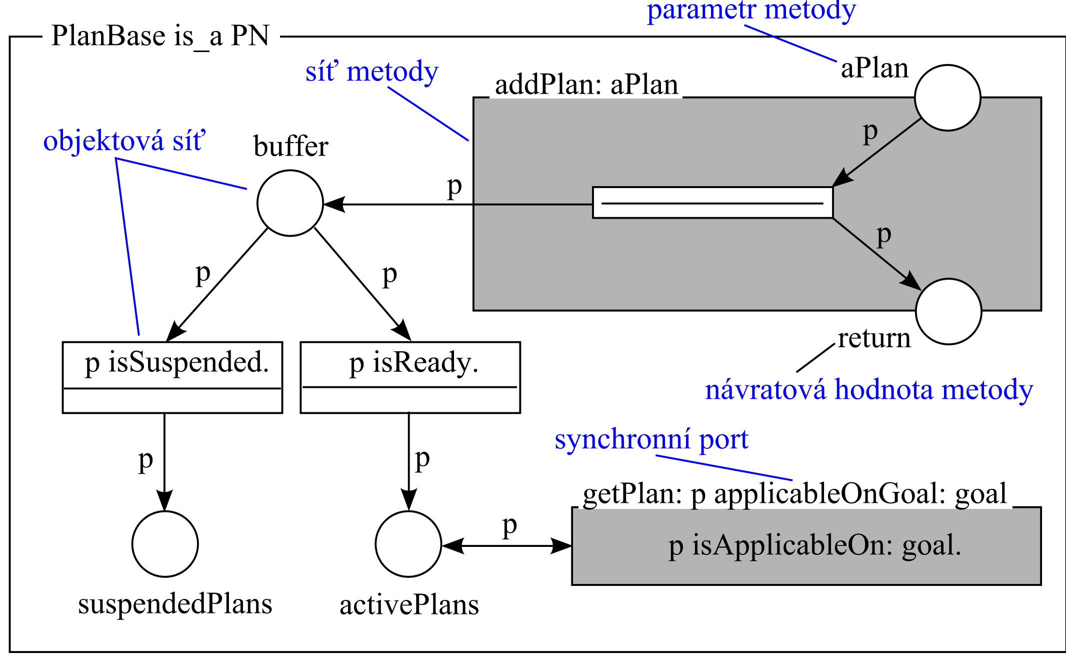
- právě jednu *objektovou síť* (která však může být prázdná),
- množinu *sítí metod*,
- množinu *synchronních portů*.

Objektová síť popisuje autonomní chování objektu. Tato síť je instanciovaná v okamžiku vytvoření objektu a její doba života je svázána s dobou života objektu.

Sítě metod popisují reakce na zprávy, které jsou objektu zasílané z akcí přechodů. Tyto sítě jsou instanciovány v okamžiku vyvolání metody a zrušeny při jejím ukončení. Pro každý parametr metody obsahuje síť speciální místo se stejným názvem, jaký má formální parametr, do kterého se v okamžiku vyvolání metody vloží značky reprezentující skutečné parametry metody. Síť dále obsahuje jedno speciální místo s názvem *return*, do kterého se uloží návratová hodnota metody. Toto místo nemůže být součástí vstupní podmínky žádného přechodu. Přechody sítě metody mohou být propojeny hranami s místy v objektové síti objektu, které tak mohou sloužit jako členské proměnné.

Synchronní porty definují reakce objektu na synchronní zprávy zasílané ze stráží přechodů. Jsou to v podstatě speciální přechody, které nemají žádnou akci a slouží k testování stavu objektu, případně provádění postranního efektu. Důležité je, že tyto porty se musí vyhodnotit atomicky, protože se stávají součástí stráže přechodu, který synchronní zprávu zaslal. Pokud je tento přechod proveden, provede se i postranní efekt synchronního portu. Synchronní porty, které nemají žádný postranní efekt a slouží tedy čistě k testování stavu objektu, nazýváme *predikáty*. Příklad třídy neprimitivního objektu ukazuje obrázek 2.4.

Třídy se v OOPN definují inkrementálně s využitím dědičnosti. Kořenem stromu dědičnosti (připomeňme, že OOPN podporují jednoduchou dědičnost) je třída *PN*, která má prázdnou objektovou síť i množiny sítí metod a synchronních portů. Tato třída definuje základní chování všech aktivních objektů v OOPN.



Obrázek 2.4: Příklad třídy neprimitivního objektu v OOPN

2.2.3 Dynamické chování OOPN

Stav OOPN se nazývá *značení*, můžeme jej chápat jako systém OOPN objektů. Dynamické chování OOPN pak odpovídá evoluci tohoto systému. Jedna třída je vždy označena jako hlavní, pro tuto třídu se na počátku simulace vytvoří jedna její instance – objekt. Inicializovanou objektovou síť tohoto objektu nazýváme *počáteční značení*.

Evoluce sítí v OOPN je založena na *událostech*, což je zásadní rozdíl oproti evoluci kódu klasických imperativních programovacích jazyků, která je založena na *procesech*. Události v OOPN odpovídají proveditelnosti přechodu. Podle typu proveditelnosti se rozlišují čtyři typy událostí [Jan98]:

- *Událost typu A* (atomic) – atomické provedení přechodu, v akci přechodu se zasílá zpráva statickému objektu. Odebrání značek ze vstupních míst, provedení akce a uložení značek do výstupních míst přechodu se provede současně, jako nedělitelná operace.

- *Událost typu N* (new) – vytvoření nového neprimitivního objektu, přechod je také proveden atomicky, ale vznikne přitom nový objekt.

- *Událost typu F* (fork) – zaslání zprávy neprimitivnímu objektu. Přechod čeká na dokončení vyvolané metody, jedná se tedy o neúplné provedení přechodu. Odeberou se značky ze vstupních míst a vytvoří se nová instance sítě odpovídající metody (to vše jako atomická operace).

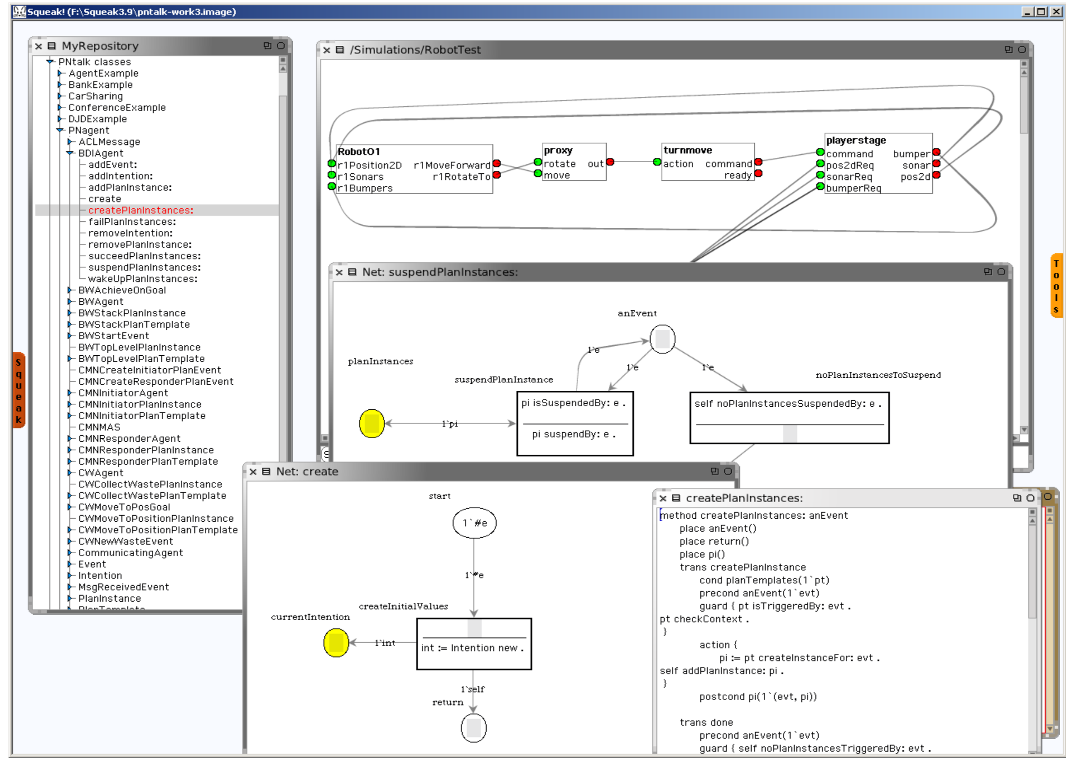
- *Událost typu J* (join) – dokončení přechodu po skončení vyvolané metody. Zruší se instance sítě odpovídající metody a uloží se značky do výstupních míst přechodu. To vše jako atomická operace.

Po každé události se provede *garbage collecting*, který zajistí skutečné zrušení instancí sítí na které neexistuje reference.

2.3 Jazyk a systém PNtalk

Na formalismu objektově orientovaných Petriho sítí je založen jazyk a systém *PNtalk* (Petri Nets & Smalltalk). Textově grafický jazyk PNtalk představuje konkrétní implementaci OOPN. Vychází přímo z jejich definice, má však některé drobné syntaktické odlišnosti a vylepšení (konstruktory, složené zprávy apod.), inspirované jazykem Smalltalk. Konkretizuje také ty části definice OOPN, kde původní specifikace ponechává určitou volnost – především v oblasti statických objektů. Syntaxi textové formy jazyka ve formě Backus-Naurovy formy je možno nalézt v [Koč04].

Systém PNtalk pak představuje nástroj pro modelování, simulaci, verifikaci a prototypování systémů, postavený na tomto jazyce, implementovaný v prostředí Smalltalk (viz obrázek 2.5). Systém PNtalk je vyvíjen na VUT v Brně od roku 1993, první verze nesla označení *PNtalk 96*. Od počátku umožňuje transparentní spolupráci objektů OOPN a Smalltalku. Současná verze tohoto nástroje nese označení *PNtalk 2007*. PNtalk 2007 byl navržen jako otevřený simulační systém na bázi metaúrovňové architektury [JK03]. Hlavní výhody tohoto přístupu spočívají v možnosti modifikovat sémantiku modelu v průběhu jeho provádění a kombinovat různé modelovací formalismy (heterogenní simulace). Nevýhodu pak představují zvýšené nároky na výpočetní výkon.



Obrázek 2.5: Systém PNtalk 2007 – ukázka grafické i textové podoby jazyka a kombinace s heterogenním modelem na bázi DEVS

Jedna z možností heterogenní simulace byla využita v modelu řízení robota prezentovaném dále v této práci. Zde se jednalo o propojení PNtalku s nástrojem SmallDEVS [JK06], taktéž vyvíjeném na VUT v Brně, který je založen na formalismu *DEVS* (Discrete Event System Specification) [ZKP00]. PNtalk je v tomto případě zapouzdřen jako atomická komponenta DEVSu, komunikace probíhá pomocí předávání značek do určených míst ze vstupních portů komponenty a naopak přesunem značek z jiných míst do výstupních portů komponenty [JK07]. Pro popis řídící části robota je tak použit vysokoúrovňový formalismus OOPN, který umožňuje snadno vyjadřovat paralelismus a zpracování plánů, zatímco pro vlastní komunikaci s robotem či simulátorem pomocí protokolu TCP/IP jsou využity jiné formalismy vhodnější pro tento účel v nezávislých komponentách. DEVS představuje rámec pro spolupráci těchto formalismů.

V současné době se celý projekt PNtalk zaměřuje zejména na podporu pro vývoj a prototypování systémů založený na modelech a simulaci (model based design – MBD). Hlavní myšlenka MBD spočívá v tom, že model je používán od počátečních fází návrhu až po finální prototyp výsledné aplikace, případně se může stát přímo její součástí. Předpokládanou oblast využití PNtalku představuje především návrh a vývoj inteligentních systémů – příkladem je framework pro tvorbu inteligentních agentů prezentovaný v této práci. Celý systém PNtalk je stále ve fázi prototypu, jedním z přínosů předložené práce tak byla spolupráce s autory PNtalku na obecné metodice modelování v PNtalku. V následující části bude prezentováno navržené rozšíření jazyka PNtalk o koncept *negativních predikátů*, které umožňuje výrazně zjednodušit jisté části modelu. Toto rozšíření bylo publikováno v [MJK08] a je používáno v modelech prezentovaných dále v této práci.

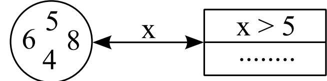
2.4 Navržené rozšíření jazyka PNtalk

Jak již bylo řečeno, formalismus OOPN, na kterém je jazyk PNtalk postaven, patří mezi vysokoúrovňové Petriho sítě a velká část jeho principů je obdobná jako u barvených Petriho sítí. Hlavní rozdíl mezi P/T a vysokoúrovňovými Petriho sítěmi spočívá v tom, že u vysokoúrovňových Petriho sítí mohou značky nést hodnotu, se kterou lze manipulovat v rámci přechodu pomocí inskripčního jazyka. Při vytváření modelu je tak potřeba rozhodnout, které části modelovaného systému vyjádřit pomocí sítě a které pomocí manipulace se značkami. Toto rozhodnutí nemusí být vždy jednoduché.

U původních P/T Petriho sítí není možné testovat nepřítomnost značky v místě. To je také důvodem, proč P/T Petriho sítě nejsou výpočetně úplné [Češ94]. Výpočetní úplnosti je možné dosáhnout pouze rozšířením formalismu, například o inhibiční hrany. Vysokoúrovňové Petriho sítě inhibiční hrany nepotřebují, jelikož stejného efektu může být dosaženo konstrukcemi inskripčního jazyka. Nezavedení inhibičních hran je důležité z hlediska analýzy sítě pomocí invariantů, avšak může způsobit jisté komplikace při modelování, což demonstruje následující příklad.

Značky budou nabývat hodnot celých čísel. Stráž jistého přechodu bude mít tvar $x > 5$ – chceme tedy, aby přechod byl proveditelný, pokud značka má hodnotu větší než pět. Často však potřebujeme také vykonat nějakou akci v případě, že podmínka splněna není. To odpovídá konstrukci *if (podmínka) then akce1 else akce2* v imperativních programovacích jazycích. Ve vysokoúrovňových Petriho sítích se tento případ modeluje dvěma přechody, jeden má ve stráži podmínku zmíněnou výše ($x > 5$) a ten druhý její negaci ($x <= 5$). Oba přechody musí být spojeny vstupní hranou s nějakým místem v síti, ve kterém budou značky uloženy. Nyní předpokládejme, že v místě již několik značek je, a my chceme, aby první přechod byl proveditelný v případě, že se v místě nachází alespoň jedna značka s danou

vlastností ($x > 5$). To je samozřejmě velmi jednoduché, jak ukazuje obrázek 2.6.



Obrázek 2.6: Příklad testu přítomnosti značky s danou vlastnosti v místě ve formalismu OOPN

Problém nastane, pokud požadujeme proveditelnost přechodu v případě, že *žádná značka v tomto místě nesplňuje danou podmínku* – bez dodatečných míst a přechodů, či úplné změny stylu ukládání značek toho nejsme schopni. Schopnost najít v místě jednu značku s danou vlastností není tedy v HLPN symetrická schopnosti otestovat, že žádná značka v daném místě vlastnost nesplňuje. To je způsobeno právě absencí inhibičních hran. V praxi to znamená, že pokud potřebujeme otestovat, zda žádná značka v daném místě nesplňuje nějakou vlastnost, musíme pro ukládání značek v místě použít komplexnější struktury inskripčního jazyka.

2.4.1 Řešení v barvených Petriho sítích

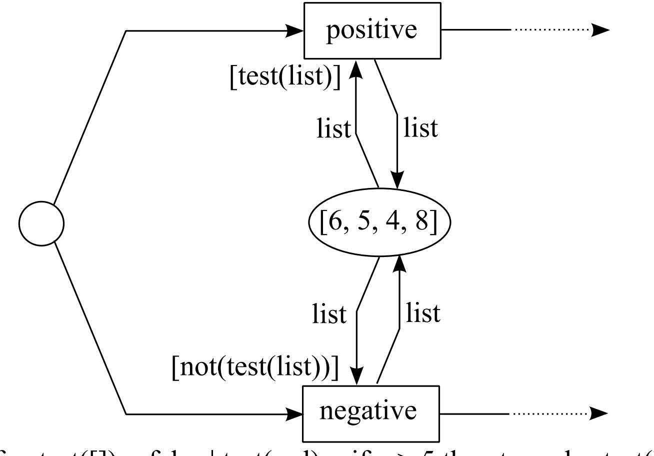
Pro modelování pomocí barvených Petriho sítí existuje výrazně propracovanější metodika, než je tomu u OOPN. Dokonce byla vytvořena celá řada *návrhových vzorů* [MvdA05], které mají usnadnit řešení typických problémů návrhu, obdobně jako je tomu u metodiky pro objektově orientovaný návrh. Tyto vzory mimo jiné nabízí možná řešení pro situaci prezentovanou výše – konkrétně se jedná o dva vzory:

První vzor se nazývá *Inhibiční hrana*. Je založen na myšlence počítání značek v daném místě pomocí jiného místa obsahující jednu značku typu celé číslo – počítadlo. Vždy, když nějaký přechod přidá či odebere z původního místa značku, musí také zvýšit resp. snížit počítadlo. Značky v původním místě tak mohou být na danou vlastnost či její negaci otestovány jedna po druhé přechodem, u kterého se pomocí počítadla zajistí, že se provede přesně tolikrát, kolik je v daném místě značek. Toto řešení je samozřejmě velmi neohrabané a komplikuje model přidáním mnoha míst a přechodů, čímž se snižuje přehlednost a tím i význam použití modelu.

Druhý vzor se nazývá *Barvená inhibiční hrana* a je mnohem vhodnější, protože umožňuje otestovat místo přímo na nepřítomnost značky s danou hodnotou či vlastností. Principem je agregování značek do jedné, která je typu *seznam*. Tento seznam pak může být otestován na nepřítomnost značky pomocí funkce inskripčního jazyka ve stráži přechodu. Aplikaci vzoru Barvená inhibiční hrana na diskutovaný příklad ukazuje obrázek 2.7.

Tento návrhový vzor na první pohled zcela uspokojivě řeší zadaný problém. Nevýhodou však je, že pokud chceme testovat nepřítomnost značky s nějakou hodnotou v daném místě, musí toto místo obsahovat právě jednu značku typu seznam. To znamená, že pro každé získání jakékoliv značky musíme používat funkci ve stráži přechodu, přičemž získání náhodné značky je v tomto případě paradoxně obtížnější modelovat než získání nějaké konkrétní značky.

Při návrhu inteligentních systémů (a vývoji založeném na modelování obecně) se často stává, že specifikace nejsou dopředu jasně dané a vyvíjejí se během modelovacího procesu, stejně jako samotný model. V tomto případě je tedy poměrně těžké a omezující dopředu určit, zda bude v daném místě potřeba testovat nepřítomnost značky s určitou hodnotou.



Obrázek 2.7: Aplikace vzoru Barvená inhibiční hrana

Jinými slovy, v extrémním případě by se měl tento vzor používat ve všech případech, kdy není dopředu zcela zřejmé, že v místě bude vždy obsažena právě jedna značka.

2.4.2 Řešení v jazyku PNtalk

Vzhledem k tomu, že cílem PNtalku je podpora pro prototypování a vývoj založený na modelování, není řešení ve stylu barvených Petriho sítí plně uspokojivé. Proto jsem se snažil najít možná rozšíření jazyka, která by umožňovala komfortní řešení těchto situací. Důležitým požadavkem na rozšíření bylo, aby mělo co nejmenší vliv na zbytek jazyka a bylo snadné pro implementaci. V optimálním případě by mělo být možné rozšíření transformovat na existující jazykové struktury, takže formalismus OOPN by nebylo potřeba vůbec měnit.

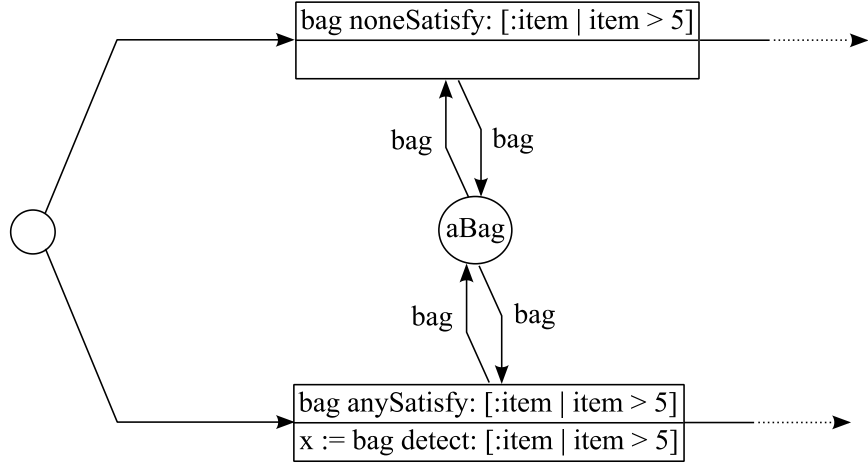
Užití kolekcí pro ukládání značek

První možné řešení představuje přímá adaptace výše zmíněného návrhového vzoru Barvená inhibiční hrana. Jako inskripční jazyk PNtalku je použit jazyk Smalltalk, pro agregaci značek v místech je tedy možné využít koncept Smalltalkovských kolekcí. Kolekce nabízejí přístup ke svým prvkům na vyšší úrovni než seznamy inskripčního jazyka barvených Petriho sítí – poskytují celou řadu metod pro testování a zpřístupnění svého obsahu. Příklad použití kolekce *Bag* pro řešení diskutovaného příkladu demonstruje obrázek 2.8.

Toto řešení nás samozřejmě nezbavuje nutnosti použití speciální agregační značky pro test nepřítomnosti značky s určitou hodnotou v daném místě, ale je již v jazyku PNtalk dostupné. V optimálním případě by tedy mělo být možné převést navržené rozšíření na tento vzor.

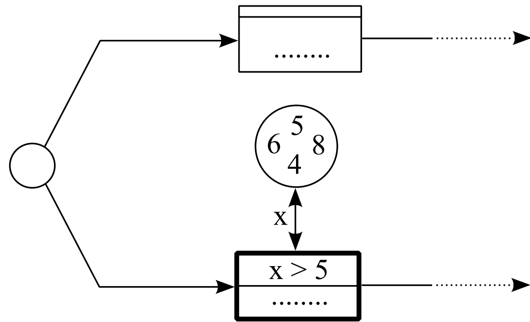
Zavedení priorit přechodů

Další možností, jak se s problémem testování nepřítomnosti značky s určitou vlastností v daném místě vyrovnat, je zavést do jazyka PNtalk priority přechodů. U P/T Petriho sítí má rozšíření formalismu o koncept priorit přechodů stejný efekt jako zavedení inhibičních hran – stanou se výpočetně úplnými.



Obrázek 2.8: Adaptace návrhového vzoru Barvená inhibiční hrana do jazyka PNtalk

Priority přechodů také řeší problém testování nepřítomnosti značky s danou vlastností v místě – umožňují totiž přímočarou implementaci konstrukce imperativních programovacích jazyků *if (podmínka) then akce1 else akce2*. V případě, že je podmínka splněna (alespoň jedna značka s danou vlastností je v místě přítomna), provede se přechod, jenž má ve stráži test této vlastnosti a kterému je přiřazena vyšší priorita. V opačném případě se provede přechod s prázdnou stráží, jemuž je přiřazena priorita nižší. Situaci znázorňuje obrázek 2.9 (přechod s vyšší prioritou je označen tučným okrajem).



Obrázek 2.9: PNtalk rozšířený o priority přechodů (přechod s vyšší prioritou je označen tučným okrajem)

Tento přístup má však několik nevýhod. Podle definice dynamiky OOPN zavedené v [Jan98] je v každém kroku simulace obecně několik proveditelných událostí (které představují podmnožinu množiny všech událostí), z nichž se jedna vybere nedeterministicky a ta je provedena. Hlavní problém spočívá v tom, že priority přechodů definují částečné uspořádání na celé této množině, což znamená, že se priority neaplikují pouze pro uspořádání přechodů, které jsme zamýšleli uspořádat, ale na všechny v daném okamžiku proveditelné přechody v modelu.

Nejjednodušší implementaci priorit přechodů představuje lexikografické uspořádání, ale to zároveň představuje nejhorší variantu uspořádání – všechny přechody se stejnou prioritou

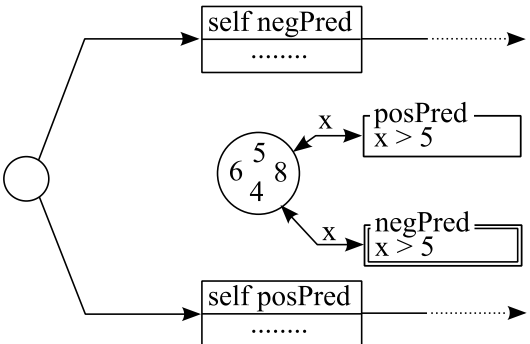
by se musely jmenovat stejně, jinak by došlo k jejich provádění v závislosti na uspořádání jejich jmen. Tímto způsobem by tak došlo k přímému rozporu s původním určením Petriho sítí – modelování paralelních procesů a nedeterministických voleb.

Rozšíření jazyka o negativní predikáty

Jinou možností je rozšířit jazyk o nějakou formu inhibičních hran. Jak bylo ukázáno v části popisující řešení u barvených Petriho sítí, obdoba klasických inhibičních hran ve stylu P/T Petriho sítí nevede k dobrým výsledkům. Proto jsem navrhl koncept *negativních predikátů*.

Negativní predikát vychází z koncepce synchronního portu, z hlediska syntaktického je s ním téměř shodný. Hlavní rozdíl spočívá v sémantice – synchronní port je proveditelný, pokud existuje nějaké navázání volných proměnných v hranových výrazech vstupních hran tak, že toto navázání splňuje podmínky definované stráží portu. Negativní predikáty fungují obráceně – jsou proveditelné v případě, že žádné takové navázání nemůže být nalezeno.

Negativní predikáty nemohou mít postranní efekty, protože tato operace by nedávala smysl – aby byl negativní predikát proveditelný, některá z proměnných nemohla být navázána. To znamená, že negativní predikáty mohou být spojeny s místy pouze testovacími hranami. Použití negativního predikátu pro řešení našeho příkladu ukazuje obrázek 2.10 (negativní predikát je označen dvojitou čárou pro odlišení od synchronních portů).



Obrázek 2.10: Ukázka použití negativního predikátu v PNtalku (negativní predikát je označen dvojitou čárou)

Je tedy zřejmé, že negativní predikáty řeší problém testování nepřítomnosti značky s určitou vlastností v daném místě. Z hlediska syntaxe jazyka PNtalk jsou totožné se synchronními porty až na dva rozdíly: V grafické podobě jazyka jsou znázorněny obdélníkem s dvojitým okrajem a s místy mohou být spojeny pouze pomocí testovacích hran. Syntax upravené textové podoby jazyka je formálně definována rozšířenou Backus-Naurovou formou v příloze A. Použití negativních predikátů je možné vždy převést na případ užití agregace značek v místech do kolekcí prezentovaný výše. Hlavní myšlenku tohoto převodu ukazuje následující podkapitola.

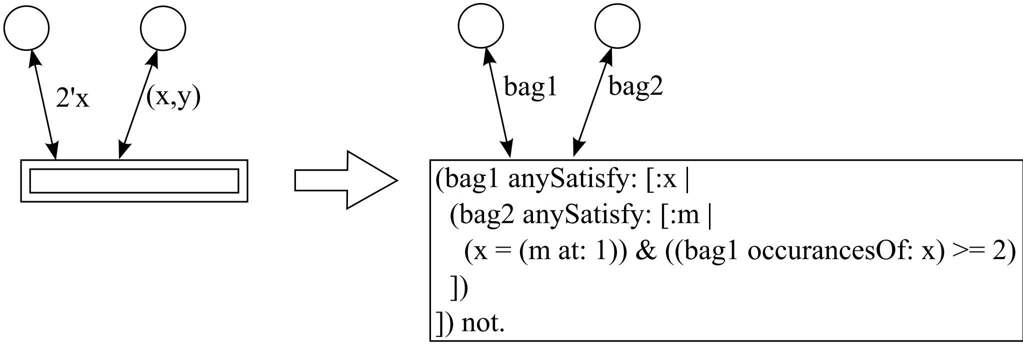
2.4.3 Převod negativních predikátů na existující jazykové struktury

V této části bude ukázán převod výrazu stráže a hranových výrazů negativního predikátu na struktury již dostupné v jazyku PNtalk. Převod předpokládá, že zbytek sítě již byl upraven tak, že v místech, která jsou s negativním predikátem spojena, jsou značky agregovány

do Smalltalkovské kolekce *Bag*. Obdobně jako negativní predikáty je samozřejmě potřeba upravit stráže a hranové výrazy přechodů a synchronních portů, které pracují s těmito místy. Postup jejich převodu je však prakticky totožný jako negativních predikátů, proto jej zde nebudu opakovat. Vlastní převod se provede pomocí následujícího algoritmu:

> Předpokládejme, že ve všech místech, které jsou s daným negativním predikátem spojeny hranou (negativní predikáty mohou být s místy spojeny pouze testovacími hranami, proto je v tomto algoritmu nebudu rozlišovat), jsme nahradili běžné vkládání značek agregací do Smalltalkovské kolekce *Bag*.
>
> Nejprve provedeme přejmenování proměnných v hranových výrazech: pokud se některá volná proměnná (např. p) vyskytuje v hranovém výrazu dvou různých vstupních hran daného predikátu, přejmenují se všechny její výskyty v jednom z nich tak, aby tato nová proměnná měla unikátní název v rámci celého predikátu (např. $p2$). Do výrazu stráže predikátu se přidá podmínka rovnosti obou proměnných (např. $p = p2$).
>
> Pro každou hranu z množiny vstupních hran predikátu provedeme následující úpravy:
>
> – Nahradíme její hranový výraz jednou novou proměnnou s unikátním názvem v rámci celého predikátu, do které se naváže kolekce, která v daném místě agreguje značky (např. $xBag$).
> – Pro každý term původního hranového výrazu, specifikující četnost značky větší než jedna (např. *c'm*), přidáme do stráže predikátu výraz tvaru *((xBag occurrencesOf: m) >= c)*.
> – V případě, že m má tvar seznamu, vytvoříme pro něj novou proměnnou (např. lm) a pro každý jeho prvek, který je konstantou (např. 11) do stráže podmínku tvaru *((lm at: pos) = 11)*, kde *pos* je číslo odpovídající pozici, na které se konstanta x v seznamu m nachází. Pokud je prvkem seznamu proměnná, která je použita někde ve výrazu stráže, nahradí se na všech takovýchto místech výrazem *(lm at: pos)*, kde *pos* je číslo odpovídající pozici, na které se proměnná v seznamu m nachází.
> – Pro každý term m vložíme stráž predikátu do následující konstrukce:
> *(xBag anySatisfy:* [: *m* | <*stráž predikátu*>])
>
> Nakonec konjunkci ve stráži predikátu upravíme tak, aby odpovídala syntaxi výrazu jazyka Smalltalk, tedy např. stráž tvaru $x > 5$. *y testWith: x.* upravíme na $(x > 5)$ & *(y testWith: x)*.
>
> Výsledný výraz ve stráži je konstrukcí inskripčního jazyka Smalltalk a vrací hodnotu *true* právě v těch případech, kdy byla stráž původního predikátu splněna. Pro splnění sémantiky negativního predikátu je tedy nutné tuto hodnotu znegovat zasláním zprávy *not*.

Negativní predikát se tak stal synchronním portem, který má ve stráži jeden výraz inskripčního jazyka, a který je proveditelný právě v těch případech, kdy byl proveditelný původní negativní predikát. Příklad převodu ukazuje obrázek 2.11.



Obrázek 2.11: Příklad převodu negativního predikátu na synchronní port pracující s kolekcemi

2.4.4 Negativní predikáty a logické programování

Koncept negativních predikátů byl inspirován implementací predikátu *not(P)* v logických programovacích jazycích, např. v jazyce Prolog. Tento přístup bývá nazýván *negace jako selhání* (negation as failure) [Cla87]. Predikát $not(P)$ selže ve všech případech, ve kterých by predikát P uspěl. Hlavní výhodou tohoto přístupu je jednoduchá a poměrně efektivní implementace.

V logickém programování tento přístup vyvolal mnoho diskuzí, a to ze dvou hlavních důvodů. Prvním z nich je *předpoklad uzavřenosti světa* (closed world assumption), který je touto implementací implikován. Jeho podstata spočívá v tom, že cokoliv, o čem není známo, že je pravdivé (resp. je možné to odvodit ze známých faktů), je považováno za nepravdivé. Tento problém souvisí zejména s použitím logických jazyků pro reprezentaci znalostí. V PNtalku negativní predikáty jednoduše říkají, že žádná značka v daném místě nesplňuje podmínku definovanou stráží tohoto predikátu.

Druhý problém spočívá v tom, že predikát $not(P)$ implementovaný pomocí negace jako selhání v Prologu neodpovídá predikátu $\neg P$ v klasické predikátové logice. V některých případech může totiž dávat neočekávané výsledky. Ukážu klasický příklad. Mějme databázi v následujícím tvaru:

dobryStudent(X) :- chytry(X), not(liny(X)).
liny(jirka).
chytry(honza).

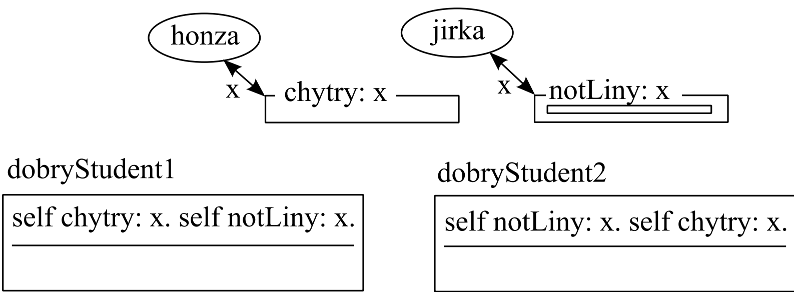
Na dotaz *dobryStudent(X)* obdržíme správnou odpověď $X = honza$. Nyní upravíme pravidlo dobryStudent na následující tvar:

dobryStudent(X) :- not(liny(X)), chytry(X).

V případě, že by *not(P)* fungovalo jako $\neg P$ v predikátové logice, měli bychom obdržet stejnou odpověď jako před úpravou. Odpověď však bude *fail* – predikát selže. To je dáno tím, že v Prologu je predikát *not(P)* implementován pomocí operátoru řezu, konkrétně:

```
not(G) :- G, !, fail.
not(G).
```

Po navázání proměnné X na term jirka není možné se kvůli operátoru řezu vrátit a vyzkoušet jiné navázání (na term honza). Obdobně je tomu v případě PNtalku, jak ilustruje obrázek 2.12. Zatímco přechod *dobryStudent1* bude proveditelný, přechod *dobryStudent2* nikoliv. Je to dáno tím, že PNtalk se snaží navázáním proměnných postupně splnit jednotlivé výrazy ve stráži přechodu. Zatímco v případě přechodu *dobryStudent1* je negativní predikát volán na již navázanou proměnnou a v důsledku toho nenajde žádnou možnou unifikaci a uspěje, v případě *dobryStudent2* je volán s volnou proměnnou, do které je schopen navázat term *jirka*. Synchronní port by tedy byl splněn, negativní predikát proto splněn není. V důsledku toho není splněna celá stráž přechodu. Při používání negativních predikátů je tedy potřeba věnovat pozornost tomu, zda budou volány na již navázané nebo volné proměnné.



Obrázek 2.12: Demonstrace rozdílného chování přechodů v závislosti na pořadí výrazů ve stráži při použití negativních predikátů. Zatímco přechod *dobryStudent1* bude proveditelný, přechod *dobryStudent2* nikoliv

Kapitola 3

Agenti a multiagentní systémy

Problematiku řešenou v komunitě zabývající se oblastí multiagentních systémů je možné rozdělit do dvou hlavních kategorií:

- Hledání vhodné *vnitřní struktury jednotlivých agentů* – tedy jak navrhnout řídící aparát uvnitř agenta tak, aby navenek vykazoval požadované vlastnosti a chování. Tato oblast má blízko k tradiční umělé inteligenci, robotice a teorii řízení, ale čerpá například také z kognitivní psychologie a filozofie akce.
- Výzkum a návrh *chování kolekce agentů* – zde se agenti považují v podstatě za černé skříňky a zkoumají se jen jejich vnější projevy, zejména to, jak ovlivňují chování ostatních agentů a celého systému. Používají se přístupy věd společenských, matematické teorie her, ekonomie, lingvistiky apod.

Obě oblasti spolu samozřejmě úzce souvisí a vzájemně se ovlivňují. Výzkum prezentovaný v této práci spadá do první kategorie. Jednotlivé části této kapitoly tak mají za cíl nejprve ukázat obecné principy návrhu vnitřní struktury agenta, zejména přístupy alternativní ke zvolené architektuře, dále pak poskytnout stručný přehled a souvislosti s druhou kategorií výzkumů, které ovlivnily některé aspekty navrženého systému.

Pojem agenta, jakožto autonomní jednotky schopné interakce s ostatními agenty, prezentovaný v úvodní kapitole, nám posloužil pro první přiblížení. Označení *agent* je však používáno v mnoha souvislostech a významech, což je dáno zejména rozmanitostí vědních disciplín, v nichž se s tímto pojmem setkáváme. Některé z těchto disciplín vůbec nesouvisí s počítačovou vědou. To je příčinou, proč dosud neexistuje žádná všeobecně uznávaná definice. Jako příklad uveďme mimořádně úspěšnou knihu [RN02], kterou mnozí považují za základní dílo v oblasti umělé inteligence. Zde je agent používán jako centrální pojem umělé inteligence, resp. umělá inteligence je chápána jako věda o *vytváření inteligentních agentů*. Protože přístupy používané v umělé inteligenci jsou velmi různorodé, agent je chápan jako cokoliv, co vnímá své prostředí pomocí senzorů a ovlivňuje jej pomocí efektorů. Tato definice je však pro potřeby této práce příliš široká. V následující části se proto pokusím vymezit tento pojem společně s pojmem multiagentního systému tak, jak jej budu v této práci nadále užívat.

3.1 Základní pojmy

3.1.1 Inteligentní agent

Pojmem *agent* se budu nadále odkazovat na systém (obvykle softwarový nebo hardwarový), který se nachází v nějakém prostředí a vykazuje v něm tyto základní vlastnosti:

- *autonomnost* – agent má své dlouhodobé cíle, které sleduje, a snaží se dosáhnout těchto cílů bez trvalého dozoru člověka nebo jiného agenta,
- *reaktivita* – agent vnímá změny prostředí, ve kterém se pohybuje, a je schopen na ně adekvátně reagovat,
- *iniciativa* – agent sám iniciativně vykonává akce, kterými mění své okolí tak, aby se přiblížil svým cílům,
- *sociálnost* – agent umí komunikovat s ostatními agenty.

Z těchto vlastností vyplývá, že agent navenek vykazuje určitou dávku inteligence, proto bývá také označován jako *inteligentní agent*. Často se také setkáváme s pojmem *racionální agent*, racionalita v tomto případě znamená, že agent vykonává pouze akce, které v kontextu jeho prostředí a situace dávají smysl, tedy mu mohou pomoci zajistit splnění jeho cílů. Podrobnější úvahy na toto téma je možno nalézt v řadě publikací, například v [WJ95] nebo [Woo02].

3.1.2 Multiagentní systém

Obecný systém S je definován jako dvojice $S = (U, R)$, kde U značí množinu všech prvků systému a R množinu všech propojení mezi prvky tohoto systému.

V souladu s touto definicí chápeme *multiagentní systém* jako systém, ve kterém se vyskytují prvky aktivní – *agenti* a pasivní – *neagentní*, které spolu dohromady vytvářejí *prostředí*, ve kterém se agenti nacházejí. Aby byl systém multiagentní, měli by se v něm nacházet alespoň dva agenti. Ti mají své cíle a protože jsou obdařeni schopností iniciativy, aktivně působí na své okolí tak, aby se stav systému přiblížil takovému, jaký po nich jejich uživatelé požadují. Tyto akce mohou probíhat současně a každá z nich vyžaduje jistý čas. Je tedy možné, že akce agenta skončí neúspěchem přesto, že podmínky v době rozhodnutí o akci zaručovaly úspěch – podmínky se během vykonávání změnily.

3.2 Architektury agentů

K návrhu vnitřní struktury agenta lze přistoupit několika způsoby. Tradičním přístupem, vycházejícím z původních myšlenek umělé inteligence, je vybavit agenta *symbolickou reprezentací světa*, včetně ostatních agentů a jeho samého (například ve formě predikátové logiky prvního řádu) a *aparátem pro odvozování* nových poznatků. Takový agent pak uvažuje o svém prostředí, svých cílech a schopnostech. Na základě těchto úvah pak provádí akce, o kterých věří, že vedou ke splnění jeho cílů, pokud jsou tyto cíle dosažitelné. Hlavními problémy tohoto přístupu jsou celková obtížnost tvorby takového systému a časová složitost práce se symbolickou reprezentací. Agenti s touto vnitřní architekturou bývají nazýváni *uvažující* nebo také *rozvážní agenti* (*deliberative agents*).

V 80. letech minulého století vládlo mezi vědci zabývajícími se umělou inteligencí všeobecné zklamání z původních příliš optimistických vizí schopností systémů se symbolickou reprezentací. Jako odpověď vznikl proud robotiky, který prosazoval myšlenku, že inteligentní chování vzniká jen v očích pozorovatele, přičemž uvnitř systému žádná symbolická reprezentace světa být nemusí. Výsledné chování je pak pouze kombinací pevně daných bezprostředních reakcí na události, které se v okolí vyskytly. Tento přístup se stal základem pro takzvané *reaktivní agenty*.

Kombinací obou těchto dvou možností dostáváme *hybridního agenta*. Obvykle se jedná o *vrstvenou architekturu*, ve které jednoduché akce vyžadující rychlou odezvu zpracovává reaktivní vrstva, zatímco komplexní úlohy a plánování obstarává vrstva využívající symbolickou reprezentaci. Následující tři podkapitoly představují jednotlivé přístupy podrobněji.

3.2.1 Reaktivní agent

Jak již bylo řečeno, reaktivní agent neobsahuje žádnou symbolickou reprezentaci světa. Nejjednodušším příkladem takového agenta je čistě reaktivní agent. Ten neobsahuje naprosto žádnou reprezentaci svého vnitřního stavu. Příkladem takového agenta [Kel94] může být mechanická beruška – dětská hračka, která má za úkol jezdit po ploše stolu a nespadnout z něj. Beruška je v přední části vybavena dvěma tykadly (senzory), které jsou přímo napojeny na přední kolečko (efektor). Pokud beruška narazí na hranu stolu, jedno z jejích tykadel se sníží, což vede k otočení kolečka a tím změně směru. Beruška tak vykazuje dostatečně inteligentní chování, aby mohla sloužit ke svému účelu.

Charakteristickou vlastností reaktivních agentů je tedy poměrně přímé mapování vjemu na akci (resp. sekvenci akcí), které se dá u čistě reaktivního agenta vyjádřit funkcí chování:

$$akce : P \to A$$

kde P je množina vjemů, které agent rozeznává a A je množina akcí, které je agent schopen vykonat. Vjemy, které je agent schopen rozpoznat jsou určeny funkcí:

$$vjem : E \to P$$

kde E představuje množinu všech možných stavů agentova prostředí. Čistě reaktivní agent tedy může být popsán jako čtveřice $PRA = (P, A, vjem, akce)$. Vliv agenta na prostředí popisuje funkce:

$$vliv : A \times E \to E$$

Čistě reaktivní agent je schopen vykonávat pouze velice omezené množství úkolů. Přímé mapování vjemu na akci totiž umožňuje provádět jen jednoduché přímočaré akce. Tento problém lze částečně odstranit zavedením *modelu chování*. Agent stále pouze automaticky provádí akce, které jsou reakcí na podněty přijaté z prostředí, a o kterých nijak neuvažuje, avšak tyto akce jsou nějakým způsobem seřazeny do logického sledu, například pomocí konečného automatu. Formálně, každý přijatý vjem ovlivňuje agentův vnitřní stav, což vyjadřuje funkce:

$$stav : P \times I \to I$$

kde I je množina vnitřních stavů agenta. Na základě vnitřního stavu a přijatého vjemu pak agent provádí akce, což vyjadřuje upravená funkce akce:

$$akce2 : P \times I \to A$$

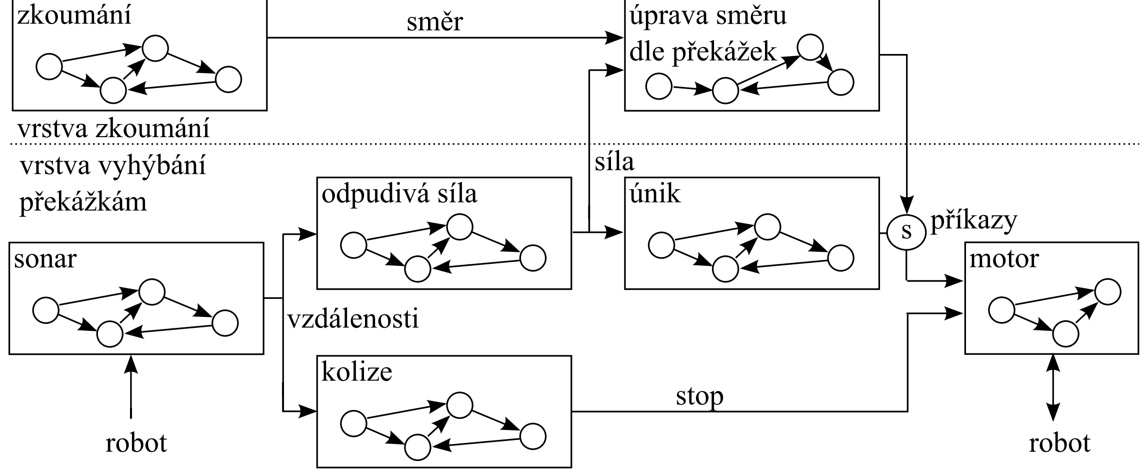
Reaktivní agent je pak šestice $RA = (P, A, I, vjem, stav, akce2)$.

Automaty řídící agenta se však při zavedení již několika málo složitějších vzorů chování stávají velmi rychle rozsáhlými a tedy nepřehlednými. Další problém nastává, pokud po takovém agentovi požadujeme, aby byl schopen provádět více akcí najednou. Proto se obvykle u netriviálních agentů zavádí takzvaná *vrstvená architektura*. Jednotlivé vrstvy se starají o různé vzory chování a obvykle mají možnost se nějakým způsobem vzájemně ovlivňovat. Tyto vrstvy bývají doplněny *mechanismem výběru akce*, ať už pomocí speciální *jednotky výběru akce* nebo vestavěných *priorit vrstev*.

Reaktivní architektury bývají podle tohoto principu vrstev nazývány jako přístup ke tvorbě agenta „zdola nahoru", protože dochází k dekompozici problému návrhu agenta podle jednotlivých požadovaných schopností, namísto klasické dekompozice na funkční jednotky. Výhodou tohoto přístupu je zejména to, že je možné poměrně snadno implementovat některé funkce agenta (například pouze pohyb), otestovat je a později v případě potřeby doplňovat další funkčnost přidáváním dalších vrstev pro požadované vzory chování.

Klasický příklad takovéto vrstvené architektury představuje *subsumpční architektura* [Bro86], navržená v 80. letech minulého století prof. Brooksem, působícím na MIT. Brooks patřil k hlavním kritikům klasického přístupu k umělé inteligenci za použití symbolické reprezentace. Inspirací při tvorbě tohoto modelu pro něj bylo chování nižších živočichů. Jeho architektura obsahuje vrstvy, přičemž každá vrstva se skládá z modulů. Chování každého z modulů je řízeno pomocí konečného automatu. Některé z modulů jsou připojeny přímo na senzory, další pak na aktuátory. Jednotlivé moduly mají definované vstupy a výstupy, které jsou vzájemně propojeny. Princip připojení dalších vrstev spočívá v připojení vstupů a výstupů některých nových modulů na existující propojení a možnost změnit vstup modulů původních vrstev (*potlačení*), případně blokovat výstup jiných (*znehybnění*).

Schéma subsumpční architektury použité pro řízení mobilního robota ukazuje obrázek 3.1. V tomto případě obsahuje architektura dvě vrstvy, spodní vrstva má za úkol vyhýbání se překážkám, včetně úniku od pohyblivých překážek, které robota „honí", zatímco horní vrstva se stará o náhodné procházení prostoru.



Obrázek 3.1: Subsumpční architektura použitá pro řízení mobilního robota

Agenti založení na reaktivní architektuře mají celou řadu výhod: jednoduchost, ekonomičnost, robustnost proti poruše, atd. Mají však také celou řadu dosud nevyřešených problémů:

- zatímco tvorba agentů s několika málo vrstvami je poměrně jednoduchá, návrh agenta s větším počtem vrstev je *výrazně* obtížnější – interakce jednotlivých vrstev se stanou příliš složité na to, aby se daly efektivně zvládnout.

- Výsledné inteligentní chování agenta vzniká kombinací chování jednotlivých vrstev a prostředí. Proto je velmi těžké pro takový návrh vytvořit obecný postup či metodiku – při vývoji je tedy nutné pokaždé projít zdlouhavým procesem experimentů, pokusů a chyb.

- Limitujícím faktorem reaktivních agentů je neschopnost plánování dalších akcí. Při pokusech s relativně jednoduchými agenty, jejichž cílem je například náhodně procházet místnosti a sbírat odpad [Con89] dosahují reaktivní agenti obvykle minimálně srovnatelných výsledků jako agenti se symbolickou reprezentací světa, velmi často s výrazně menšími nároky na výpočetní systém. Jsou však úkoly, které nelze reaktivnímu agentovi zadat [Kub04] – například aby se přesunul na nějaké konkrétní místo v laboratoři a tam vysbíral všechny plechovky. Agent totiž nemá žádnou představu o okolním světě, kterou by pro provedení takového úkolu potřeboval.

3.2.2 Uvažující agent

Jestliže přístup ke tvorbě reaktivního agenta se dá popsat jako „zdola nahoru“ – návrhář provádí dekompozici agenta podle úkolů, které má agent plnit, u uvažujícího agenta je to vždy „shora dolů“. Celý systém se rozloží na jednotlivé funkční bloky (senzory, báze znalostí, jednotka pro odvozování, plánování, efektory apod.), navrhnou se jejich vazby a teprve po implementaci všech částí je systém provozuschopný – nezávisle na složitosti akcí, které od agenta požadujeme.

Uvažující agent obsahuje jako jednu z klíčových komponent symbolickou reprezentaci světa, požadovaného chování a svých akcí. Nad touto symbolickou reprezentací pak agent provádí syntaktické manipulace. Reprezentací mohou být například formule predikátové logiky prvního řádu a syntaktická manipulace pak odpovídá logické dedukci, resp. dokazování teorémů. Označme L množinu všech vět predikátové logiky prvního řádu a $D = \wp(L)$ množinu databází (množin) vět L. Prvky množiny D označíme $\Delta, \Delta_1, \dots$. Rozhodování agenta je modelováno pomocí množiny ρ odvozovacích pravidel, zápis $\Delta \vdash_\rho \varphi$ značí, že φ může být odvozeno z Δ pouze pomocí ρ.

Uvažujícího agenta můžeme popsat obdobně jako reaktivního pomocí funkcí $vjem$, $stav$ a $akce$. Funkce $vjem$ je zcela shodná jako u reaktivního agenta:

$$vjem : E \rightarrow P$$

Funkce $stav$ je obdobná jako u reaktivního agenta, vnitřní stav je však dán agentovou reprezentací světa:

$$stav : P \times D \rightarrow D$$

Akce je pak zvolena na základě stavu vnitřní reprezentace:

$$akce : D \rightarrow A$$

Uvažující agent je popsán opět jako šestice $DA = (P, A, D, vjem, stav, akce)$. Zjevně klíčová je funkce $akce$, která na základě aktuálního stavu reprezentace rozhoduje, jakou akci z množiny A zvolit. Toho je možné dosáhnout tak [Woo02], že pro každou akci vytvoříme odvozovací pravidla tvaru $\varphi(...) \rightarrow \psi(...)$, kde φ a ψ jsou predikáty (které mohou obsahovat

volné proměnné, jež jsou při aplikaci pravidla navázány). V nejjednodušším případě bude φ popisovat stav databáze a ψ akci, která se má provést. Příkladem může být pravidlo:

$$PoziceRobota(x, y) \wedge PoziceOdpadu(x, y) \rightarrow Do(zvedniOdpad)$$

Tedy, nachází-li se robot na stejné pozici jako odpad, má provést akci *zvedniOdpad*. Funkce *akce* se pak snaží pro každou akci $\alpha \in A$ odvodit $Do(\alpha)$. Pokud $\Delta \vdash_\rho Do(\alpha)$, pak bude akce α vykonána.

Jako příklady projektů, které se dají zařadit do této kategorie, jmenujme systémy Concurent MetateM [Fis94] a AOP [Sho90, Sho93]. Přístup založený na logice je elegantní a poskytuje jasnou sémantiku, proto je pro mnoho vědců velmi lákavý. Tento přístup má však také celou řadu problémů, které daly vzniknout mnoha podoborům klasické umělé inteligence (rozpoznávání obrazu, reprezentace znalostí, automatické odvozování apod.). Hlavním problémem však zůstává inherentní časová náročnost dokazování vět, která způsobuje u mnohých pochybnosti, zda se někdy podaří sestrojit agenta s touto architekturou pracujícího v netriviálním časově omezeném prostředí.

Systémy založené na praktickém usuzování

Na rozdíl od *teoretického usuzování* (theoretical reasoning), kde je cílem odvozování rozhodnout, zda je či není daná formule pravdivá, *praktické usuzování* (practical reasoning) se zabývá tím, jaká akce se má vykonat, aby se nějaká formule pravdivou stala. Praktické usuzování se skládá ze dvou hlavních částí – *zvažování* (deliberation) a *plánování* (means ends reasoning, planning). Ve fázi zvažování se volí cíl, kterého se má dosáhnout. Výsledkem této fáze je *záměr*. Ve fázi plánování se pak hledá vhodná akce, či sekvence akcí, která dosažení záměru zajistí. Intuitivně tento přístup odpovídá způsobu, jaký používáme k řešení problémů my, lidé. Vývoj systémů založených na praktickém usuzování skutečně čerpá z oborů, jako je *kognitivní psychologie* (cognitive psychology), *filozofie akce* (philosophy of action) a *teorie racionálního výběru* (rational choice theory).

Jedním z pramenů pro oblast zvažování byla práce filozofa D. Dennetta [Den87], který tvrdil, že člověk je charakterizován svou *intencionalitou* (intentional stance), schopností sebereflexe a přisuzování intencionality jiným jedincům. Při popisování chování jedinců používáme často věty jako:

Karel trénoval, protože *chtěl* vyhrát a *věřil* že to dokáže.

Tento přístup staví na *naivní psychologii* (folk psychology), která umožňuje vysvětlovat a předvídat lidské chování tím, že se člověku přiřadí atributy jako *představy o světě*, *touhy*, *preference* apod. Stejný přístup se ukázal být užitečný při popisu inteligentních agentů.

Klíčovým pojmem pro zvažování je *záměr*. Problematikou záměrů se zabýval zejména filozof M. Bratman [Bra87]. Záměr představuje cíl, kterého agent hodlá dosáhnout. Záměr má přetrvávající, perzistentní charakter. Agent nemůže od záměru upustit, pokud k tomu nemá „dobrý důvod". Situace, které umožňují agentu odložit záměr jsou v podstatě dvě:

- Agent si je vědom, že záměr byl splněn, to znamená, že dosáhl cíle, který odpovídal tomuto záměru.

- Agent zjistil, že záměru nikdy za žádných okolností nemůže být dosaženo.

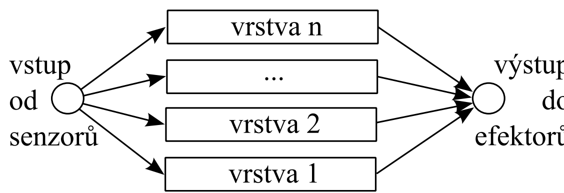
Záměry tvoří jednu z klíčových komponent architektury BDI a proto se jimi budu podrobněji zabývat v následující kapitole věnované této architektuře.

Druhý krok – hledání sekvence akcí, která zajistí splnění zvoleného záměru, neboli *plánování* – patří mezi klasické disciplíny umělé inteligence. Zřejmě prvním a dodnes nejznámějším plánovacím programem byl STRIPS [FN71]. Problematika plánování je extenzivně pokryta v [RN02]. Přes velký pokrok, který se v této oblasti podařilo dosáhnout, zůstává hlavním problémem plánování časová složitost, zejména v prostředích, která se rychle mění a je tedy potřeba plány v průběhu jejich provádění revidovat. Z tohoto důvodu se systémy realizující praktické usuzování často vybavují „knihovnou plánů“, která obsahuje parametrizované šablony plánů, společně se situacemi, ve kterých je vhodné tyto plány použít. Stejně tomu je u frameworku prezentovaného v této práci, proto se zde klasickým plánováním nebudu zabývat.

Zřejmě nejznámější architekturou založenou na praktickém usuzování je architektura BDI, které je věnována celá kapitola 4, příklady projektů a zhodnocení proto ponechám do této kapitoly.

3.2.3 Hybridní agent

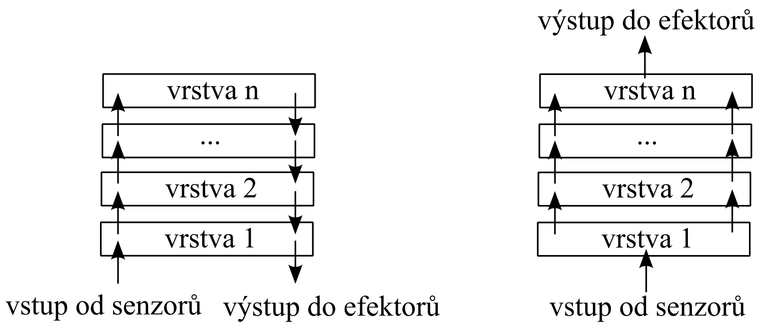
Agenti s reaktivní architekturou jsou schopni rychlé odezvy, avšak mají problémy se zvládáním komplexních úkolů. Uvažující agenti mají problémy s rychlou odezvou na jednoduché podněty, protože používají aparát, který umožňuje zvážit všechny alternativy a vybrat optimální akci, což se hodí spíše pro komplexní úkoly. Proto jsou vyvíjeny architektury, které v sobě zahrnují obě paradigmata, případně i některá další. Obvykle se jedná o *vrstvené architektury*. Vrstvy mohou být propojeny buď *horizontálně* nebo *vertikálně*. U horizontálního vrstvení (viz obrázek 3.2) dostávají všechny vrstvy stejný senzorický vstup a mají k dispozici přístup k efektorům. V tomto případě musí existovat nějaký řídící mechanismus, který rozhodne, která vrstva bude v daném okamžiku kontrolovat agenta. Příkladem hybridní architektury s horizontálním vrstvením je projekt TouringMachines [Fer92].



Obrázek 3.2: Schéma horizontálně vrstvené architektury

V případě vertikálního vrstvení (viz obrázek 3.3) je se senzory a efektory obvykle spojena jen jedna vrstva. Podle toku dat a řízení rozlišujeme vrstvení s *jednoprůchodovým řízením* (na obrázku 3.3 vpravo, o akci rozhoduje horní vrstva, zatímco spodní vrstva dostává senzorický vstup) a *dvouprůchodovým řízením* (na obrázku 3.3 vlevo, vstup od senzorů i příkazy efektorům posílá spodní vrstva).

Příkladem takovéto architektury je InteRRaP (Integration of Reactive Behaviour and Rational Planning) [Mul97], která obsahuje tři vrstvy. Nejnižší *reaktivní vrstva* vykonává reakce na události, které vyžadují rychlou odezvu, jako je vyhýbání se překážkám. Tyto události jsou určeny takzvaným *popisem situací*. Události, které nezapadají do žádného vzoru takového popisu jsou předány vyšším vrstvám. Druhá vrstva se nazývá *vrstva lokálního plánování*, a je zodpovědná za situace, které vyžadují uvažování, případně tvorbu plánu. Nejvyšší vrstva, *vrstva kooperativního plánování*, odpovídá za řešení konfliktů s ostatními agenty a za koordinaci plánů při řešení společného úkolu.



Obrázek 3.3: Schéma vertikálně vrstvené architektury

Výhodou hybridních agentů je jejich schopnost reagovat téměř srovnatelně rychle jako reaktivní agenti a zároveň vytvářet plány jako uvažující agenti. Hlavní nevýhodou je pak větší komplexnost celého systému a přídavná režie na synchronizaci jednotlivých vrstev.

3.3 Prostředí a interakce v multiagentních systémech

Již při vymezení pojmu *agent* jsme uvedli, že se jedná o systém, který se nachází v nějakém *prostředí*. Prostředí je tedy pro agenta velmi důležité, především proto, že představuje jeho sféru působnosti a zdroj podnětů, bez kterých by jeho existence neměla smysl (svázanost agenta s jeho prostředím ho mimo jiné odlišuje od expertních systémů [Woo02]). Toto prostředí však může být agenty také využíváno, například pro komunikaci. Správné využití prostředí může u agentů výrazně snížit náročnost jejich vývoje. Výzkumem na toto téma se zabývá v současné době celá řada vědců, dokonce v rámci komunity zabývající se multiagentními systémy vznikl pojem „environmental engineering“. Dobrý přehled o možnostech a současném stavu využívání prostředí při tvorbě agentních systému podává D. Weyns v článku [WOO07].

Do agentova prostředí však obvykle počítáme i výpočetní systém, v rámci kterého agent „běží“. U softwarového agenta by měl tento výpočetní systém agentovi poskytovat potřebné prostředky pro jeho chod, správu životního cyklu, vyhledávání ostatních agentů a zasílání zpráv. Tato oblast je velmi důležitá pro spolupráci heterogenních agentů, proto byla také jednou z prvních oblastí, které se dotkla standardizace. V dnešní době tak již existují normy [FIP02a, FIP04] pro to, jaké služby by mělo prostředí agentům poskytovat, jak by tyto služby měly být pojmenovány a jaké rozhraní by měly mít. Standardizaci v oblasti agentních technologií je věnována samostatná podkapitola dále v textu.

Jak již bylo řečeno, v prostředí se typicky nacházejí další agenti. Na rozdíl od problematiky řešené v klasických distribuovaných systémech má každý z těchto agentů *své vlastní úkoly a cíle*, které mu zadal jeho uživatel. Tyto cíle mohou být u dvou agentů v jednom extrémním případě shodné, ve druhém extrému zcela protichůdné. Případy mezi těmito dvěma extrémy dávají prostor k *vyjednávání*. Agent typicky nemá přístup k celému prostředí – obvykle může vnímat a ovlivňovat pouze jeho část. V tomto případě může být *kooperace* s dalšími agenty nezbytná. Velká část prací o multiagentních systémech se tedy zabývá tím, jak řešit *konflikty* mezi agenty a naopak jak mají agenti vyjednávat pro dosažení *kooperace* a jak tuto kooperaci posléze *koordinovat*.

Jedním z možných přístupů, jak k analýze chování agentů v rámci celého systému

přistoupit, je použít *matematickou teorii her*. Akce, které může agent vykonat se v pojmech teorie her nazývají *strategie*. Každý agent má své *preference* o tom, v jakém stavu by se prostředí mělo nacházet. Tyto preference se vyjádří funkcí, která každý stav ohodnotí reálným číslem. Na základě toho se dají strategie porovnávat a je možné analyzovat, jak by se měl agent v dané situaci zachovat. Klasickou ukázkou toho přístupu je *vězňovo dilema* [RN02].

Pokud dva agenti nemají zcela protichůdné cíle, mohou dosáhnout *oboustranně výhodné dohody*. Pro uzavírání takových dohod však musí existovat mechanismy – *protokoly*, které definují jakým způsobem má vyjednávání probíhat. V prostředí multiagentních systémů si velkou oblibu získaly zejména různé druhy *aukce* [Woo02], které se používají především pro přidělování zdrojů či rozdělování práce. Aby protokol mohl plnit svou funkci v prostředí heterogenních agentů, musí být všem návrhářům známa jeho přesná specifikace. Proto byly také pro tuto oblast vyvinuty mezinárodní standardy (např. [FIP02b]).

Problémy spolupráce agentů nastíněné v této části se vlastního návrhu architektury agenta, která je hlavním tématem předložené práce, dotýkají spíše okrajově. Proto zde nebyly rozebírány detailně, důležité však jsou dva poznatky:

- agent musí být vybaven schopnostmi komunikace na poměrně *vysoké úrovni*,
- musí být schopen *uvažovat* o akcích zaslání a příjmu zprávy obdobně jako o všech svých ostatních (fyzických) akcích.

3.4 Komunikace v multiagentních systémech

Komunikace mezi agenty může mít celou řadu podob. Například mezi nejjednodušší patří reaktivní komunikace, která bývá založena na zanechávání značek v prostředí, což představuje obdobu chování hmyzu žijícího ve společenstvích [BF89].

U agentů se symbolickou reprezentací světa však tato komunikace probíhá typicky na daleko vyšší úrovni. Základ pro tento způsob komunikace tvoří *teorie řečového aktu* (speech act theory) [Aus75], ve které se ke komunikaci přistupuje jako k provádění akcí. Každé zaslání zprávy tak má kromě informačního obsahu zprávy za cíl u adresáta vykonat nějakou změnu v jeho kognitivním stavu. Na základě toho se dají z komunikace abstrahovat různé druhy zpráv (oznámení, požadavek, příkaz, apod.). Z této teorie vycházejí vysokoúrovňové jazyky pro komunikaci mezi agenty.

Jazyk *KQML* (Knowledge Query Manipulation Language) [FFMM94] byl vyvinut jako součást projektu agentury DARPA za účelem výměny informací a znalostí mezi autonomními informačními systémy. Jeho syntaxe je velmi podobná jazyku LISP. Je založen na zprávách, každá zpráva odpovídá jednomu řečovému aktu. Skládá se z *performativu*, následovaného volitelným počtem *atributů*. Performativ určuje typ zprávy, tedy ve většině případů jakému řečového aktu zpráva odpovídá. Každý atribut má tvar dvojice, přičemž první člen představuje název atributu a druhý jeho hodnotu. Zpráva tedy může vypadat například takto:

```
(inform
  :sender book-seller-agent
  :reciever client-agent
  :content "price(isaac-asimov-foundation, 25)"
  :language Prolog
)
```

Performativ této zprávy je *inform*, tedy význam zprávy je takový, že agent identifikovaný atributem *:sender* zasílá informaci agentovi specifikovaného atributem *:receiver*. Vlastní obsah zprávy je zapouzdřen v atributu *:content*. Tento obsah může být zapsán v libovolném jazyce, v rámci KQML označovaného jako *obsahový jazyk*. Identifikace obsahového jazyka je specifikována atributem *:language*.

Jazyk KQML si získal poměrně značnou oblibu, stal se však také terčem celé řady kritik. Problémy způsobuje zejména to, že sémantika jazyka nebyla formálně definována, což vede k různému pochopení u jednotlivých implementací a následné nekompatibilitě. Navíc postupně vznikla celá řada verzí jazyka, které obsahovaly různé množiny performativů. Tato množina se také mnohým zdála příliš velká (v původní verzi přibližně 40 performativů) a navržená *ad hoc*.

Tato kritika byla podnětem pro vznik jazyka FIPA ACL [FIP01], který je syntakticky prakticky shodný s KQML, avšak má formálně definovanou sémantiku (pomocí jazyka FIPA SL, který vychází z práce [BS97]) a pevně danou množinu dvaceti performativů, jejichž výběr byl proveden na základě pečlivé analýzy potřeb, které vyvstaly při používání KQML. Jazyk FIPA ACL představuje v dnešní době mezinárodní standard [FIP01], proto se s ním v návrhu frameworku prezentovaného v této práci počítá jako s jazykem pro komunikaci (v implementovaném prototypu se používá jeho podmnožina).

Jazyky KQML i FIPA ACL předpokládají použití dalšího jazyka pro výměnu vlastního obsahu zpráv. Z hlediska návrhu frameworku není výběr takového jazyka příliš podstatný, protože tento jazyk se obvykle volí v závislosti na konkrétní aplikaci. Pro jednoduché příklady je například možné používat přímé předávání hodnot v atributu zprávy :content, nebo použít predikáty jazyka Prolog, jak tomu bylo u ukázky zprávy KQML výše. V prostředí, kde se pohybuje celá řada heterogenních agentů, kteří byli vytvořeni různými institucemi, je však potřeba shodnout se na interpretaci obsahu zprávy stejně důležitá, jako na jazyku pro komunikaci. Tento problém se snaží řešit koncept *ontologií*. Ontologie je formální, explicitní specifikace vyjádření konceptu zprávy. Ontologie tedy udává význam symbolů ve zprávě [Gru93]. Aby mohly být ontologie plně využívány při komunikaci heterogenních agentů, musí být umístěny na nějakém veřejně dostupném úložišti. Řešením jsou obecné ontologické slovníky a ontologické servery, což jsou místa, kde jsou ontologie definovány jako domény. Komunikující agenti mohou tato místa využívat a při komunikaci uvádět ontologie vztažené k zasílaným zprávám.

3.5 Standardizace agentních technologií – organizace FIPA

Organizace FIPA[1] – Foundation for Intelligent Physical Agents je nezisková organizace, jejím cílem je vytvářet průmyslové standardy pro problematiku agentů a multiagentních systémů za účelem podpory znovupoužití kódu a interoperatibility. Byla založena ve Švýcarsku v roce 1996, v roce 2005 se stala součástí IEEE Computer Society (jako jedna ze standardizačních komisí). Většina jejích norem byla přijata v období kolem roku 2002, po vstupu do asociace IEEE se začala zabývat také standardizací napojení agentních systémů na neagentní technologie. V současné době (rok 2008) má FIPA 44 členů, členská základna je tvořena převážně firmami zabývajícími se agentními technologiemi (mezi jinými Boeing, Siemens, Toshiba a další) a univerzitami.

Normalizace se zaměřuje na následující oblasti:

[1] http://www.fipa.org

- *Abstraktní architektura* – normalizuje obecné požadavky na komponenty agentního systému, tyto normy tvoří základ všech ostatních norem.
- *Správa agentů* – tyto normy specifikují životní cyklus agenta, definují základní principy identifikace agentů a podpůrných služeb pro jejich běh.
- *Meziagentní komunikace* – standardizuje strukturu zpráv, ontologie, interakční protokoly, obsahové jazyky apod.
- *Přenos zpráv* – tyto normy se zabývají tím, jak má probíhat výměna zpráv v různých prostředích, zejména Internetu.
- *Aplikační oblasti agentů* – normy v této oblasti jsou většinou ve vývoji, zaměřují se na to, jaké služby by měl agent poskytovat v dané aplikační doméně (např. osobní asistent) a jak by měl s ostatními agenty komunikovat, tedy zejména doporučené ontologie.

V dnešní době by tak každá multiagentní aplikace, u které se počítá s jiným než čistě akademickým či prototypovým použitím měla splňovat alespoň základní koncepty definované v těchto normách. Zjevně nejdůležitější je otázka komunikace pomocí zasílání zpráv – výměna zpráv v síťovém prostředí je totiž nejtypičtějším způsobem interakce heterogenních agentů.

Kapitola 4

Architektura BDI

Architektura BDI představuje v dnešní době zřejmě nejrozšířenější přístup k tvorbě agentů založených na praktickém usuzování. Zkratka BDI pochází z anglických slov *Beliefs* (představy), *Desires* (touhy) a *Intentions* (záměry), které společně s plány tvoří základní prvky této architektury. Na architekturu BDI lze nahlížet ze tří úhlů pohledu:

- *Filozofického:* filozofický základ této architektury položil M. Bratman v knize [Bra87], ve které se snažil modelovat lidské praktické usuzování. Lidé jsou nazíráni jako plánující agenti, kteří používají představy (to co ví o světě), touhy (to co chtějí, touhy mohou být ve vzájemném rozporu) a záměry (to co se rozhodli dosáhnout, záměry musí být na rozdíl od tužeb konzistentní). Jeho práce vychází z *naivní psychologie* (folk psychology), která zavádí potřebné koncepty jako jsou představy, touhy apod.

- *Formálního (logického):* konceptům představ, tužeb a záměrů je v tomto přístupu přiřazen přesný logický význam. Používané logiky vycházejí z modálních logik se sémantikou možných světů.

- *Implementačního:* reálné aplikace navazují na filozofický a logický přístup poměrně volně. Představy jsou obvykle omezeny na formu databáze, cíle jsou modelovány jako události v systému a záměry odpovídají zásobníkům právě prováděných plánů. Tato odtažitost reálných aplikací od formální teorie vedla k mnoha pokusům o alternativní formalizaci realizovaných aplikací.

V následujících částech se budu postupně zabývat jednotlivými přístupy podrobněji – nejprve se zaměřím na základní komponenty architektury, následně ukáži východiska pro formalizaci a nakonec několik implementovaných systémů společně s pokusy o jejich alternativní formální popis.

4.1 Základní principy a komponenty architektury BDI

4.1.1 Představy

Představy reprezentují agentovy *znalosti*. Tyto znalosti se typicky týkají prostředí, ve kterém se agent nachází, jeho schopností, ostatních agentů v systému či historie jeho akcí – tedy v zásadě jsou to poznatky o *stavu světa*. Znalosti o tom, jak tento stav *měnit* jsou reprezentovány procedurálně ve formě plánů, kterým je věnována samostatná část dále v textu. Co bylo důvodem pro zavedení pojmu *představa*, namísto pojmu *znalost*, běžně

používaného v klasické umělé inteligenci? Informace, které má agent k dispozici, nemusí být nutně pravdivé (zejména v kontextu dynamického prostředí, popřípadě špatně fungujících agentových senzorů). Může se tedy stát, že agent vytvoří špatný plán či provede nesmyslnou akci kvůli tomu, že vycházel z chybných předpokladů. Cílem zavedení pojmu představa je tedy zdůraznit *subjektivitu* informací, které jsou agentovi dostupné.

Z hlediska filozofického má zcela jistě význam zabývat se *meta-představami*, tedy představami o představách. Prof. Dennett [Den87] dokonce dělil systémy podle jejich schopností mít tyto „představy vyšších řádů" – systém, který měl představy, přání a další mentální stavy, nazýval *intencionálním systémem prvního řádu*. Systém který měl představy, přání a další mentální stavy o představách a přáních svých a ostatních agentů nazýval *intencionálním systémem druhého řádu*. *Intencionální systémy třetího řádu* mohly mít představy o meta-představách atd.

Schopnost modelovat představy a přání ostatních agentů je zjevně klíčová pro předvídání jejich chování. Aby však bylo možné takové závěry korektně odvozovat, musí pro to být k dispozici formální aparát. Jistých pokroků bylo v této oblasti dosaženo, avšak časová náročnost a složitost implementace takovýchto systému činí tento přístup v současné době pro praktické použití nereálným. Současné implementace se tak obvykle omezují na intencionální systémy prvního řádu. Představy jsou pak obvykle skladovány v jistém druhu databáze, která je často nazývána *báze představ* (belief base). Hlavní problémy návrhu báze představ se týkají toho, jakým způsobem by měly být znalosti reprezentovány a jak by měly být aktualizovány – například jakým způsobem by měl agent řešit konflikt mezi informacemi ze senzorů a své báze představ. Přístupů k této problematice je celá řada, krátce se o nich zmíním u jednotlivých konkrétních implementací.

4.1.2 Touhy

Touhy (v některých systémech a u některých autorů nazývané cíle) reprezentují motivaci agenta. Vyjadřují objekty, informace nebo situace, kterých se agent snaží dosáhnout. Ne všechny touhy musí být v daném okamžiku splnitelné. Používání pojmu cíl namísto touha obvykle poněkud zužuje význam tohoto konceptu v tom smyslu, že cíle by měly být konzistentní. Agent by například neměl mít cíle jít dnes večer do kina a zůstat dnes večer doma, přestože by obě tyto možnosti pro něj byly lákavé.

4.1.3 Záměry

Pojem záměr vyjadřuje odhodlání agenta dosáhnout zvolené touhy či cíle. Představuje ústřední pojem v práci prof. Bratmana [Bra87]. Bratman o záměrech formuloval celou řadu bodů:

- Záměr představuje *zadání problému*, agent se snaží nalézt způsob, jak jej dosáhnout.
- Záměr představuje pro agenta *mez přípustnosti* (screen of admisibility) pro přijímání dalších záměrů.
- Agent si uchovává záznam o tom, zda jeho pokusy o dosažení záměru byly úspěšné.
- Agent věří, že je záměr možné splnit.
- Agent nevěří, že se nikdy v budoucnosti nemůže přiblížit k dosažení záměru.
- Agent věří, že se za určitých podmínek přiblíží k dosažení záměru.

- Agent nemusí mít v úmyslu všechny vedlejší účinky, které nastanou při dosahování záměru.

Pokud agent záměr jednou přijme, nemůže od něj upustit, pokud nenastane jedna ze dvou situací:

- Agent věří, že dosáhl záměru, tedy splnil cíl.
- Agent zjistí, že záměr již nikdy v budoucnu nebude možné splnit.

Agent samozřejmě může sledovat více záměrů najednou, typicky jednotlivým záměrům přiřazuje různé priority, případně dočasně odkládá záměry, pro jejichž dosažení není momentálně schopen sestavit plán.

4.1.4 Plány

Plány představují čtvrtý základní koncept architektury BDI, zkratka by tak měla znít spíše „BDIP". Zatímco ve formalizaci architektury BDI jsou zmiňovány spíše okrajově, v praktických realizacích jsou považovány za ústřední pojem – představují procedurální reprezentaci agentových znalostí o tom, jak měnit své prostředí tak, aby dosahoval svých cílů. Plány mohou být hierarchické, tj. obsahovat další plány. Stejně tak mohou být pouze částečně konkretizované (obsahovat podcíle nebo proměnné). Unifikace se pak v tomto případě provádějí při inicializaci, případně v průběhu vykonávání plánu. Konkrétní struktura plánů se mezi jednotlivými implementovanými systémy liší, některé charakteristické rysy jsou však společné. Plán typicky sestává z hlavičky a těla. Hlavička obsahuje cíl, který plán umožňuje dosáhnout, a množinu podmínek, které musí být splněny, aby byl plán v dané situaci aplikovatelný. Tělo sestává ze sekvence atomických akcí, které je agent schopen vykonat. Kromě těchto akcí se obvykle povoluje vyvolání podcílů a testování báze znalostí.

Tyto plány, resp. spíše šablony plánů, jsou předem vytvořeny programátorem dané aplikace a bývají uloženy v *knihovně plánů*. Při výskytu nějaké události se pak v této knihovně hledají plány, které představují vhodnou reakci na danou událost. Agenti s architekturou BDI tak typicky neprovádějí žádné plánování v klasickém slova smyslu (planning from first principles). Důvodem pro to je zejména časová náročnost takového plánování, která by způsobovala příliš dlouhé prodlevy v odezvě. Typickým použitím BDI agentů jsou totiž prostředí, kde jsou časová omezení značná a prostředí se rychle mění. V takových prostředích není klíčová schopnost sestavit optimální plán, ale spíše schopnost rozpoznat, kdy daný plán již nemá smysl a zkusit jiný přístup, případně celý záměr odložit. Ukázky jednotlivých konkrétních realizací plánů budou následovat u popisu příslušných systémů.

4.2 Formalizace BDI

Pro architekturu BDI bylo vytvořeno několik formálních popisů. Dnes již klasický popis představuje logika nazývaná BDI CTL, která kombinuje multimodální logiku s temporální logikou CTL*. Autory této logiky jsou prof. Rao a prof. Georgeff [RG92]. Na tuto práci navázal prof. Wooldridge se svojí Logic of Rational Agents (LORA) [Woo01]. Logiku BDI CTL společně s potřebnými souvislostmi v následující podkapitole velice stručně představím. Alternativní formalizace BDI vychází ze situačního kalkulu [Rei01].

4.2.1 Modální logiky

Základ pro všechny logiky, které budou zmíněny dále, představují modální logiky. Modální logiky umožňují popisovat koncepty jako jsou možnost, nemožnost či nutnost. Oproti logikám klasickým přidávají do jazyka dva nové operátory $\Box$ – operátor nutnosti a $\Diamond$ – operátor možnosti. Tyto operátory jsou vzájemně duální, tedy $\Box\varphi \Leftrightarrow \neg\Diamond(\neg\varphi)$ a naopak.

Modální logiky používají odlišnou sémantickou interpretaci než logiky klasické. Tato interpretace je založena na *teorii možných světů* [Hin62]. Hlavní myšlenkou této teorie je, že na rozdíl od sémantiky klasické logiky, kde bereme v úvahu jen jeden svět (ten reálný) a uvažujeme co je a co není v tomto světě pravda, zde pracujeme s množinou „virtuálních" světů, přičemž daná formule může být v některých těchto světech pravdivá, v jiných nikoliv. Tyto světy jsou spolu spojeny takzvanou *relací přístupnosti*, která určuje, jak spolu jednotlivé světy „souvisí". Formální rámec pro tyto pojmy se nazývá *Kripkeho struktura*, což je čtveřice $M = (S, S_0, R, L)$, kde S je spočetná množina stavů, $S_0 \subseteq S$ množina počátečních stavů, $R \subseteq S \times S$ totální relace, nazývaná relace přístupnosti a $L : S \rightarrow 2^{AP}$ pravdivostní zobrazení (AP je množina atomických výroků), jež určuje, které atomické výroky jsou v jednotlivých stavech pravdivé. Pomocí relace R je pak nadefinována sémantika operátorů $\Box$ a $\Diamond$. Například $\Box\varphi$ ve stavu (světě) S platí, právě když formule φ platí ve všech stavech, které jsou ze světa S přístupné.

Modální logiky dále zavádí nové odvozovací pravidlo, tzv. pravidlo nutnosti $\varphi \Rightarrow \Box\varphi$ a 5 axiomů:

K: $\Box(\varphi \Rightarrow \psi) \Rightarrow \Box\varphi \Rightarrow \Box\psi$

T: $\Box\varphi \Rightarrow \varphi$

D: $\Box\varphi \Rightarrow \Diamond\varphi$

4: $\Box\varphi \Rightarrow \Box\Box\varphi$

5: $\Box\varphi \Rightarrow \Box\Diamond\varphi$

Tyto axiomy odpovídají v sémantické stránce různým vlastnostem relace přístupnosti, například axiom 4 odpovídá tranzitivitě R. Podle toho, které axiomy jsou použity, dostáváme různé varianty modálních logik. Za základní bývá považována modální logika založená na axiomech K a T. Výborný úvod do problematiky modálních logik poskytuje kniha prof. Peregrína [Per04].

4.2.2 Temporální logiky

Z logik modálních vycházejí logiky temporální. Temporální logiky zavádí sadu operátorů, které vymezují platnost formulí s ohledem na čas. Nejznámější jsou zřejmě logiky LTL (Linear Temporal Logic) a CTL/CTL* (Computational Tree Logic). Zatímco LTL pracuje s lineárním časem, CTL a CTL* umožňují větvení času v budoucnosti. Sémantika těchto logik je opět dána Kripkeho strukturou – relace R je zde chápána jako vyjádření časové posloupnosti jednotlivých stavů, které tak tvoří takzvané „cesty".

Logika CTL zavádí následující operátory (s neformálním popisem sémantiky):

- Operátory cesty:
  - A φ – All: φ musí platit na všech cestách, které vycházejí ze současného stavu.

– E φ – Exists: φ platí alespoň na jedné cestě vycházející ze současného stavu.

- Operátory stavu:
  - N φ – Next: φ musí platit v následujícím stavu.
  - G φ – Globally: φ musí platit v celé cestě vycházející ze současného stavu.
  - F φ – Finally: φ musí platit v nějakém stavu nacházejícím se na cestě vycházející ze současného stavu.
  - φ U ψ – Until: φ musí platit, dokud v nějakém stavu nebude platit ψ.
  - φ W ψ – Weak until φ musí platit, dokud v nějakém stavu nebude platit ψ, přičemž toto na rozdíl od předchozího případu nemusí nastat.

Rozdíl mezi logikami CTL a CTL* spočívá v tom, že v logice CTL se musí dohromady nacházet vždy dvojice operátorů: jeden operátor cesty následován operátorem stavu, zatímco v CTL* toto omezení neplatí. Mezi CTL* a CTL i LTL platí vztah CTL $\subset$ CTL* a LTL $\subset$ CTL*, LTL s CTL mají společný průnik, avšak neplatí, že by jedna byla podmnožinou druhé. Podrobněji se o temporálních logikách zmiňuje například [Zbo04].

4.2.3 BDI CTL logika

Logika BDI CTL vychází z logiky CTL* . K jejímu jazyku přidává tři modální operátory: Bel, Des a Int. Sémantika vychází opět z Kripkeho struktury, ale tentokrát se jedná o sedmici $M = (W, (S_w, w \in W), (R_w, w \in W), L, B, D, I)$, kde W je množina světů, S_w stavy světa w, R_w relace mezi stavy světa w, L ohodnocení atomických formulí v každém stavu každého světa a B, D, I jsou relace mezi světy z množiny W: $B \subseteq W \times W$, $D \subseteq W \times W$, $I \subseteq W \times W$. Oproti předchozím logikám, kde byla pouze jedna množina stavů S a jedna relace R, zde je jich celá řada pro každý ze světů $w \in W$.

Sémantika operátorů Bel, Des a Int je pak pro formuli f, světy $v, w \in W$ a jejich stavy s_w, s_v definována takto:

$$M, s_w \models Belf \iff \forall v, (v, w) \in B : M, s_v \models f$$
$$M, s_w \models Desf \iff \forall v, (v, w) \in D : M, s_v \models f$$
$$M, s_w \models Intf \iff \forall v, (v, w) \in I : M, s_v \models f$$

Tedy aby platila formule $Belf$, musí platit formule f pro stavy s_v všech světů v, které jsou se světem w v relaci B (obdobně pro zbylé dva operátory). Axiomatika této logiky je poměrně rozsáhlá, je možné ji nalézt například v [Zbo04].

Na to, jaký by měl být vztah mezi jednotlivými modalitami, neexistuje jednotný názor. Jednotlivé přístupy se nazývají realismy. Tři nejčastěji uváděné realismy jsou:

- *silný realismus*, kde platí $B \subseteq D \subseteq I$,
- *realismus*, kde $I \subseteq D \subseteq B$, a konečně
- *slabý realismus*, kde $B \cap D \neq \emptyset$, $B \cap I \neq \emptyset$, $D \cap I \neq \emptyset$ a dále platí $Intf \rightarrow \neg Des \neg f$, $Intf \rightarrow \neg Bel \neg f$ a $Desf \rightarrow \neg Bel \neg f$.

Z těchto realismů vyplývá, jaké chování agent vykazuje. V silném realismu pečlivě vychází jen z toho, co je v daném okamžiku platné (přání jsou podmnožinou představ), v realismu je tomu naopak. Slabý realismus definuje vztah těchto modalit volněji, pouze předpokládá jejich neprázdný průnik a udává pravidla, které by měla pro vztah mezi modalitami platit.

4.3 Realizace architektury BDI

Implementovat agenta přesně podle výše uvedené teorie je poměrně obtížné a ve skutečnosti neexistuje žádná funkční aplikace, která by to takto dělala. Jediným náznakem je stále abstraktní model interpretu BDI agenta, kterou prezentovali autoři BDI CTL logiky Rao a Georgeff v [RG92]. Ostatní implementace BDI systémů souvisí s touto teorií poměrně volně, principy jejich přístupu si postupně stručně představíme.

4.3.1 Abstraktní interpret BDI agenta

Abstraktní model interpretu BDI agenta měl přispět k posunu od teorie k praktickým aplikacím. Různí autoři uvádějí mírně odlišné varianty původního algoritmu, ta která je zde prezentována, byla převzata z publikace [Kub04]. Interpret po inicializaci pracuje v nekonečné smyčce, kdy provádí reakce na události, které v jeho okolí nastaly. Celý algoritmus vypadá takto:

```
Bel = Bel0 // inicializace představ
Int = Int0 // inicializace záměrů
While (true) do
  Events = percept() // přijmi nové události z prostředí
  Bel = updateBeliefs(Bel, Events) // aktualizuj představy
  Goal = options(Bel, Int) // vyber nesplněné dosažitelné cíle
  Int = filter(Bel, Goal, Int) // na základě těchto cílů přijmi záměr
  Plan = plan(Bel, Int) // sestav plán pro dosažení tohoto záměru
  execute(Plan) // vykonej plán
  Int = dropSuccessful() // odstraň splněné záměry
  Int = dropImpossible() // odstraň nesplnitelné záměry
```

4.3.2 IRMA

IRMA (Intelligent Resource-bounded Machine Architecture) [BIP91] představuje jednu z prvních realizací architektury BDI. Na jejím návrhu se přímo podílel prof. Bratman, architektura tak vychází poměrně přesně z jeho teoretických úvah. Konceptuální schéma jádra systému znázorňuje obrázek 4.1. Na obrázku jsou datové struktury vyznačeny elipsami, procesy obdélníky. Je tedy patrné, že hlavní koncepty teorie BDI, tedy především jednotlivé modality, byly převedeny na datové struktury. Další klíčové komponenty jsou:

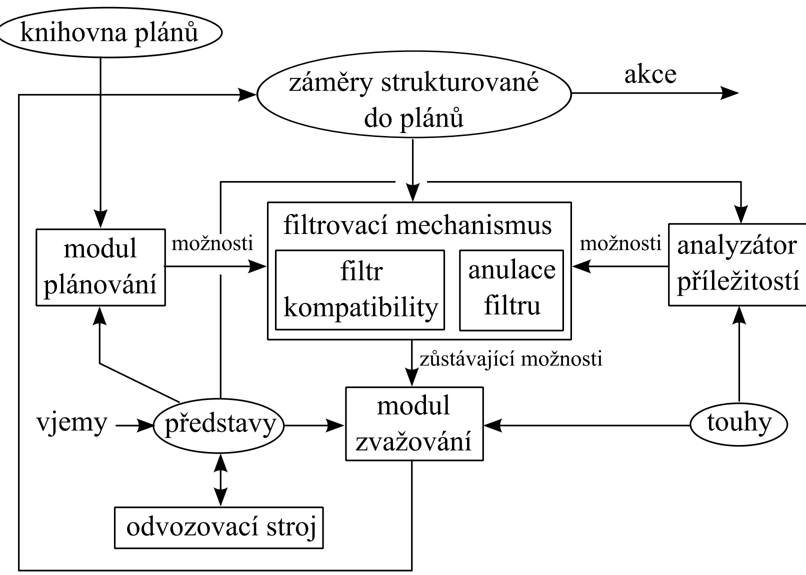
- *Modul plánování* (means ends reasoner), který nabízí možnosti, jak dosáhnout agentových cílů. Konkrétněji, vybírá šablony plánů z knihovny plánů a doplňuje je o unifikace proměnných a podcílů pro dosažení aktuálních cílů. Takto konkretizované plány předává jako možnosti do *filtrovacího mechanismu*

- *Analyzátor příležitostí* (oportunity analyser), jež přijímá podněty ze senzorů a navrhuje možnosti, které nastaly v důsledku změny prostředí.

- *Filtrovací mechanismus*, který přijímá možnosti z obou výše zmíněných modulů a vybírá z nich ty, které jsou kompatibilní s agentovými záměry (záměry zde odpovídají plánům, jež agent právě provádí). Toto filtrování musí být výpočetně efektivní, protože jeho hlavním cílem je zrychlit a zefektivnit práci modulu zvažování. Součástí tohoto

mechanismu mohou být pravidla, která propustí i možnosti, které by jinak přes filtry neprošly (anulace filtru) – tento přístup umožňuje řešit různé aplikačně specifické výjimky.

- *Modul zvažování* (deliberation unit), který z možností, jež prošly filtry, vybírá nové záměry, popřípadě upravuje stávající plány na základě nových skutečností.
- *Odvozovací stroj* (reasoner), který má na starosti odvozování nových poznatků z agentových představ.



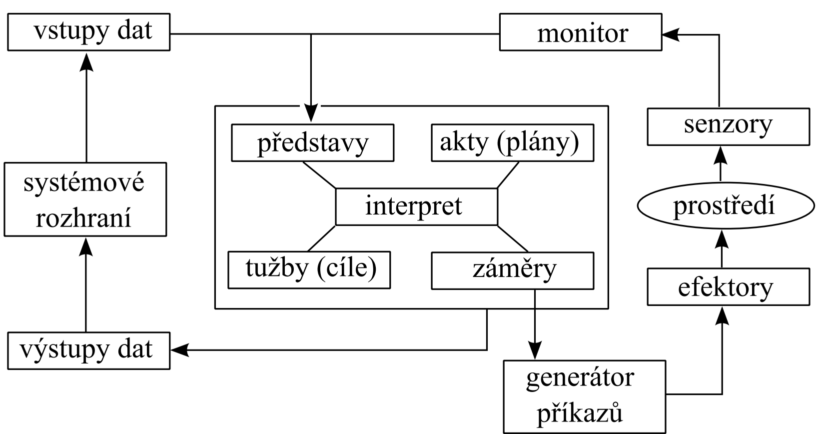
Obrázek 4.1: Schéma architektury IRMA

Z popisu jednotlivých komponent je snad zřejmé, jak systém na globální úrovni funguje. Problémem však je vyvážení spolupráce jednotlivých komponent, tedy zejména jak často provádět zvažování a přeplánování. Existují dva krajní přístupy – *opatrný* (cautious) agent bude přehodnocovat své záměry již při relativně malých změnách prostředí, zatímco *odhodlaný* (bold) agent setrvá ve vykonávání zvoleného plánu, dokud to jen bude možné. V prvním případě agent často zbytečně zvažuje alternativy a váhavost spojená s častým měněním plánu může mít negativní vliv na efektivitu jeho chování, stejně jako ve druhém případě trvaní na zvoleném plánu i v situacích, kdy je plán odsouzen k neúspěchu. Optimální strategie leží zřejmě mezi těmito dvěma krajními případy, avšak vhodná míra „opatrnosti“ se liší podle prostředí a aplikace.

4.3.3 PRS

Architektura PRS (Procedural Reasoning System) [GL87], [GI89] je v dnešní době stále zřejmě nejznámější implementací BDI principů a stala se základem pro celou řadu dalších systémů. Název této architektury se snaží vyzdvihnout fakt, že znalosti postupů pro dosažení cílů jsou uloženy v procedurální formě (tedy v plánech). Tyto struktury však autoři nenazývají plány – v původních článcích je nazývají *oblasti znalostí* (knowledge areas – KA), v pozdější literatuře pak *akty* (acts).

Schéma architektury PRS ukazuje obrázek 4.2. Jádro systému se skládá z pěti komponent – čtyři hlavní datové struktury odpovídají jednotlivým modálním operátorům plus akty (plány); interpret řídí chování celého systému. Z datových struktur jsou zřejmě nejzajímavější akty. Skládají se z hlavičky, kde jsou definovány podmínky, které musí platit pro jejich spuštění; a těla, které má formu grafu. V tomto grafu se kromě jednotlivých akcí mohou vyskytovat také podcíle, alternativy, cykly, rekurze a několik koncových stavů znamenajících úspěch či neúspěch.

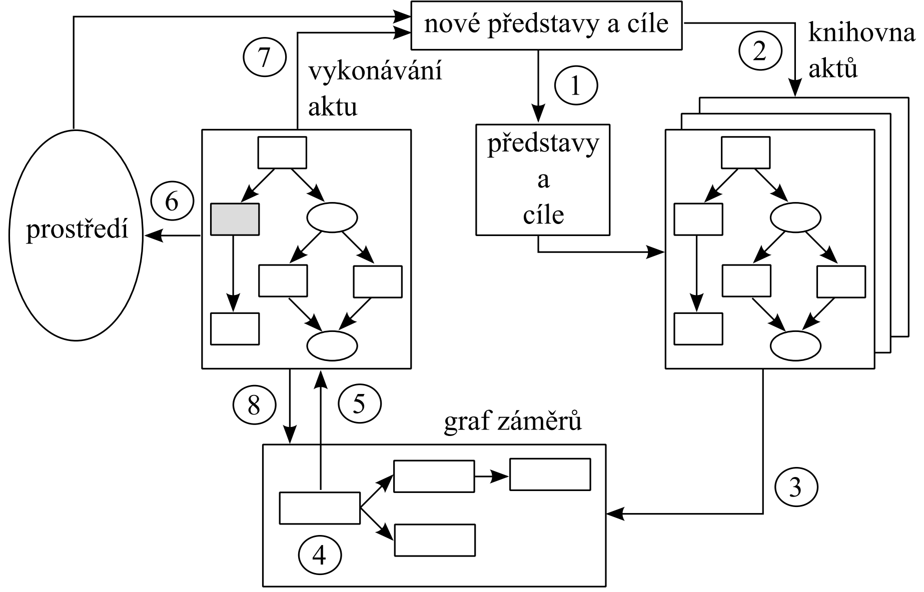


Obrázek 4.2: Schéma architektury PRS

Představy jsou vyjádřeny pomocí formulí predikátové logiky prvního řádu. PRS rozlišuje tři typy *tužeb* (cílů) – k jejich značení se používá operátorů ! ? a #. Výraz (! p), kde p je popis nějakého stavu, značí cíl „dosáhni, aby p platilo", (? p) znamená „otestuj, zda p platí" a konečně (# p) „udržuj stav systému takový, aby p platilo". *Záměry* jsou reprezentovány jako struktury, které obsahují původní akt a dále všechny akty, které vznikly při vykonávání tohoto aktu. Autoři uvádějí, že tyto záměry odpovídají procesům v běžném systému. Všechny záměry, které se agent rozhodl realizovat jsou uchovávány v tzv. struktuře záměrů, což je množina, nad kterou je definováno částečně uspořádání, které určuje priority a pořadí vykonávání jednotlivých záměrů.

Chování systému je určeno smyčkou interpretu, kterou znázorňuje obrázek 4.3. Jak je vidět, interpret pracuje v nekonečném cyklu, přičemž každá smyčka se skládá z osmi kroků:

1. Do systému dorazí nové představy a cíle.
2. Tato změna může spustit některé akty z knihovny aktů.
3. Některé z těchto aktů mohou být vybrány a přidány do grafu záměrů.
4. Z grafu záměrů je vybrán akt k interpretaci.
5. Ten je aktivován.
6. Z tohoto aktu je vykonán jeden krok, který může ovlivnit prostředí.
7. Tato změna může vyvolat nové představy a cíle.
8. Také může dojít k úpravě záměrů.



Obrázek 4.3: Schéma chování interpretu PRS

Krok 3 bývá nazýván *meta-plánování*. Jeho cílem je pro danou událost vybrat z množiny aplikovatelných aktů ty, které budou opravdu vykonávány. Meta-plánování může být v nejjednodušším případě implementováno jako vybrání prvního či náhodného aplikovatelného aktu, případně mohou být jednotlivým aktům přiřazeny fixní priority. Nejkomplexnější variantu představuje spuštění *meta-plánu*, který optimální akt vybere dynamicky podle okolností.

Původní PRS byl implementován v jazyku LISP, později byla na jeho základě vytvořena celá řada dalších verzí, o nichž se stručně zmíníme v následující části.

4.3.4 Systémy vycházející z PRS

UM PRS, JAM

UM-PRS (University of Michigan PRS) [LHDK94] představuje přímočarou reimplementaci originálního PRS v jazyce C++. Od originálního PRS se liší především jazykem použitým pro reprezentaci oblastí znalostí a podporou pro integraci s existujícím neagentním softwarem (UM-PRS pak představuje v takovém systému řídící nadstavbu). Použit byl zejména v oblasti řízení robotů, konkrétně reálných autonomních terénních vozidel.

Na UM-PRS navázal systém JAM [Hub99], který koncepčně z UM-PRS vychází a přidává celou řadu dalších vlastností – zejména zabudovanou možnost komunikace s ostatními agenty a migrace agentů po síti. JAM byl implementován v jazyce Java, což umožňuje snadnou integraci s existujícím softwarem v tomto jazyce pomocí jeho reflektivních vlastností. Oba systémy jsou pro nekomerční využití dostupné zdarma včetně zdrojových kódů pod licencí ve stylu GNU.

dMARS, JACK

Systém dMARS (Distributed Multi-Agent Reasoning System) představuje přímého následovníka architektury PRS. Vytvořen byl v Australském institutu pro umělou inteligenci (AAII) pod vedením prof. Georgeffa, jednoho z hlavních tvůrců originálního PRS (bývá nazýván systémem „druhé generace PRS"). Byl implementován v jazyku C++ a úspěšně nasazen v řadě průmyslových aplikací, z nichž asi nejznámější je systém pro správu letiště s názvem OASIS [GR96], který je často uváděn jako jedna z nejpokročilejších aplikací agentních systémů vůbec. Poskytován byl na komerční bázi. Se zánikem AAII skončil i projekt dMARS, stal se však základem pro systém JACK.

Hlavní vylepšení systému dMARS oproti PRS (kromě efektivnější implementace a celé řady podpůrných nástrojů) se týkalo plánů. Plány (u tohoto systému se již neoznačovaly jako akty) byly oproti PRS poněkud komplexnější – skládaly se ze spouštěcích podmínek; kontextu plánu (seznam představ, kterým agent musí věřit, aby mělo provedení plánu smysl); těla, které mělo formu stromu; udržujících podmínek, které musely platit po celou dobu provádění plánu a seznamu akcí, které se mají provést v případě úspěchu, respektive neúspěchu plánu.

Na systém dMARS navázal projekt JACK Agent Language [BRHL99]. Opět se jedná o komerční software, vyvíjí jej firma Agent Oriented Software, do které přešla řada vývojářů z AAII. Na rozdíl od dMARS představuje rozšíření jazyka Java o principy agentově orientovaného programování: poskytuje bázové třídy, rozhraní a metody, stejně tak jako rozšíření jazyka pro definice a sémantická rozšíření pro zpracování běhu agentního systému. Tvorba agenta tak spočívá ve zdědění bázových tříd a doplnění aplikačně specifických agentových schopností, báze představ, plánů a dalších podpůrných komponent. Systém poskytuje grafické rozhraní pro některé části návrhu, jako je tvorba plánů. Hlavní koncepty – tj. chování interpretu a základní datové struktury jsou obdobné jako tomu bylo u dMARS (a potažmo PRS).

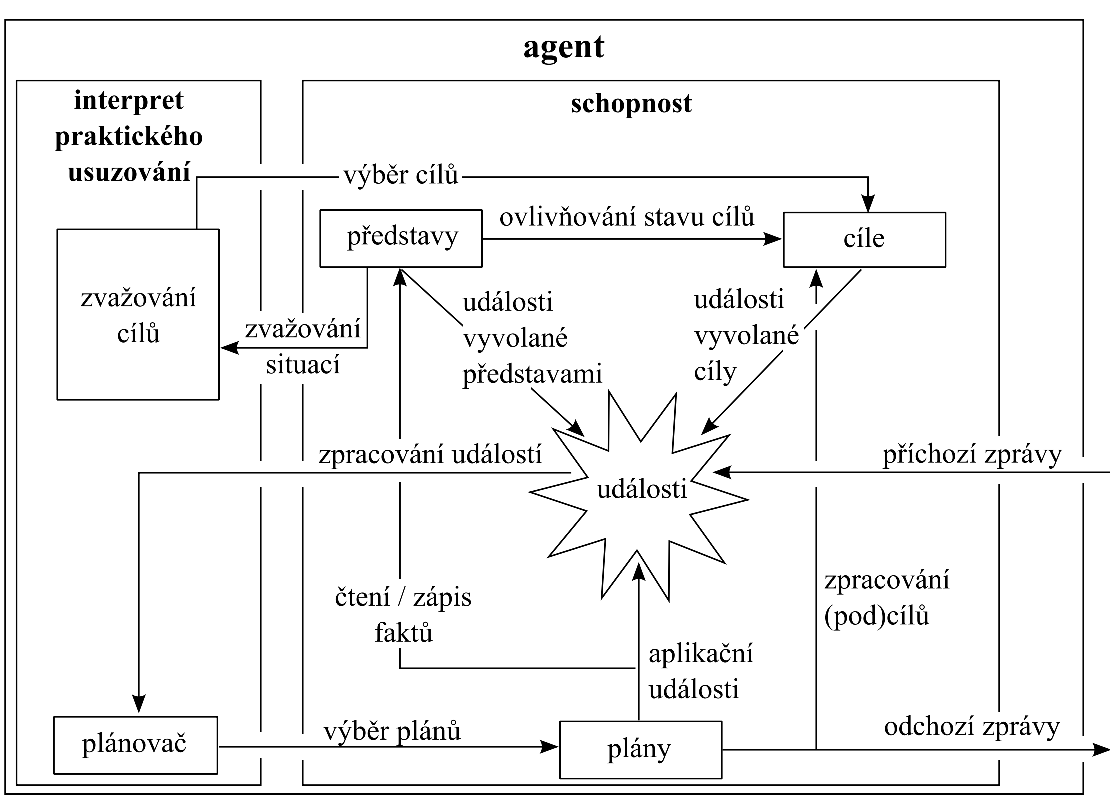
Sémantika systému JACK není formálně definována (vzhledem k tomu, že se jedná o nadmnožinu jazyka Java, bylo by nutno nejprve formalizovat Javu [Win05]). JACK se zaměřuje především na otázky softwarového inženýrství a nasazení celého systému v praktických aplikacích. Systém v této oblasti zavádí oproti dMARS řadu rozšíření, například zavedení pojmu *schopnost* (capability), což je prostředek pro strukturování agenta (obdoba modulů u běžného programování) – každá schopnost v sobě obsahuje cíle, představy a záměry, agent se pak skládá z několika takových schopností, přičemž jednotlivé schopnosti mohou být použity u více agentů.

4.3.5 JADEX

Oproti předchozím projektům, které by se daly nazvat „klasikou" v oblasti implementací BDI, JADEX [PBL03] je poměrně nový a stále velmi aktivní projekt (vývoj začal v roce 2002). Celý systém je postaven na platformě JADE [BR01] – middlewaru pro tvorbu agentních aplikací, který poskytuje prostředky a služby pro správu agentů a jejich komunikaci. JADEX k těmto prostředkům přidává BDI nadstavbu.

Převážná část informací o agentovi je v JADEXu uložena v tzv. *definičním souboru agenta* (Agent Definition File), který má formát XML. Zde se definují agentovy představy, cíle, schopnosti a mnoho dalších vlastností. Jediná část, která se vytváří odděleně jsou těla plánů – ta jsou uložena ve speciálních souborech a jejich obsah se vytváří pomocí běžného kódu programovacího jazyka Java.

Konceptuální model agenta je zobrazen na obrázku 4.4 (ten byl vytvořen podle ilustrací v online manuálu[1]). Jak je vidět, architektura nevychází přímo z PRS, například neobsahuje explicitně odlišené touhy a záměry. Obě tyto kategorie jsou spojeny do *cílů* a jejich rozlišování je částečně nahrazeno zavedením *životního cyklu cílů*. JADEX totiž vychází z myšlenky, že cíle jsou konkrétní, momentální přání agenta. Pro jakýkoliv cíl agent provádí více či méně přímé akce k jeho dosažení až do té doby, kdy považuje cíl za splněný, nesplnitelný anebo nadále nežádoucí. Na rozdíl od většiny BDI systémů JADEX nepožaduje, aby cíle byly konzistentní. Životní cyklus cílů obsahuje tři hlavní stavy – alternativa (option), aktivní (active) a odložený (suspended). Cíle mohou být čtyř druhů – vykonej (perform), dosáhni (achieve), otestuj (query) a udržuj (maintain). Stejně jako systém JACK obsahuje mechanismus *schopností* pro strukturování agenta do modulů. Agent tak obsahuje jeden řídící mechanismus a několik schopností, mezi které řízení rozděluje příchozí zprávy.



Obrázek 4.4: Konceptuální model agenta v systému JADEX

Reprezentace představ je velmi jednoduchá, v současné době nepodporuje žádnou formu odvozování (například založené na logice). Plány se stejně jako u systémů založených na PRS skládají z hlavičky a těla. Hlavička je specifikována v definičním souboru agenta a obsahuje podmínky pro spuštění plánu. Tělo je pak tvořeno běžným kódem jazyka Java, který má za úkol dosažení nějakého cíle nebo reakci na nějakou událost. To má jistou nevýhodu v tom, že jakékoliv řídící elementy musí být do plánu explicitně zakomponovány programátorem – například reakce na dosažení či selhání podcíle apod. Zaměření systému JADEX je obdobné jako u JACK – snahou autorů bylo přiblížit BDI architekturu co nejvíce

[1]http://vsis-www.informatik.uni-hamburg.de/projects/jadex/jadex-0.96x/userguide/index.single.html

vývojářům se zkušenostmi s klasickým softwarovým inženýrstvím.

4.3.6 Další implementace BDI

Tento výčet implementací architektury BDI samozřejmě není zdaleka úplný. Existuje celá řada dalších reimplementací PRS, stejně jako systémů, které souvisí s BDI spíše volně. Cílem bylo ukázat principy implementace a klasické projekty v této oblasti, které byly zdrojem inspirace a srovnání s navrženým frameworkem. Další zajímavé systémy zahrnují projekty 3APL a 2APL [Das08], Nuin [DW03], COSY [BS92] a GRATE* [Jen93].

4.4 Formalizace systémů založených na architektuře BDI

Jak bylo ukázáno v předchozích částech, jeden směr vývoje realizací BDI architektury se snaží přiblížit přístupy BDI ke klasickému softwarovému inženýrství a od formální teorie se oproti svým předchůdcům spíše vzdaluje. Tento přístup reprezentují v současné době především projekty JACK a JADEX. Existují však naopak také projekty, které se snaží mezeru mezi teorií architektury BDI a jejími praktickými realizacemi zúžit. Zřejmě nejznámější prací v této oblasti je jazyk AgentSpeak(L) a formální specifikace několika systémů ve specifikačním jazyce Z. Těmto přístupům se budu nyní krátce věnovat.

4.4.1 AgentSpeak(L)

Vznik projektu AgentSpeak(L) [Rao96] byl motivován velkými rozdíly mezi původní formální teorií BDI, ve které jsou představy, přání a záměry vyjádřeny jako modální operátory a implementujícími systémy vycházejícími z PRS, kde jsou tyto realizovány jako datové struktury.

Systém AgentSpeak(L) má dvě hlavní komponenty. První z nich představuje interpretovaný jazyk, který je založen na podmnožině predikátové logiky obohacené o události a akce. Tento jazyk může být chápán jako alternativní forma formalizace BDI agenta, přičemž sémantika jazyka zachycuje právě hlavní rysy systémů PRS a dMARS. Operační sémantika tohoto jazyka je formalizována pomocí přechodového systému. Představy jsou stejně jako u PRS a dMARS vyjádřeny pomocí formulí predikátové logiky. Cíle mohou být podobně jako u dMARS dvou druhů – dosažení a testování platnosti formule. Na rozdíl od PRS však plány mají lineární strukturu a navíc oproti dMARS je značně zjednodušena jejich hlavička (žádné udržovací podmínky či množiny akcí k provedení po úspěchu, resp. neúspěchu plánu apod.).

Druhou komponentu pak tvoří interpret tohoto jazyka (resp. přesný popis algoritmu, jak má být tento jazyk interpretován). AgentSpeak(L) tak představuje formální popis implementace BDI agenta. Tento popis je však na poměrně vysoké úrovni abstrakce – AgentSpeak(L) zachycuje pouze základní rysy, proto byla později různými autory publikována řada rozšíření a vylepšení, např. [BdOJB+02, MVB03].

Interpret rozšířené verze jazyka AgentSpeak(L) byl později realizován v jazyce Java projektem Jason [BH05]. Hlavní rozšíření se týká možnosti komunikace agentů pomocí jazyka založeném na řečových aktech. Jason tedy umožňuje vyjádřit rozhodovací proces agenta ve formálně definovaném jazyce. Napojení na okolí je provedeno pomocí volání primitivních akcí, které může programátor vytvořit v jazyce Java.

4.4.2 Formální specifikace dMars a 3APL

Dalším přístupem k formalizaci architektur BDI agentů je použití specifikačního jazyka Z [Bow96]. Tímto způsobem bylo popsáno vysokoúrovňové chování systému 3APL [dHL00] a dMARS [dLG^{+}04]. Tato formalizace umožňuje jasně definovat chování jednotlivých systémů, případně jednotlivé systémy porovnávat, nicméně není přímo proveditelná.

Kapitola 5

Modelování BDI agentů objektově orientovanými Petriho sítěmi

Cílem předchozích kapitol bylo ukázat současný stav výzkumu v oblasti multiagentních systémů, se zaměřením na vnitřní architektury agentů, zejména architekturu BDI, ze kterého vyplývají hlavní teoretická východiska pro výzkum prezentovaný v této práci. Tato kapitola popisuje hlavní přínos disertační práce – framework *PNagent*, který umožňuje modelování agentů s architekturou BDI objektově orientovanými Petriho sítěmi. PNagent byl navržen a prototypově implementován v nástroji PNtalk. Koncepce, návrh a aplikace frameworku byly publikovány na uznávaných vědeckých konferencích [KMZJ07], [MKJZ08b], k recenzi byla odeslána taktéž publikace do tématicky zaměřeného vědeckého časopisu [MKJZ08a]. Předtím než budou podrobně rozebrány jednotlivé aspekty návrhu frameworku, pokusím se shrnout východiska, která z předchozích kapitol vyplývají a tedy podrobněji specifikovat cíle práce v jejich kontextu.

5.1 Požadavky, cíle, východiska

Jak bylo ukázáno ve třetí kapitole, zřejmě nejperspektivnějším přístupem ke tvorbě vnitřní architektury agentů se zdají být systémy založené na praktickém usuzování, případně hybridní architektury, které tento systém doplňují o rychlé reakce na bezprostřední podněty. Čtvrtá kapitola se zabývala nejúspěšnějším přístupem k tvorbě systémů založených na praktickém usuzování – architekturou BDI. Zde jsme viděli, že architektura BDI není jasně daná specifikace, jak vytvořit vnitřní strukturu agenta, spíše se jedná o třídu systémů, které více či méně vycházejí ze základních konceptů formulovaných filosofem Bratmanem. Některé systémy se v současné době snaží o přiblížení se ke klasickému softwarovému inženýrství, jiné se naopak pokouší dát implementacím formální rámec alternativní k původní příliš expresivní logice.

Jedním z poměrně často diskutovaných témat v klasickém softwarovém inženýrství je v dnešní době vývoj systémů založený na modelech (Model Driven Engineering) – například ve formě xUML (Executable UML) [MB02]. Objektově orientované Petriho sítě a nástroj PNtalk se snaží být kompatibilní alternativou k těmto přístupům, přičemž si zachovává formální základ. Navržený framework tak můžeme chápat jako reprezentanta obou dvou aktuálních trendů v oblasti výzkumu BDI architektury – jak prakticky použitelný systém z pohledu softwarového inženýrství, tak formálně definovaný systém s jasnou specifikací umožňující aplikaci formálních metod verifikace přímo na vysokoúrovňový popis. Při návrhu

frameworku jsem se řídil zejména následujícími požadavky:

- *otevřenost* – aplikační programátor by měl tvořit konkrétní modely multiagentních systémů stejnými prostředky, jakými byl vytvořen vlastní framework. Hranice mezi frameworkem a aplikacemi by tak neměla být ostrá jako například ve dříve zmíněném systému Jason, návrhář by měl mít možnost snadno přizpůsobovat chování frameworku podle potřeby konkrétního modelu. Tento přístup zároveň umožní experimenty se samotnou architekturou BDI.

- *vizuální reprezentace* – systém by měl poskytovat aplikačnímu programátorovi přehledné grafické rozhraní pro tvorbu celého systému, na rozdíl od většiny existujících systémů pro tvorbu BDI agentů, kde je tvorba jednotlivých komponent poměrné neintuitivní (např. tvorba aktů pomocí jazyka Lisp v PRS), což vede k nutnosti doplňovat systém o speciální grafické nástroje, které činí systém složitější a omezují jeho otevřenost.

- *podpora pro vývoj systémů založený na modelech* – aplikační programátor by měl mít možnost navrhnout a implementovat jednotlivé části struktury agenta na různých úrovních abstrakce. Výsledný model by měl být přímo proveditelný alespoň jako prototyp aplikace, v optimálním případě přímo jako cílová aplikace.

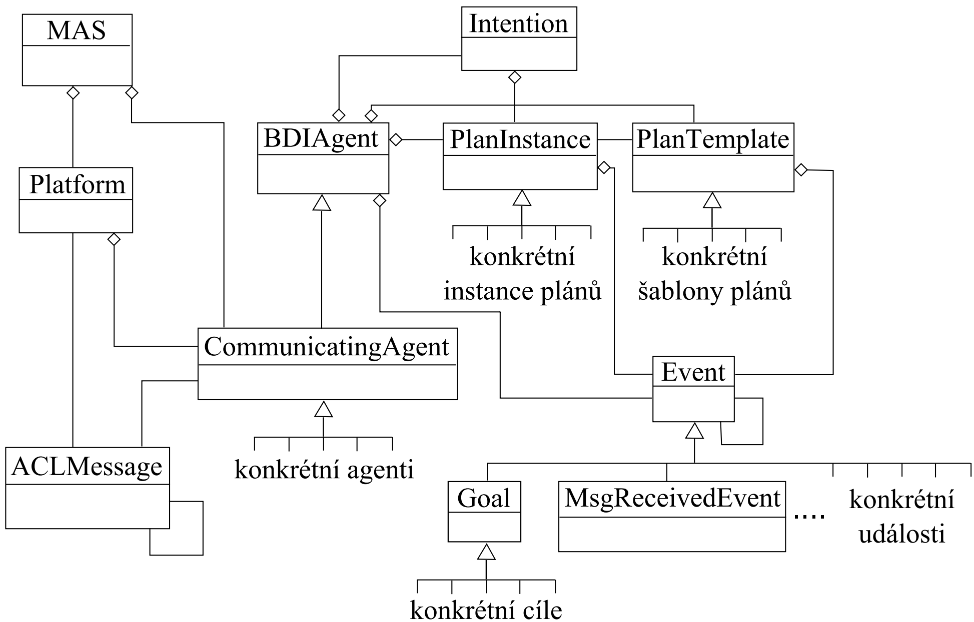
- *navázání na uznávané standardy FIPA* – všechny části frameworku, které spadají do oblastí, které byly standardizovány organizací FIPA by měly být s těmito specifikacemi kompatibilní a alespoň částečně je implementovat.

- *zachování formálního základu* – přesto, že PNtalk poskytuje prostředky pro propojení s dalšími programovacími paradigmaty, při tvorbě frameworku by měl být zachován formální rámec u všech částí, které by potenciálně mohlo být potřeba formálně verifikovat.

5.2 Přehled frameworku PNagent

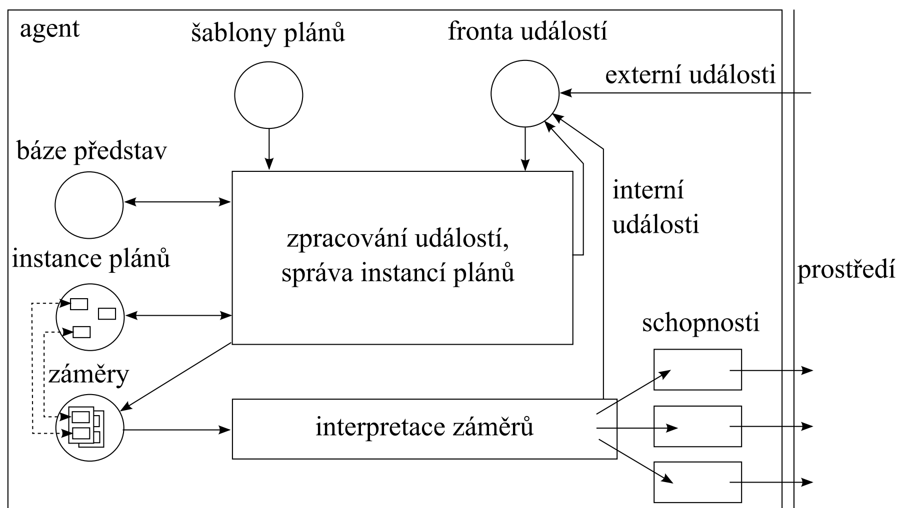
Framework PNagent poskytuje kolekci tříd, které usnadňují tvorbu agentů s architekturou BDI, schopných komunikovat pomocí vysokoúrovňového jazyka pro meziagentní komunikaci (v současném prototypu je podporována podmnožina jazyka FIPA ACL). Zjednodušený diagram tříd celého systému na obrázku 5.1 ukazuje nejdůležitější třídy frameworku a jejich vztahy.

Bázovou třídou pro všechny konkrétní agenty je ve frameworku třída *BDIAgent*, která poskytuje základní abstrakci pro interpret, spolu s podpůrnými strukturami a metodami. Každý agent obsahuje množinu plánů, které jsou rozděleny na šablony (objekty podtříd bázové třídy *PlanTemplate*) a instance (objekty podtříd bázové třídy *PlanInstance*). Vztah šablon a instancí plánů je takový, že šablony poskytují prostředky pro vytváření instancí. Instance plánů jsou strukturovány v záměrech (objekty třídy *Intention*), které mají podobu zásobníku. Veškeré změny v systému se provádějí pomocí událostí (objekty podtříd bázové třídy *Event*). Některé základní události jsou přímo součástí frameworku (například přidání či odebrání představy, úspěch či selhání instance plánu, příjem zprávy apod.). Další, aplikačně specifické události musí doplnit aplikační programátor. Speciálním druhem událostí jsou cíle (objekty podtříd bázové třídy *Goal*), které se od běžných událostí liší tím, že jsou perzistentní.



Obrázek 5.1: Zjednodušený diagram tříd frameworku PNagent zachycující nejdůležitější třídy a jejich vztahy

Obrázek 5.2 ukazuje základní funkčnost interpretu agenta – procesy jsou zobrazeny jako obdélníky, datové struktury jako kruhy. Interpret provádí dva hlavní procesy. Prvním je správa událostí, která zahrnuje modifikaci báze představ, tvorbu a správu životního cyklu instancí plánů a tvorbu nových záměrů. Druhým procesem je pak interpretace záměrů, která má za následek ovlivňování prostředí a tedy tvorbu dalších událostí. Oba procesy budou podrobně popsány v části zabývající se interpretem.



Obrázek 5.2: Schéma základní funkčnosti rozhodovacího jádra agenta ve frameworku PNagent

Třída *CommunicatingAgent* představuje nadstavbu nad rozhodovacím jádrem agenta, která přidává zejména prostředky pro meziagentní komunikaci pomocí vysokoúrovňového jazyka, tedy zejména schopnost přijímat a odesílat zprávy (realizované objekty třídy *ACLMessage*). Vlastní přenos zpráv je uskutečňován pomocí platforem (objekty třídy *Platform*), které představují prostředí, ve kterém se agenti nacházejí. Objekt třídy MAS (Multi Agent System) pak představuje celý model, typicky se v něm provádí inicializace, případně sběr statistik, vyhodnocování experimentů, apod.

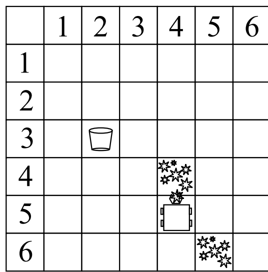
Aplikační programátor musí při tvorbě konkrétního modelu ve frameworku PNagent přidat alespoň tyto komponenty: bázi představ a metody pro agentovy schopnosti do třídy agenta, externí události pro modelované prostředí a aplikačně specifické plány (šablony a instance). V následujících sekcích budou jednotlivé komponenty diskutovány podrobněji.

5.3 Reprezentace báze představ

První aspekt frameworku, na který se zaměříme, je reprezentace agentových představ. Tato reprezentace má totiž, jak uvidíme dále, vliv prakticky na všechny další komponenty systému. Pro názornost bude zvolený přístup demonstrován na jednom z klasických testovacích prostředí, zvaném *Cleaner world*.

5.3.1 Ukázkový systém – Cleaner world

Prostředí Cleaner world je reprezentováno diskrétní 2D mřížkou. Na každém z polí se může nacházet odpad nebo popelnice. Odpad se objevuje v průběhu simulace náhodně. V prostředí se dále nachází autonomní robot (agent), jehož cílem je přesunout všechen odpad do popelnic. Robot je schopen vykonat tři atomické externí akce: zvednout odpad, upustit odpad a přesunout se na vedlejší políčko. Přenášet může vždy odpad pouze z jednoho pole. Situace může vypadat například jako na obrázku 5.3. Zde se na pozici (2,3) nachází popelnice a na pozicích (4,4) a (5,6) odpadky. Robot se nachází na pozici (4,5) a momentálně přenáší odpad.



Obrázek 5.3: Ukázka prostředí Cleaner world

5.3.2 Tradiční způsob reprezentace báze představ

Klasický způsob reprezentace představ u agentů s architekturou BDI je obdobný použití *faktů* v logickém programovacím jazyce *Prolog*. Představy jsou tedy vyjádřeny jako literály predikátové logiky prvního řádu neobsahující proměnné. Za použití syntaxe Prologu by reprezentace příkladu zmíněného v předchozí podkapitole mohla vypadat například takto:

```
wastePosition(5,6).
wastePosition(4,4).
binPosition(2,3).
agentPosition(4,5).
carryingWaste.
```

Máme-li takovouto reprezentaci, můžeme tvořit odvozovací pravidla, která se *dotazují* nad těmito fakty. Taková pravidla se pak používají při řadě rozhodovacích procesů uvnitř agenta, například při posuzování relevance určitého plánu při výskytu události. Jako příklad takového pravidla v Cleaner worldu uveďme test, zda je agent na stejném poli jako popelnice a přenáší odpad. V Prologu bychom takové pravidlo mohli vyjádřit například takto:

```
canDropWaste :- binPosition(X,Y), agentPosition(X,Y), carryingWaste.
```

Alternativním přístupem k použití logiky pro bázi představ může být například ukládání představ v relační databázi. Představy jsou pak vyjádřeny jako relace, k dotazování se používá jazyk SQL. Tento způsob je zřejmě méně flexibilní, zato však může dosahovat větší výkonnosti díky propracovaným technikám optimalizace.

Při návrhu báze znalostí je tedy potřeba zvolit jednak vlastní reprezentaci, která by měla poskytovat zejména snadný způsob ukládání a odebírání faktů, a dále mechanismus umožňující efektivní dotazování nad těmito fakty. Otázky automatického odvozování nových znalostí z báze představ či revize představ (např. řešení nesouladu mezi obsahem báze znalostí a informací ze senzorů) jsou obvykle ponechány na případných rozšířeních aplikačních programátorů.

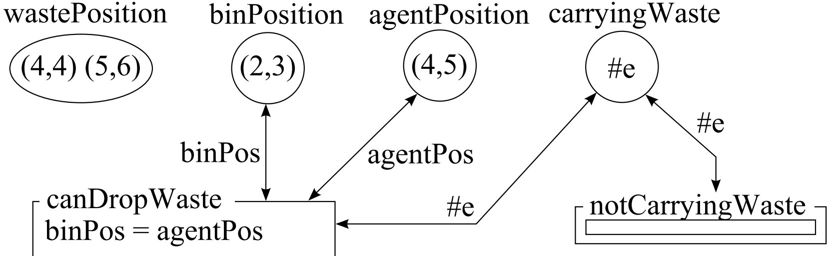
5.3.3 Reprezentace báze představ pomocí prostředků OOPN

Reprezentaci představ pomocí logiky je samozřejmě možné použít i v prostředí OOPN. Znamenalo by to však buď implementovat odvozovací aparát pomocí prostředků Petriho sítě či inskripčního jazyka, anebo celý systém práce s reprezentací z OOPN vyjmout a zpracovávat reprezentaci představ externě, například pomocí objektu zapouzdřujícího některý existující interpret Prologu, dostupný v prostředí systému Smalltalk. Stejně tak v případě použití relační databáze.

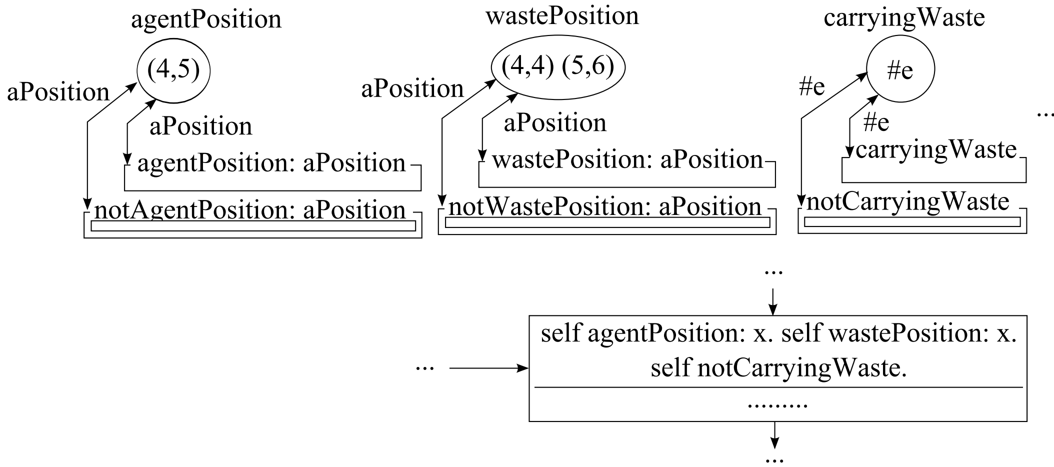
OOPN však umožňují jiný přístup k reprezentaci znalostí, založený přímo na dostupných strukturách OOPN. Výrazy odpovídající faktům v Prologu se vyjádří pomocí míst a značek. Odvozování se pak provádí pomocí synchronních portů. Pro každý „typ“ predikátu se vytvoří jedno místo, jak to ukazuje obrázek 5.4 pro příklad Cleaner worldu. Značky pak reprezentují hodnoty jednotlivých formulí (připomeňme, že v OOPN struktura tvaru (x,y) reprezentuje seznam a symbol #e značku bez specifické hodnoty – „černou značku“).

Odvozovací pravidla se vyjádří synchronními porty. Jsou možné dva přístupy: buď se vytvoří jeden synchronní port vyjadřující celé pravidlo, jak to ukazuje obrázek 5.4, anebo se pro každé místo vytvoří synchronní port se stejným jménem, který umožňuje pouze navázat libovolnou značku z tohoto místa, a tyto porty se pak podle potřeby spojí s dalšími omezujícími podmínkami až ve stráži přechodu, jako konjunkce výrazů (obrázek 5.5).

Důležitou součástí této reprezentace je test nepřítomnosti značky s určitou hodnotou v místě (odpovídající použití predikátu *not* v jazyce Prolog). Jak jsme viděli ve druhé kapitole, tento test je obvykle ve vysokoúrovňových Petriho sítích problém. V jazyce PNtalk byl proto zaveden koncept negativních predikátů, které svou sémantikou predikátu *not* odpovídají.



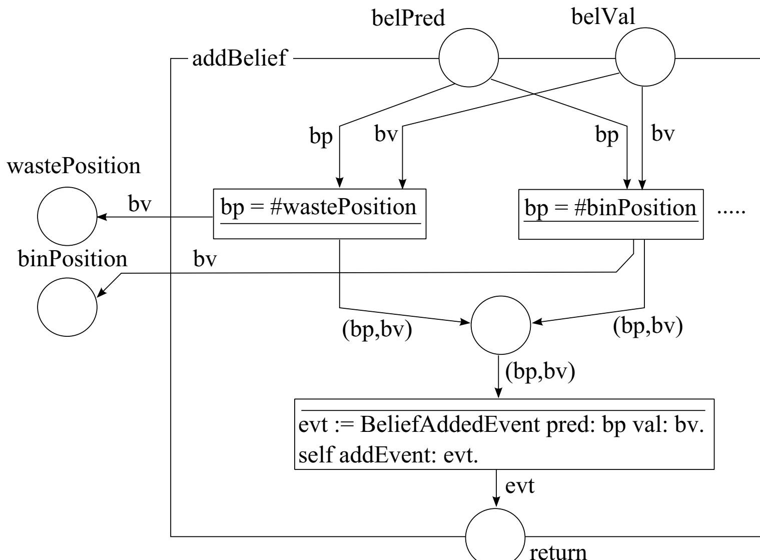
Obrázek 5.4: Reprezentace představ o prostředí Cleaner world pomocí OOPN



Obrázek 5.5: Odvozování nad reprezentací prostředí Cleaner world pomocí skládání synchronních portů a negativních predikátů ve stráži přechodu

Hlavní výhodou tohoto způsobu reprezentace v kontextu frameworku PNagent, která nakonec rozhodla o volbě tohoto přístupu, je zachování kompaktnosti celého systému – představy jsou vyjádřeny pomocí stejných výrazových prostředků jako zbytek systému. Báze představ ve frameworku PNagent tedy vypadá tak, že se pro každý typ představy, který se má v bázi znalostí nacházet, vytvoří místo a dvojice predikát / negativní predikát, které potom mohou být testovány v rámci přechodů ve všech ostatních komponentách (plánech, událostech, apod.). Příklad ukazuje obrázek 5.5. Manipulace se značkami (tedy přidávání a odebírání jednotlivých konkrétních predikátů) se provádí pomocí metod *addBelief* a *removeBelief*, do kterých se pro každý typ predikátu přidá speciální přechod, jak to ukazuje pro metodu *addBelief* obrázek 5.6. Tyto metody pak zajišťují korektní vyvolání interních událostí při změně báze znalostí.

Báze představ je v současné prototypové implementaci umístěna přímo v objektové sítě třídy BDIAgent. V budoucnosti by u rozsáhlejších projektů mohla být oddělena do samostatného objektu, který by zpřístupňoval predikáty a negativní predikáty pro testování báze představ a metody *addBelief* a *deleteBelief* pro její změny (vkládání či odstraňování značek). Pro účely prototypu je však současný přístup flexibilnější a plně dostačující.



Obrázek 5.6: Metoda addBelief, která slouží k přidávání představ do báze. Pro každý typ představy ji aplikační programátor musí rozšířit podle schématu ukázaného pro představy wastePosition a binPosition

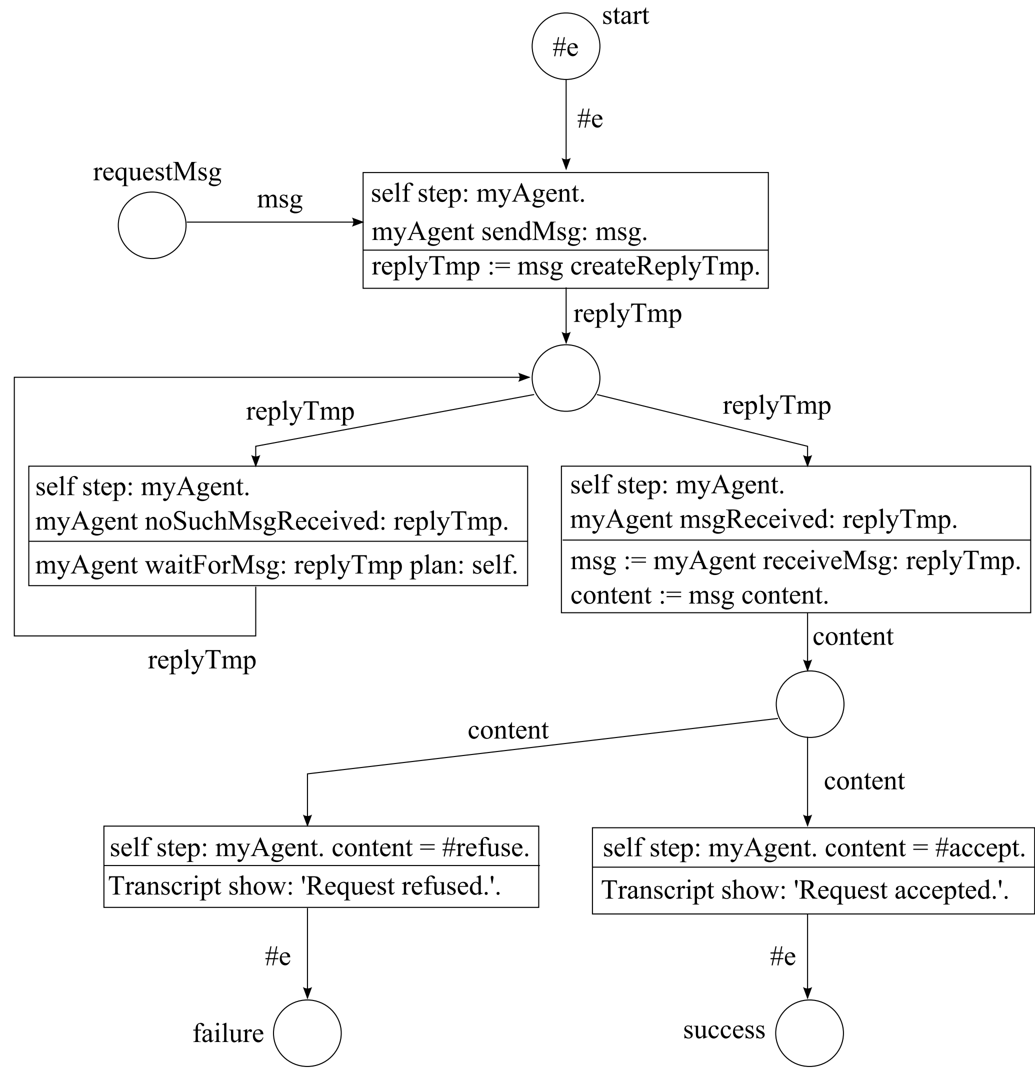
5.4 Plány

Ve čtvrté kapitole jsme viděli, že plány hrají v realizacích architektury BDI klíčovou roli, protože reprezentují agentovy procedurální znalosti o tom, jak dosahovat cílů. Typicky jsou vytvářeny aplikačním programátorem v době návrhu aplikace, agent tedy neprovádí žádné plánování v klasickém smyslu. Plány však nicméně mohou být neúplně specifikované, případně obsahovat podcíle.

Framework PNagent podporuje koncept plánů dvěma abstraktními třídami, které poskytují prostředky pro vytváření instancí plánů, správu jejich životního cyklu, synchronizaci s dalšími plány, komunikaci s objektem třídy *BDIAgent*, atd. Rozdělení konceptu plánů do dvou bázových tříd má převážně technické důvody – objektový model OOPN neposkytuje koncept *třídních metod*. Část funkčnosti plánu, která má na starosti tvorbu konkrétních instancí je soustředěna ve třídě *PlanTemplate*, zbytek funkčnosti se pak nachází ve třídě *PlanInstance*. Objekt třídy PlanTemplate tak vlastně hraje roli třídních metod třídy PlanInstance. Aplikační programátor musí vždy vytvořit dvojici konkrétních tříd zděděním těchto dvou abstraktních tříd a doplnit do nich konkrétní funkčnost. V případě třídy PlanTemplate jsou to podmínky, při kterých je plán aplikovatelný a případná aplikačně specifická inicializace instance plánu. U třídy PlanInstance je to pak především vlastní obsah plánu (tedy podsíť definující sled akcí, podcílů apod.) a případně podmínky pro změnu stavu plánu. Při inicializaci agenta jsou umístěny šablony všech plánů (tedy instance podtříd třídy PlanTemplate) do speciálního místa v objektové síti třídy BDIAgent, které hraje roli *knihovny plánů*. Zde jsou pak šablony plánů testovány při interpretaci událostí v systému, kdy se zjišťuje, zda je plán *relevantní* pro danou událost. Pokud se plán vyhodnotí jako relevantní a je zvolen z množiny všech relevantních plánů, vytvoří se jeho instance. Tento proces bude podrobně rozebrán později společně s konceptem událostí.

5.4.1 Přidávání specifické funkčnosti instance plánu

Objektová síť třídy PlanInstance obsahuje mimo jiné tři důležitá místa – *start*, *success* a *failure*. Úkolem aplikačního programátora je doplnit mezi tato tři místa přechody a místa pro konkrétní funkčnost daného plánu. Každý přechod by měl odpovídat provedení agentovy atomické akce nebo vyvolání podcíle. Místa slouží k uložení stavu instance plánu mezi jednotlivými kroky. Jeden krok plánu by měl odpovídat provedení jednoho přechodu, nicméně framework to nijak nevynucuje, je tedy možné provést více přechodů (včetně paralelních větví, variant či cyklů) jako jeden krok. Framework zde ponechává možnost používat všechny prostředky dostupné v prostředí PNtalku. Příklad instance plánu, která má na starosti jednoduché odeslání zprávy a čekání na odpověď ukazuje obrázek 5.7. Obrázek zachycuje pouze aplikačně specifickou část objektové sítě instance plánu.



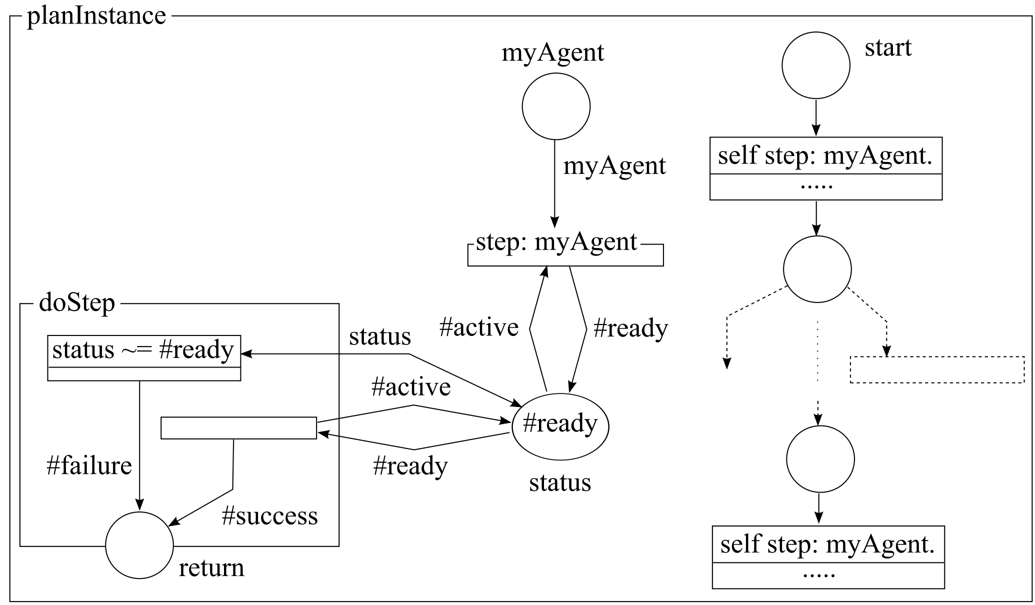
Obrázek 5.7: Ukázka aplikačně specifické části objektové sítě instance plánu

Typická struktura přechodu v podsíti realizující funkčnost instance plánu vypadá tak, že ve stráži se volá jako první podmínka proveditelnosti synchronní port *self step: myAgent*.

Ten má dva úkoly: postranním efektem definuje konec kroku plánu (změnou stavu instance plánu, tímto mechanismem se podrobně zabývá další podkapitola) a zároveň poskytuje referenci na objekt agenta, která se pak používá u většiny dalších volání metod a synchronních portů v rámci přechodu – jak pro testování báze znalostí, tak pro volání agentových atomických akcí. Ve stráži přechodu se typicky volají synchronní porty představené v podkapitole o reprezentaci báze znalostí, které omezují proveditelnost přechodu, případně provádějí konkretizaci parametrů plánu. Volání agentových atomických akcí, případně vyvolávání podcílů pak probíhá v těle přechodu. Podcíl se vyvolává vytvořením objektu reprezentujícího příslušný cíl a zavolání metody *self postSubgoal: aGoal*, která zajistí všechny technické detaily (navázání podcíle na záměr, uspání instance plánu, probuzení při dosažení tohoto podcíle, atd.).

5.4.2 Životní cyklus instance plánu

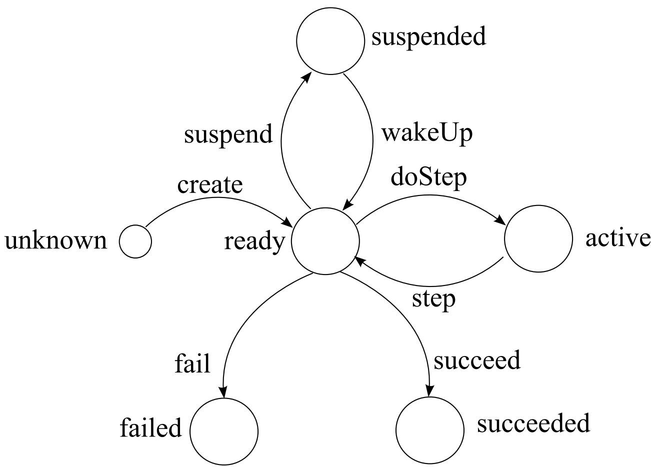
Každá instance plánu po vytvoření projde inicializací a přejde do stavu *#ready* (plán je připraven k vykonávání). Aby mohl být proveden jeden krok plánu, musí být stav nastaven na *#active*. To provádí interpret voláním metody instance plánu *doStep*. Konec kroku je určen tím, že se ve stráži přechodu v podsíti realizující vlastní funkčnost plánu zavolá synchronní port *step: myAgent*, který kromě získání reference na agenta nastaví stav plánu zpět na *#ready*. Volání tohoto synchronního portu tedy odpovídá provedení jednoho kroku plánu (všechny přechody, které toto volání neobsahují se provedou v rámci jednoho kroku). Mechanismus provedení kroku instance plánu pomocí volání *doStep* a *step: myAgent* ukazuje obrázek 5.8.



Obrázek 5.8: Mechanismus provedení kroku instance plánu

Kromě stavů *#ready* a *#active* může plán čekat na nějakou událost, typicky vykonání podplánu, splnění podcíle nebo příjem zprávy. V tom případě se nachází ve stavu *#suspended*. K uspání a vzbuzení plánu slouží několik metod, které nastavují *šablony událostí*, které musí nastat, aby ke změně stavu došlo. Vlastní mechanismus práce s událostmi je obdobný

jako při vytváření plánu, podrobně bude diskutován v podkapitole zabývající se událostmi. Poslední dva stavy, ve kterých se může plán nacházet, se nazývají *#succeeded* a *#failed* a odpovídají koncovým stavům při úspěchu, resp. selhání plánu. Při přechodu do těchto stavů se automaticky volají metody succeed, resp. fail, které zajišťují vytvoření odpovídajících událostí. Aplikační programátor je může samozřejmě předefinovat pro dosažení specifické funkčnosti. Celý životní cyklus instance plánu ukazuje obrázek 5.9.



Obrázek 5.9: Životní cyklus instance plánu

5.4.3 Priority plánů

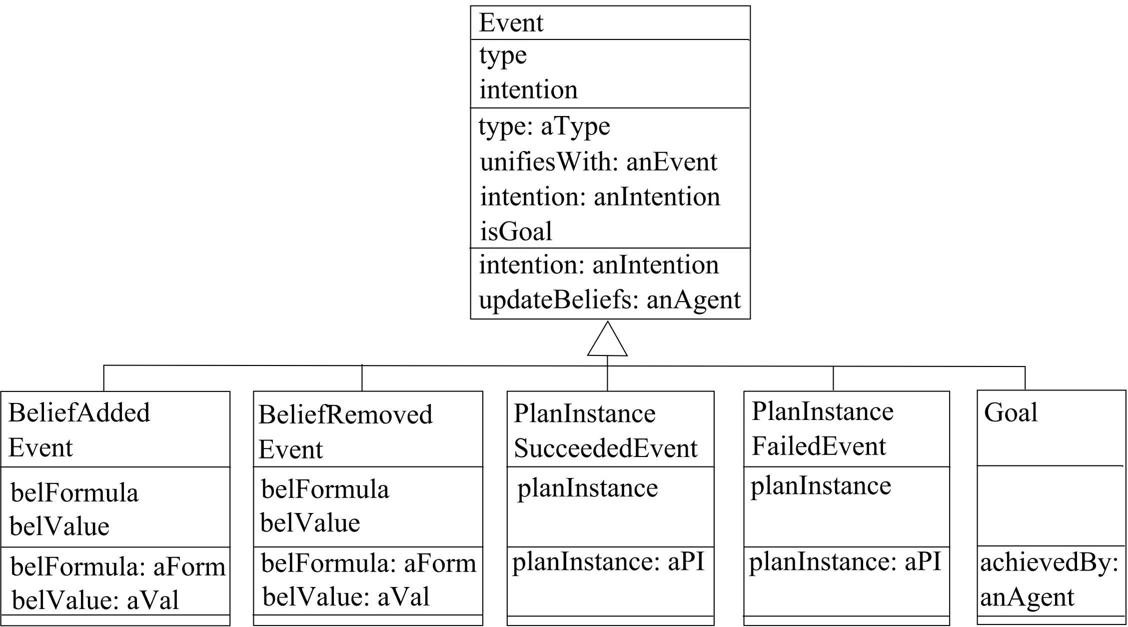
Každá šablona i instance plánu má definovánu svoji prioritu. U šablon může priorita rozhodovat o tom, který plán bude zvolen při dané události; u instancí pak o pořadí vykonávání jednotlivých instancí plánů v rámci provádění záměrů. Instance prioritu implicitně získává od šablony plánu, v průběhu vykonávání instance plánu je možné ji programově měnit pomocí dvojice metoda / synchronní port *priority: aPriority* (synchronní port slouží k získání hodnoty, metoda pro nastavení). Priority jsou jen jeden ze způsobů, který se v rámci interpretace agenta k řešení možnosti aplikace více plánů používá, podrobněji se tímto tématem zabývá podkapitola Interpret.

5.5 Události

Události reprezentují všechny změny v systému, které je agent schopen rozpoznat, případně zachytit svými senzory. Podle místa vzniku se události rozlišují na externí (změny v agentově prostředí) a interní (změny uvnitř agenta). Ve frameworku PNagent jsou externí události generovány objekty třídy *Platform*, která modeluje agentovo prostředí – takové události jsou vždy aplikačně specifické, a tedy musejí být vytvořeny aplikačním programátorem. Interní události jsou generovány uvnitř agenta a používají se především pro informování plánů o změnách báze představ, stavu instancí plánů, nových cílech, apod.

Základem pro koncept událostí je abstraktní třída *Event*, která definuje potřebná rozhraní, zejména poskytuje synchronní porty pro testování typu události a unifikovatelnosti. Postupem při unifikaci se podrobně zabývá následující podkapitola. Dále také obsahuje referenci na záměr, která u interních událostí definuje, který záměr událost vyvolal. U externích

událostí je tato vazba vytvořena v případě, že dojde k vytvoření nového záměru. Odvozené třídy mohou obsahovat volitelný počet míst, které pak nesou informace specifické pro tuto událost. Princip reprezentace těchto informací je obdobný jako u reprezentace představ, diskutovaný dříve. Nejdůležitější metody, vztahy a interní události definované přímo v rámci frameworku ukazuje diagram na obrázku 5.10. Jedná se o drobně modifikovaný diagram tříd (rozšířený o možnost vyjádřit synchronní porty). Obdélník označující třídu je tedy rozdělen na čtyři části, v horní části je uveden název třídy, v další části jsou atributy (jež v OOPN odpovídají místům), třetí pole obsahuje synchronní porty a konečně spodní část metody.



Obrázek 5.10: Modifikovaný diagram tříd pro události ve frameworku PNagent

5.5.1 Unifikace událostí, spouštění plánů

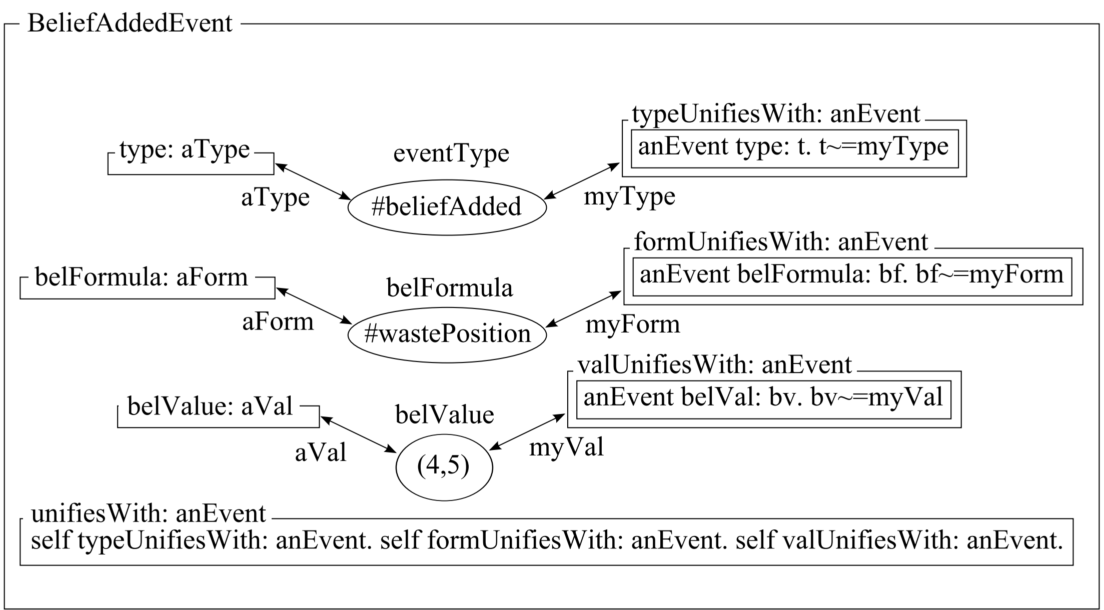
Dvě události se ve frameworku PNagent považují implicitně za unifikovatelné, pokud mají stejný typ a obsah jejich aplikačně specifických míst je unifikovatelný. Typ je určen místem *type*, zavedeným v bázové třídě Event (typicky odpovídá názvu konkrétní třídy události, kterou je však v současné verzi PNtalku nesnadné získat). Aplikačně specifickými místy se rozumí místa, která obsahují vlastní popis události – například v případě události reprezentující objevení nového odpadu jeho pozici. Tato místa se považují za unifikovatelná, pokud obsahují alespoň jednu stejnou hodnotu, nebo je alespoň jedno z těchto míst prázdné (tato podmínka se obvykle zjednoduší, protože takováto místa mohou typicky obsahovat pouze jednu značku).

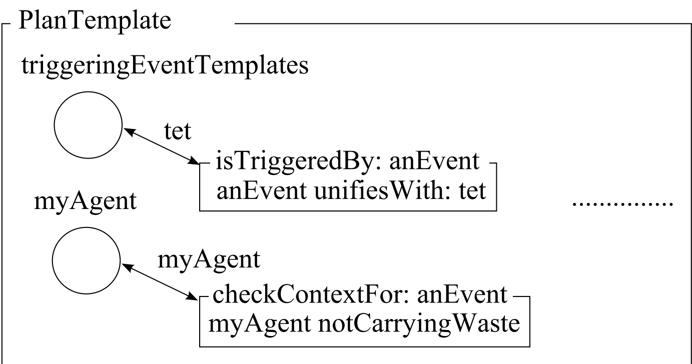
Koncept unifikace událostí je používán při všech změnách stavu instancí plánů. Princip bude ukázán na vytváření instancí; pro uspávání, probouzení, předčasné dosažení úspěchu a selhávání instancí se liší jen názvy a umístění metod a synchronních portů.

Každá šablona plánu obsahuje místo *triggeringEventTemplates*, do kterého aplikační programátor vloží *šablony událostí*, jež mohou vyvolat vytvoření instance tohoto plánu. Šablona události je vlastně běžná událost (tedy objekt příslušné podtřídy třídy Event),

která nemusí mít žádné značky v aplikačně specifických místech (což odpovídá volným proměnným) anebo jich naopak může mít několik (což vyjadřuje alternativy, pro které je plán relevantní). Například v případě plánu, který se má spustit při objevení nového odpadu, by šablona události neměla v místě reprezentujícím pozici odpadu žádnou značku – byla by unifikovatelná s jakoukoli konkrétní událostí objevení odpadu.

Testování, zda je plán pro danou událost relevantní, sestává ze dvou kroků. V prvním kroku se testuje, zda je událost unifikovatelná s některou ze šablon. Pokud ano, volá se synchronní port šablony plánu *checkContextFor: anEvent*, který může dotazovat bázi znalostí a vyjádřit tak další omezení pro relevanci plánu. Synchronní porty zapojené do unifikace událostí při spouštění plánu ukazuje obrázek 5.11. Zde se jedná o událost přidání představy do báze znalostí, která je implementovaná samotným frameworkem. U aplikačně specifických událostí musí synchronní port *unifiesWith: anEvent* implementovat aplikační programátor, přitom má možnost koncept unifikace konkrétních událostí upravit podle potřeby.





Obrázek 5.11: Ukázka principu unifikace událostí při vytváření instance plánu (implementace unifikace zde předpokládá nejvýše jednu značku v aplikačně specifických místech události)

5.5.2 Cíle

Cíle jsou ve frameworku považovány za speciální události, které se odlišují od ostatních události v tom, že jsou perzistentní. Jak bude ukázáno později v podkapitole o interpretaci, standardní událost je zpracována jedním cyklem interpretu, při kterém může, ale také nemusí vyvolat vytvoření instance plánu (případně změnit stav některých existujících instancí), a poté ze systému odchází. Cíle naproti tomu v systému zůstávají, dokud nejsou splněny nebo odloženy. Splnění cíle je testováno synchronním portem *satisfied*. Framework nerozlišuje mezi typy cílů (například cíle dosažení, udržení, apod.), typ cíle je určen implementací portu satisfied, kde je ponechána volnost aplikačnímu programátorovi. Při rozhodování o použití cíle nebo běžné události by se měl aplikační programátor řídit tím, že události odpovídají reaktivnímu chování (agent reaguje na změny v prostředí), zatímco cíle proaktivnímu (agent se snaží něčeho dosáhnout ze své iniciativy). Cíle tedy bývají obvykle interní (vyvolané existujícími instancemi plánů).

5.6 Záměry

Záměry jsou ve frameworku PNagent realizovány jako zásobníky instancí plánů. Každá externí událost nebo cíl, pro kterou byl nalezen nějaký aplikovatelný plán, vede k vytvoření nového záměru zároveň s instancí tohoto plánu. Všechny další případné podplány tohoto plánu jsou pak vkládány do tohoto záměru. Pokud je instance plánu na vrcholu zásobníku proveditelná (je ve stavu *#ready*), záměr může být vykonán jako celek. Po dokončení instance plánu na vrcholu zásobníku je tato instance odstraněna a je vzbuzena instance bezprostředně pod ní (která ji vyvolala). Po dokončení všech plánů v zásobníku je celý záměr odstraněn.

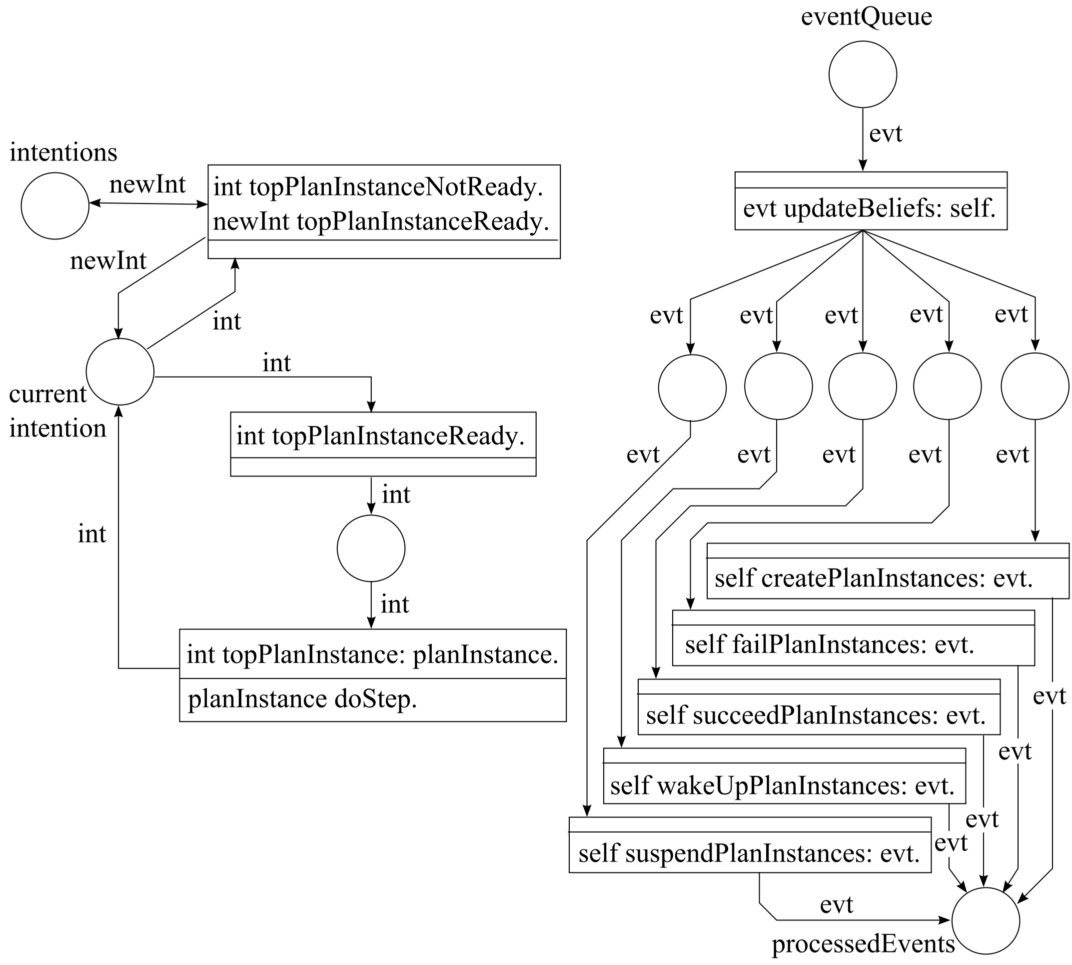
Události mohou paralelně k tomuto mechanismu měnit stav všech plánů v zásobníku, tedy je možné, že některý plán v zásobníku se stane neadekvátním (selže). Pak jsou všechny jeho podplány ze zásobníku odstraněny a hledá se alternativní řešení (vyvoláním události či znovuzasláním cíle).

Záměry mohou mít přiřazeny priority, které představují jeden z implementovaných mechanismů výběru záměrů pro provádění. Tuto prioritu přebírají automaticky od instance plánu, která vedla ke vzniku záměru. Priorita může být stejně jako u instancí plánů průběžně programově upravována.

5.7 Interpret

Interpret je implementován objektovou sítí třídy *BDIAgent*. Tato abstraktní třída slouží jako bázová třída pro všechny agenty vytvářené ve frameworku PNagent. Poskytuje všechny společné struktury a metody. Konkrétní třídy agentů k ní přidávají aplikačně specifické části, tedy zejména místa a synchronní porty pro bázi představ a metody, které reprezentují agentovy externí akce.

Část objektové sítě třídy BDIagent realizující chování interpretu je znázorněna na obrázku 5.12. Zřejmě největším rozdílem oproti klasickým implementacím BDI agentů je to, že tradiční cyklus interpretu – získej nové události, vytvoř odpovídající plány, uprav strukturu záměrů, vyber jeden záměr a proveď jeden krok jeho aktuálního plánu – je rozdělen do dvou nezávislých částí. Část zobrazená na obrázku vpravo je zodpovědná za zpracování událostí – události jsou postupně odebírány z fronty událostí, mají možnost upravit agentovu bázi znalostí a poté jsou paralelně předány metodám, jež mají za úkol vytváření

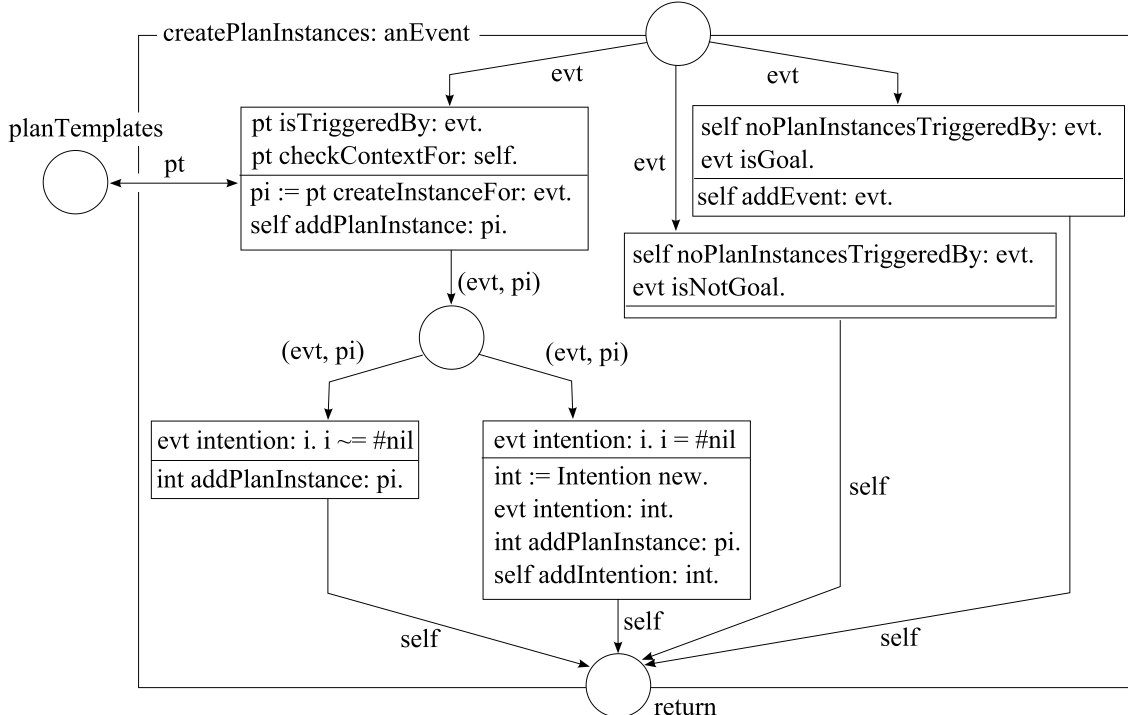


Obrázek 5.12: Část objektové sítě třídy BDIagent realizující chování interpretu

nových instancí plánů, resp. správu stavu existujících instancí (tedy uspávání, vzbouzení atd.). Druhá část, zachycená na obrázku vlevo, se stará o interpretaci struktury záměrů. Záměry jsou tedy interpretovány nezávisle na zpracování událostí. Je samozřejmě možné modelovat cyklus jako jeden celek, nicméně rozdělení se zdá být více flexibilní – v případě, že je synchronizace potřeba, může být dodána explicitně. Volné prokládání obou částí cyklu může být modelováno jednoduše díky tomu, že OOPN jsou založeny na událostech místo procesů – v klasických programovacích jazycích, jako je například Java by takové prokládání muselo být modelováno použitím souběžně běžících vláken.

5.7.1 Zpracování událostí

Každá událost má v prvním kroku svého zpracování možnost ovlivnit bázi znalostí agenta. To se děje předefinováním metody třídy Event *updateBeliefs: anAgent*, ve které je možné volat metody třídy BDIAgent *addBelief* a *deleteBelief* popsané dříve. Tento krok v případě změny báze znalostí samozřejmě vede k vyvolání dalších (interních) událostí. Jádro zpracování událostí interpretem spočívá v paralelním zavolání pěti metod třídy BDIAgent, které provádějí testování, zda je tato událost unifikovatelná s některou šablonou události pro jed-



Obrázek 5.13: Metoda třídy BDIAgent sloužící k vytváření instancí plánů a záměrů pro danou událost

notlivé změny stavu instancí plánu. Tyto metody jsou si značně podobné, proto bude opět ukázána pouze metoda *createPlanInstances: anEvent*, která se stará o vytváření nových instancí plánu. Její síť ukazuje obrázek 5.13.

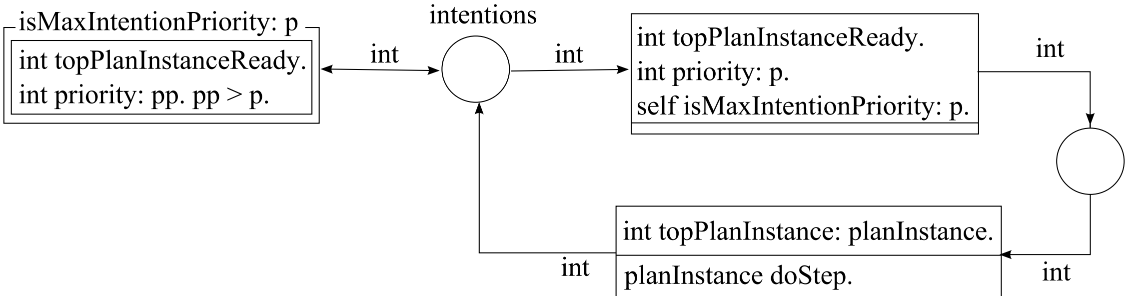
Framework poskytuje různé varianty implementace této metody, které se liší způsobem, jakým se přistupuje ke knihovně plánů (místo *planTemplates*). Nejjednodušší varianta implementace, která je ukázána na obrázku 5.13, vybere z množiny relevantních plánů jeden náhodně. Výběr plánu bývá nazýván meta-plánování a v současné době framework poskytuje možnost náhodného výběru, výběru plánu s nejvyšší prioritou a konečně spuštění speciálního aplikačním programátorem specifikovaného meta-plánu, který vybere nejvhodnější plán. Pokud událost nepatří dosud k žádnému záměru (tj. jedná se o externí událost), je vytvořen záměr nový. Není-li v daném okamžiku nalezen žádný relevantní plán, je metoda ukončena. V případě, že událost byla cílem, je poslána k opakovanému zpracování.

5.7.2 Interpretace záměrů

Interpret na obrázku 5.12 vykonává jeden záměr krok po kroku tak dlouho, dokud je tento záměr proveditelný a poté vybere náhodně jiný proveditelný záměr. Další varianta interpretace záměrů, kterou framework poskytuje, spočívá v tom, že v každém kroku se vybere z proveditelných záměrů ten s nejvyšší prioritou a krok jeho instance plánu na vrcholu zásobníku je pak vykonán. Tento přístup ukazuje obrázek 5.14.

Interpretace záměrů představuje vlastní funkčnost agenta – zde se v jednotlivých krocích realizují agentovy plány, tedy provádějí akce, které ovlivňují prostředí. Popis struktury a funkčnosti rozhodovacího jádra agenta tak byl dovršen. Následující podkapitola se bude

zabývat podpůrnými strukturami, reprezentujícími agentovo prostředí, zejména ve vztahu ke komunikaci s ostatními agenty. Ukázky konkrétních funkčních příkladů modelů jsou uvedeny v příloze B – Cleaner world a příloze C – Block world, rozsáhlejší aplikaci frameworku pak podkapitola 5.9.



Obrázek 5.14: Interpretace záměrů na základě priorit

5.8 Prostředí a meziagentní komunikace

Rozhodovací jádro BDI agenta popsané v předchozích podkapitolách se dá používat samostatně k řešení úloh, které mají dopředu známy všechny parametry (například úkol přeskládávání kostek prezentovaný v příloze C). Takový přístup je však velmi netypický – již v definici agenta je zdůrazněna návaznost na prostředí, se kterým agent neustále interaguje. Navíc takový přístup není efektivní – výhody reaktivního plánování se projeví jen v dynamicky se měnícím prostředí, ve kterém je potřeba upravovat agentovy cíle a plány během výpočtu na základě nových podmínek. Ve statických prostředích je architektura BDI ve většině případů méně efektivní než klasické plánovacími algoritmy.

Prostředí tedy představuje důležitou komponentu každého agentního systému, potažmo frameworku PNagent. Na prostředí se dá nahlížet ze dvou úhlů – jednak představuje zdroj podnětů a pole působnosti agenta (tento pohled je aplikačně specifický), zároveň však agentovi také poskytuje základní služby. Tyto služby, z nichž zřejmě nejdůležitější je zabezpečení prostředků pro vlastní běh agenta a komunikaci s dalšími agenty, jsou obdobné pro všechny agentní systémy a jak bylo ukázáno ve třetí kapitole, standardizované organizací FIPA. Obě funkce prostředí zastává ve frameworku PNagent třída *Platform*, tyto funkce budou nyní stručně diskutovány.

5.8.1 Platforma jako zdroj podnětů agenta

Již jsem zmínil, že funkce platformy jako zdroje podnětů a pole působnosti agenta je vždy aplikačně specifická. Tuto funkčnost tedy musí doplnit aplikační programátor tak, že zdědí bázovou třídu Platform a do její objektové sítě doplní podsíť, která představuje „fyzikální zákony“ daného prostředí. Vlastní předávání změn prostředí agentům se děje zasíláním zpráv *addEvent: anEvent* objektům třídy BDIAgent. Ve frameworku může objektu agenta tuto zprávu zasílat pouze objekt třídy Platform, protože ten jediný má na objekt agenta referenci. V případě, že je model napojen na reálné prostředí (z pohledu PNtalku, toto prostředí může být i dalším simulátorem, jak bude ukázáno dále), představuje třída Platform prostředníka, který přijímá vstupy z tohoto prostředí a převádí je na události pro

agenta. Stejně tak agentovy akce typicky spočívají v zasílání zpráv objektu platformy, která poté modifikuje svůj model prostředí, případně efekty agentových akcí předává do reálného prostředí. Platforma tak představuje vrstvu abstrakce prostředí pro vnitřní model agenta. Tento přístup bude demonstrován v následující kapitole, věnující se ukázkové aplikaci frameworku PNagent.

5.8.2 Služby poskytované platformou

Objekty třídy Platform poskytují v současné prototypové verzi agentům tři základní služby:

- *Správu životního cyklu* – zejména prostředky pro vytváření a zabíjení agentů, přidělování jednoznačných identifikátorů (AID), apod.
- *Službu bílých stránek* (vyhledávání agentů podle jejich AID).
- *Přenos zpráv* mezi agenty.

Jako komunikační paradigma je použito zasílání zpráv, jazyk odpovídá v prototypu podmnožině jazyka FIPA ACL. Zprávy jsou implementovány objekty třídy *ACLMessage*, která svou strukturou odpovídá zprávám jazyka FIPA ACL. Obsahuje tedy místa, synchronní porty pro získávání a metody pro nastavování jednotlivých položek: AID příjemců, AID odesílatele, performativu, obsahu zprávy atd.

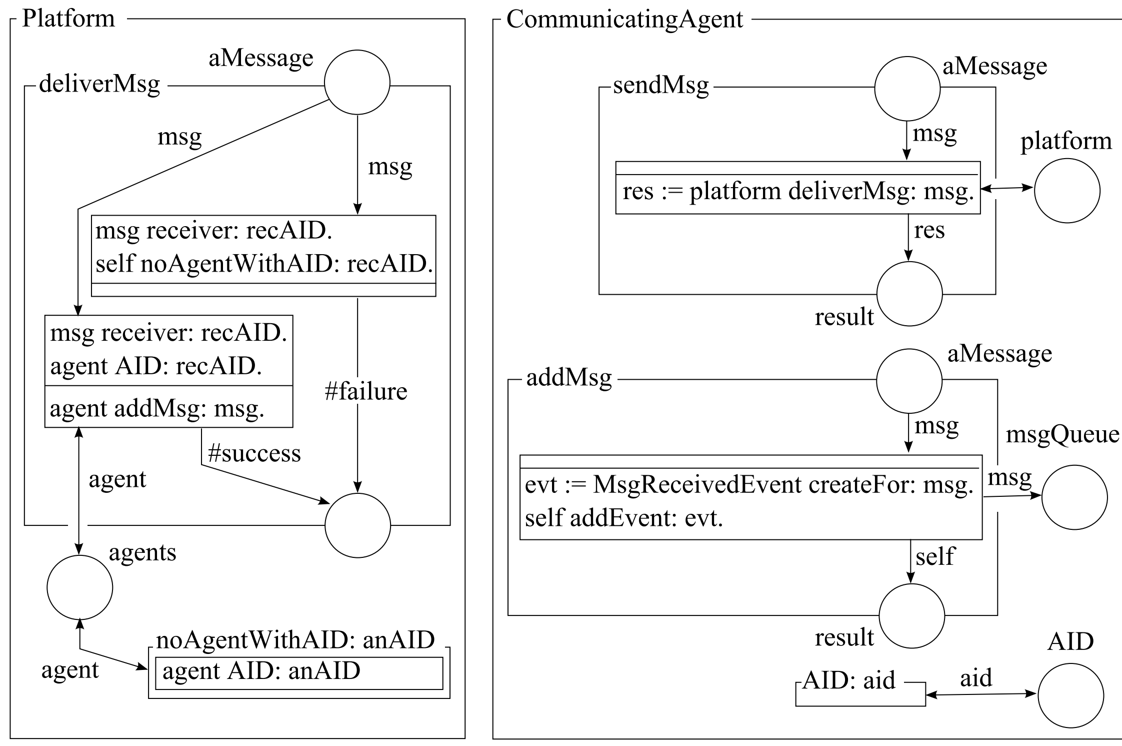
Na straně agenta je komunikace postavena jako vyšší vrstva nad základním rozhodovacím jádrem. Základní komponenty pro komunikaci jsou implementovány třídou *CommunicatingAgent*, která dědí rozhodovací jádro od třídy *BDIAgent* a přidává k němu zejména:

- Prostředky pro jednoznačnou identifikaci agenta (AID).
- Metody pro zasílání a příjem zpráv (ke kterým BDI jádro může přistupovat jako k agentovým schopnostem).
- Frontu příchozích zpráv a tvorbu událostí spojených s příchozími zprávami.

Veškerá komunikace probíhá přes agentovo prostředí, realizované objektem třídy Platform, který se stará o vlastní přenos zpráv mezi agenty, stejně jako přidělování jedinečných AID. Metody, místa a synchronní porty realizující zaslání zprávy v rámci jedné platformy na straně agenta i platformy ukazuje obrázek 5.15.

Zprávu agent zašle voláním své metody *sendMsg: aMessage*, která vyvolá vlastní přenos zprávy přes platformu voláním její metody *deliverMsg: aMessage*, jež předá zprávu příjemci voláním jeho metody *addMsg: aMessage*. Ta u příjemce vyvolá patřičnou událost a uloží zprávu do jeho fronty zpráv. Z této fronty ji může převzít některá z jeho existujících instancí plánů, případně může být na základě události příjmu zprávy vytvořena nová instance plánu. Vlastní převzetí zprávy se provádí voláním metody *receiveMsg: aMsgTemplate*. Parametr *aMsgTemplate* představuje šablonu zprávy, která má být přijata. Koncept šablon zpráv funguje obdobně jako koncept šablon událostí popsaný dříve.

Vzhledem k tomu, že oblast komunikace a správy agentů je v dnešní době poměrně dobře standardizovaná, byl při návrhu kladen důraz na prinicipielní soulad s těmito standardy. Tyto standardy však v prototypu nebyly implementovány do všech detailů. Ty by bylo potřeba doplnit při nasazení frameworku v praxi, pro výzkumné a prototypové účely současná implementace postačuje. Proto byla pozornost zaměřena spíše na aspekty rozhodovacího jádra.



Obrázek 5.15: Metody, místa a synchronní porty objektů třídy CommunicatingAgent a Platform realizující zaslání zprávy v rámci jedné platformy

5.9 Příklad aplikace frameworku PNagent

Kromě několika ukázkových příkladů zmíněných v předchozích částech této kapitoly, jejichž cílem bylo otestovat a demonstrovat základní vlastnosti frameworku PNagent, byla jeho funkčnost a použitelnost jako celku ověřena na úloze tvorby formace mobilních robotů. Součástí tohoto projektu bylo ověření použitelnosti multiparadigmatického modelování v prostředí systémů PNtalk a SmallDEVS. Samotná úloha formace mobilních robotů vychází z článku týmu prof. Zieglera [HZ04], který na ní ukazuje možnosti použití formalismu DEVS pro vývoj systémů založený na modelech.

5.9.1 Neformální zadání úlohy

V prostředí s překážkami se pohybuje skupina mobilních robotů. Každý robot se na počátku pohybuje náhodně po prostoru a vyhýbá se překážkám. Pokud jeho senzory zachytí překážku, tak si ověří, zda se jedná o skutečnou překážku, nebo o jiného robota. Pro toto rozlišení zašle všem ostatním robotům zprávu obsahující data ze svých sonarů. Pokud některý z ostatních robotů našel podobnost mezi těmito údaji a daty ze svých senzorů, všichni roboti zastaví a tento robot se pohne. Pokud se změnou sonarového vjemu potvrdí, že tito dva roboti se zaznamenali navzájem, vyšle robot, který protokol inicioval, příkaz „následuj mě“. Oba se pak natočí do stejného směru a tím vytvoří formaci. Od té chvíle se první robot nadále pohybuje náhodně po prostoru, přičemž příkazy svým aktuátorům přeposílá také druhému robotu. Ten je beze změny vykonává, ale na své pozici. Formace

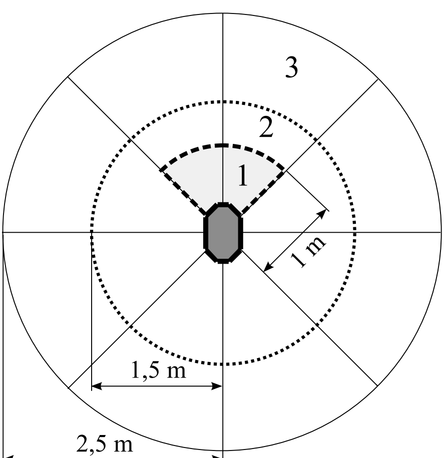
se tedy může rozpadnout v případě, že druhý robot nemůže provést příkaz, protože mu v tom brání překážka. V tom případě zašle vedoucímu zprávu „konec formace“, vedoucí robot přestane posílat příkazy a nadále se pohybují nezávisle na sobě, jak tomu bylo před vznikem formace.

Stejně jako tým prof. Zieglera jsme pro experimenty použili pouze dva roboty, rozšíření na libovolný počet robotů ve skupině by však vyžadovalo jen drobné úpravy komunikačního protokolu.

5.9.2 Mobilní roboti

Jako systém představující prostředí, ve kterém se roboti nacházejí, stejně jako abstrakci samotných robotů, byl použit simulátor Player/Stage[1]. Jedná se o jeden z nejpoužívanějších simulačních systémů v rámci komunity zabývající se robotikou. Skládá se ze dvou samostatných částí. *Player* je jazyk a platformě nezávislý síťový server pro ovládání robotů, který poskytuje interface pro kontrolu celé řady robotického a senzorového hardware pomocí protokolu TCP/IP. *Stage* simuluje populaci mobilních robotů, kteří se pohybují a přijímají podněty z 2D bitmapového prostředí [GVH03]. Výhodou použití kombinace Player/Stage je možnost obvykle zcela transparentní záměny simulovaných robotů v prostředí Stage za roboty reálné. Obě části systému jsou dostupné zdarma pod licencí GNU GPL.

Při experimentech byl použit jeden ze základních modelů robota implementovaných simulátorem Stage, ActiveMedia Pioneer P3-DX[2], vybavený osmi sonary o rozsahu 45° s dosahem 2,5 m, rozmístěnými radiálně kolem robota, osmi nárazníky, kompasem a rádiem, umožňujícím zasílání zpráv ostatním robotům (v realizovaném prototypu bylo zasílání zpráv implementováno lokálně v rámci frameworku PNagent). Schématický obrázek robota s vyznačenými zónami používanými pro detekci překážek ukazuje obrázek 5.16. Příkazy, které je schopen robot vykonat (odpovídají agentovým schopnostem) jsou čtyři: robot se může pohybovat vpřed, zastavit, otočit se o daný úhel a zaslat zprávu pomocí rádia.



Obrázek 5.16: Schéma použitého mobilního robota s vyznačením nárazníků a sonarů. Zóna 1 (nebezpečná zóna) je používána pro změnu směru při vyhýbání překážkám, zóna 2 (testovací zóna) pro testování, zda se v blízkosti nachází jiný robot a zóna 3 představuje dosah sonarů.

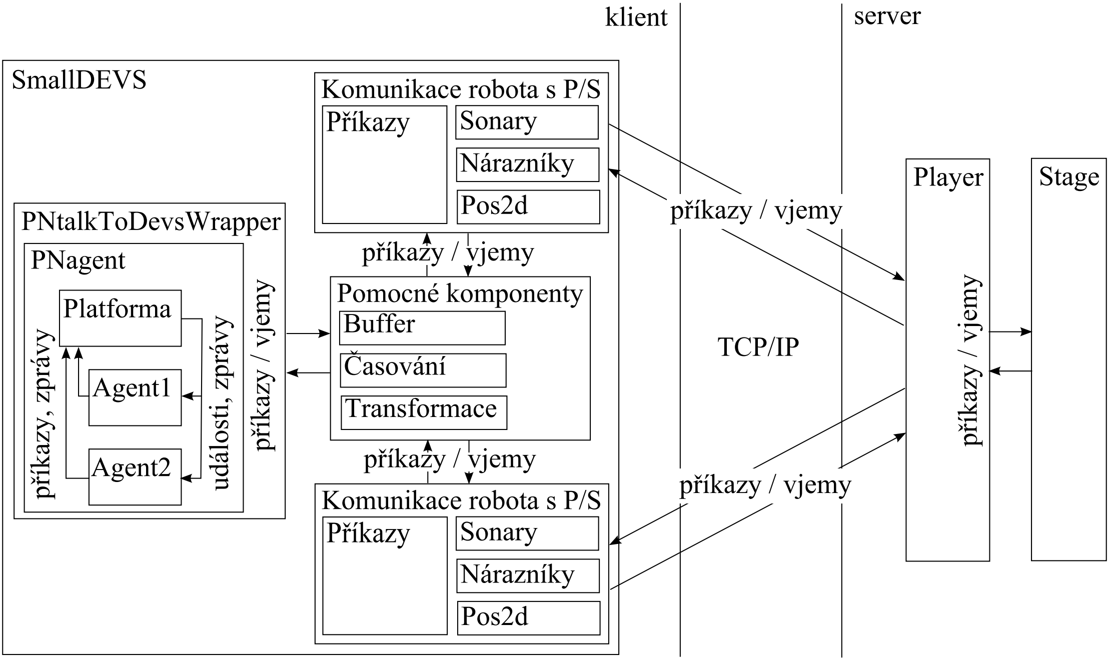
[1] http://playerstage.sourceforge.net/
[2] http://www.activrobots.com/ROBOTS/p2dx.html

5.9.3 Struktura modelu

Jak již bylo zmíněno v úvodu, celý model se skládá z několika heterogenních částí, jejichž propojení ukazuje obrázek 5.17. Část označená na obrázku jako server je kompletně realizována simulátorem Player/Stage. Na straně klienta je spuštěno prostředí Squeak Smalltalk, ve kterém běží systém SmallDEVS.

SmallDEVS [JK06] je systém vyvíjený na VUT v Brně, který je založen na formalismu *DEVS* (Discrete Event System Specification) [ZKP00]. Jedná se o poměrně nízkoúrovňový prostředek s přesnou formální specifikací, založený na systému propojených komponent, který je díky tomu vhodný jako prostředek pro propojování různých heterogenních systémů.

Realizovaný model obsahuje celou řadu komponent, z nichž většina se stará o komunikaci se simulátorem Player/Stage, další poskytují různé transformace příkazů a vyrovnávací paměti pro komunikaci. Samotný systém PNagent, který běží v prostředí PNtalk je zapouzdřen jako atomická komponenta DEVSu [JK07], která přijímá vstupy ze senzorů všech robotů v systému a následně vysílá příkazy.



Obrázek 5.17: Schéma realizovaného modelu tvorby formace mobilních robotů

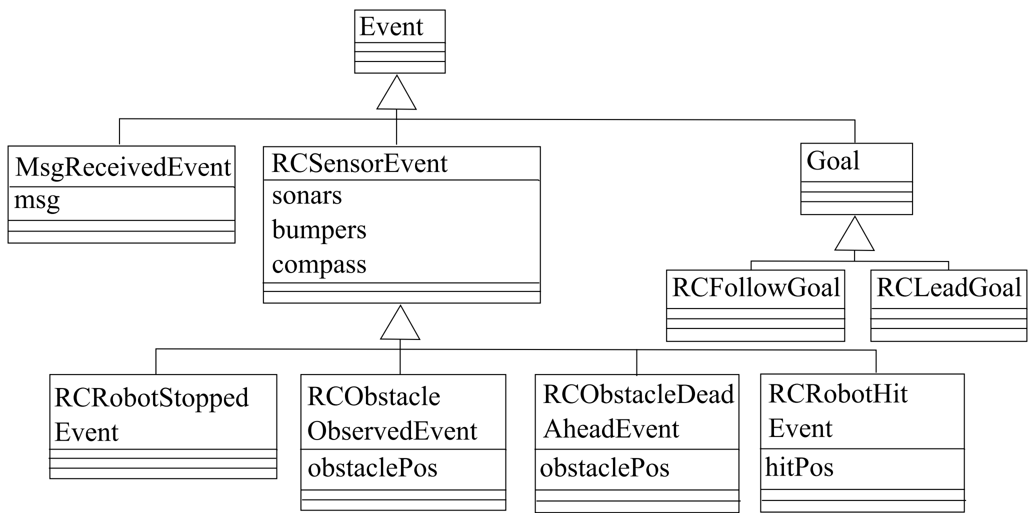
Z pohledu frameworku PNagent tato komunikace vypadá tak, že v objektu třídy *Platform* jsou některá místa označena jako vstupní a jiná jako výstupní. Kdykoliv dorazí data ze senzorů některého z robotů, objeví se v příslušném vstupním místě značka. Tento vstup je pak objektem třídy Platform analyzován a v případě, že je pro daného agenta významný, je vygenerována příslušná událost ve smyslu architektury BDI a předána danému agentovi. Naopak pokud se jeden z agentů rozhodne vykonat akci, zavolá příslušnou metodu objektu třídy Platform, která umístí značku do výstupního místa, kde je převzata obalující komponentou DEVSu a odeslána.

Aby mohla být jednoduše využita komunikační infrastruktura poskytovaná frameworkem PNagent, je celá populace agentů (robotů) umístěna uvnitř jedné komponenty. Ko-

munikace mezi jednotlivými agenty tak probíhá pouze v rámci systému PNagent přes objekt třídy Platform. V případě, že by bylo nutné vytvořit systém jako distribuovaný, není potřeba v samotných agentech provádět žádné změny – komunikace bude pouze probíhat vzdáleně mezi jednotlivými platformami umístěných na různých strojích. V modelu však tato možnost nebyla prozatím realizována.

5.9.4 Předzpracování dat a události

Simulátor Player/Stage zasílá periodicky do příslušné DEVS komponenty data ze senzorů. Tato data jsou následně předána komponentě zapouzdřující systém PNtalk, kde se objeví jako značky ve vstupních místech objektu třídy Platform. Tato data by mohla být předávána v nezměněné podobě objektům agentů, jako událost „nová data ze senzoru“. Jako lepší způsob se však jeví data již při příjmu v platformě předzpracovat a agentům pak zasílat již pouze relevantní události – například že byla zachycena překážka v nebezpečné vzdálenosti před robotem, robot zastavil apod. Na základě těchto událostí se pak přímo vyvolávají konkrétní plány pro řešení dané situace (namísto speciálního plánu, který by pokaždé analyzoval údaje ze senzorů a na základě toho vytvářel obdobné interní události, což by pro rozhodovací proces byla zbytečná zátěž navíc). Události použité v modelu a jejich vztah ukazuje obrázek 5.18.



Obrázek 5.18: Hierarchie událostí použitých v modelu tvorby formace mobilních robotů

Události tvoří tři skupiny. Veškeré události, které nesou informaci z agentových senzorů (robot zastavil, byla zpozorována překážka v testovací zóně, nebezpečná překážka před robotem a konečně robot narazil) s sebou nesou kromě popisu události zároveň hrubá data ze senzorů. Další událost je k dispozici pro příjem zpráv, ta je používána při rádiové komunikaci. Cíle „veď formaci“ a „následuj“ slouží k vyvolávání příslušných plánů po úspěšném ukončení protokolu tvorby formace.

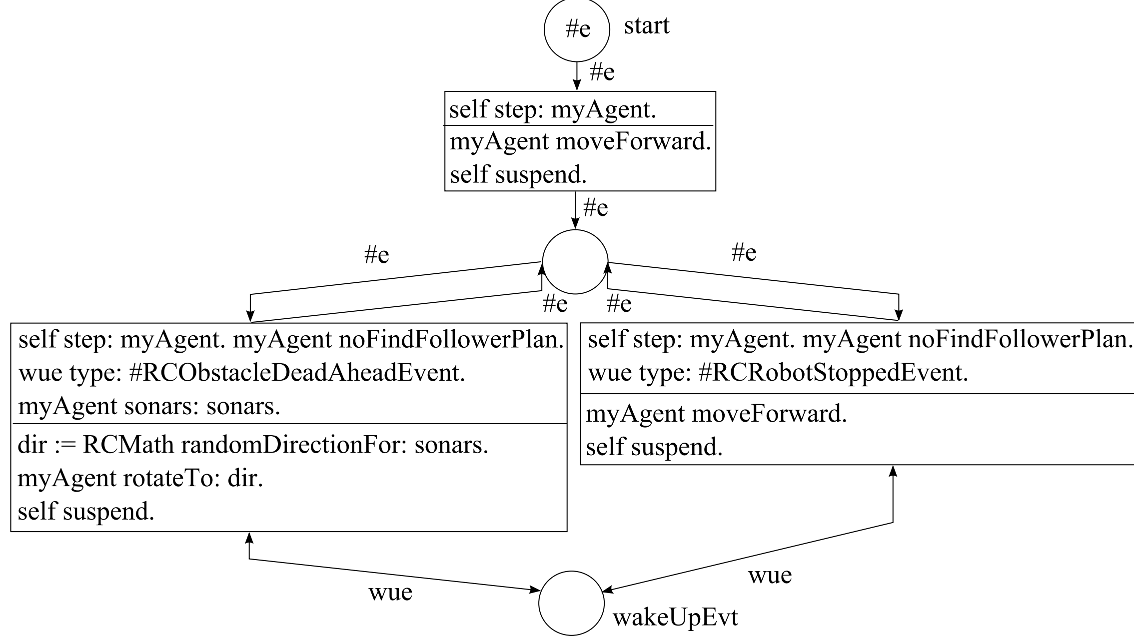
5.9.5 Báze představ a plány

Při současném zadání úlohy není báze představ příliš rozsáhlá – úkolem robotů je pohybovat se náhodně a pouze při kontaktu s překážkou provést předem danou aktivitu. Báze představ

tak obsahuje jen popis ostatních robotů pro možnost navázání komunikace a dále speciální meta-informace pro synchronizaci plánů. Extenzivněji by byla využita v případě, že by roboti vytvářeli mapu svého prostředí, což je jedním z možných budoucích rozšíření této úlohy.

Plány naopak představují klíčovou součást této úlohy. Na obrázcích dále v textu je znázorněno pět hlavních plánů, které zajišťují z velké části splnění úkolu. Pro přehlednost bylo z obrázků vynecháno ošetření některých chybových stavů – například při selhání zaslání zprávy, apod.

Náhodný pohyb agenta s vyhýbáním překážkám je realizován plánem *RCRandomMovePlan*, který ukazuje obrázek 5.19. Tento plán je vytvořen při výskytu události *RCRobotStoppedEvent* a ihned po svém vytvoření aktualizuje bázi znalostí tak, aby další plány pro náhodné procházení nebyly vytvářeny. Jeho funkčnost spočívá v tom, že pokud se robot zastaví, vyšle příkaz jeď vpřed. Pokud se vyskytne překážka v nebezpečné zóně, zvolí se nový směr tak, že se vyberou sonary, které vrací nejvyšší hodnotu (nejvzdálenější překážku, nebo volný prostor) a z nich se jeden zvolí náhodně. Robot se pak natočí tímto směrem. Vlastní výpočty natočení, stejně jako řada dalších výpočtů v modelu jsou z technických důvodů realizovány externě třídou jazyka Smalltalk pojmenované *RCMath* (verze nástroje PNtalk použitá při experimentech zatím nepodporuje některé potřebné konstrukce). V případě, že byl zahájen protokol tvorby formace, je provádění náhodného procházení blokováno negativním predikátem *noFindFollowerPlan*. Při zahájení tohoto protokolu není plán zrušen, pouze čeká na dokončení protokolu. Plán pro náhodné procházení tedy nemá definovány podmínky ukončení, po svém vytvoření existuje po celou dobu života agenta.



Obrázek 5.19: Plán *RCRandomMovePlan* realizující náhodný pohyb robota

V případě, že robot zpozoruje překážku v testovací zóně, spustí agent plán pro zjištění, zda se jedná o překážku, či jiného robota (samozřejmě za předpokladu, že tento protokol již nebyl zahájen jiným agentem). Plán *RCFindFollowerPlan*, jež je ukázán na obrázku 5.20,

zastává funkci role iniciátora protokolu, která je následující:

- Robot se zastaví, zašle data ze svých sonarů ostatním agentům a čeká na odpověď.
- Pokud je odpověď záporná, protokol končí.
- V opačném případě se druhý robot před odesláním zprávy přesunul, analyzuje se tedy aktuální vstup ze sonarů.
- Pokud nebyla zaznamenána žádná změna, zašle se tomuto agentovi oznámení, že se jednalo o náhodnou shodu vstupů a protokol končí.
- V opačném případě se mu zašle pokyn k vytvoření formace, dojde k natočení do předem daného směru a očekává se jeho potvrzení.
- Po obdržení potvrzení se spustí plán pro vedení formace pomocí vytvoření příslušného cíle.

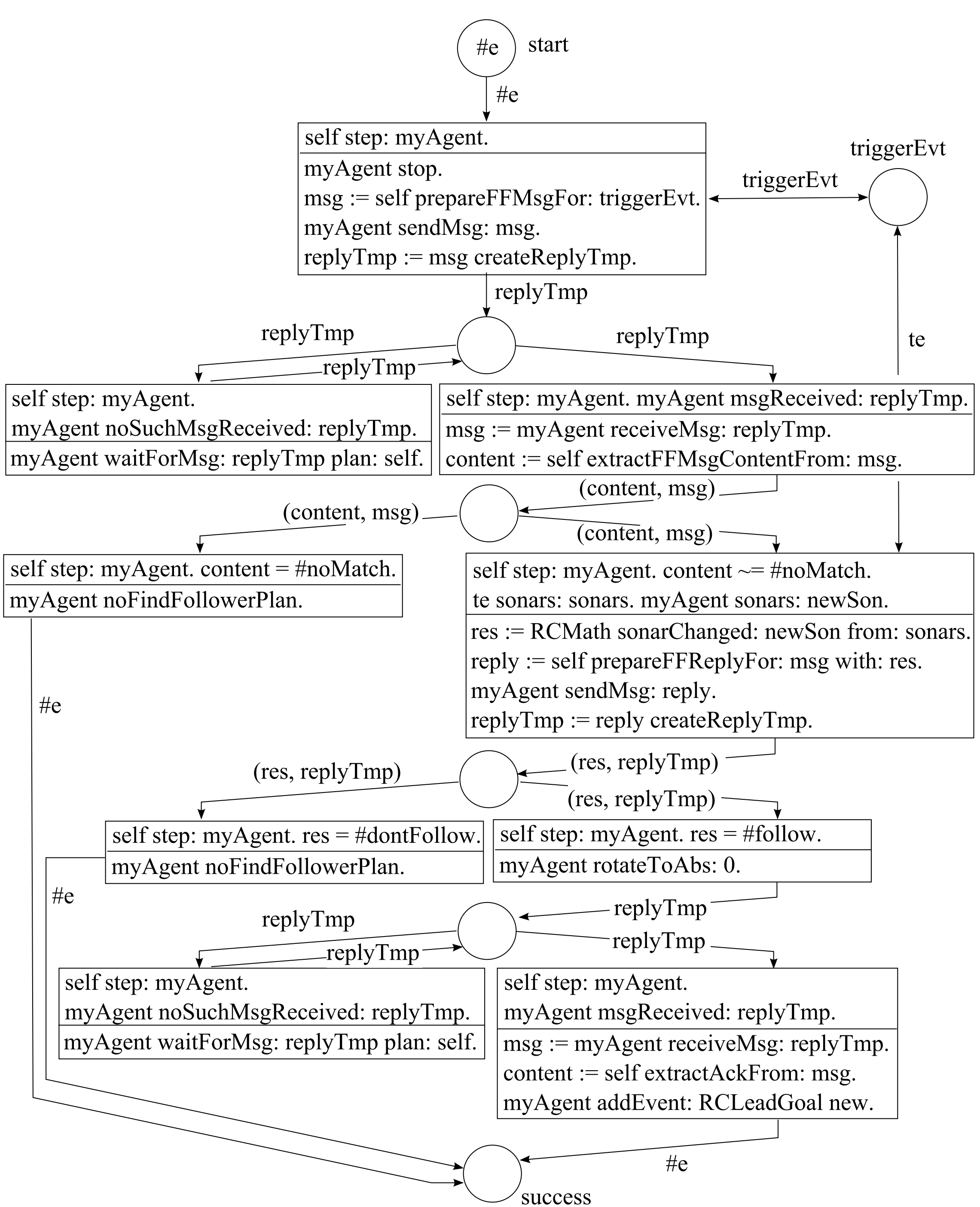
Druhá role protokolu tvorby formace je implementována plánem *RCReplyFindFollowerPlan*. Spouštěcí podmínkou tohoto plánu je, že agent přijme zprávu od iniciátora zahajující protokol. Schéma plánu ukazuje obrázek 5.21 a jeho účel je následující:

- Robot se zastaví a analyzuje data, která obdržel ve zprávě.
- Pokud není nalezena shoda, odešle příslušnou zprávu iniciátorovi a protokol končí.
- V opačném případě se robot natočí směrem, kde by se měl nacházet iniciátor a přijede až na hranici nebezpečné oblasti.
- Poté informuje iniciátora, že byla nalezena shoda a že se přesunul pro ověření.
- Pokud iniciátor nezjistí změnu ve svých senzorech, protokol končí a formace není utvořena.
- V opačném případě se robot natočí do smluveného směru, spustí plán pro sledování a zašle vedoucímu zprávu, že je připraven, čímž je utvořena formace.

Poté co se ustanoví pomocí těchto dvou plánů formace, je u iniciátora na základě cíle zaslaného z plánu *RCFindFollowerPlan* vytvořena instance plánu *RCLeadPlan*. Obsah plánu ukazuje obrázek 5.22. Plán vychází z plánu pro náhodné procházení, avšak kromě příkazu robotovi je vždy vygenerována také zpráva s tímto příkazem pro agenta, který jej následuje. Dále je odchytávána událost přijetí zprávy o tom, že se formace ruší. V případě přijetí této zprávy je plán ukončen a agent pokračuje v náhodném procházení prostoru.

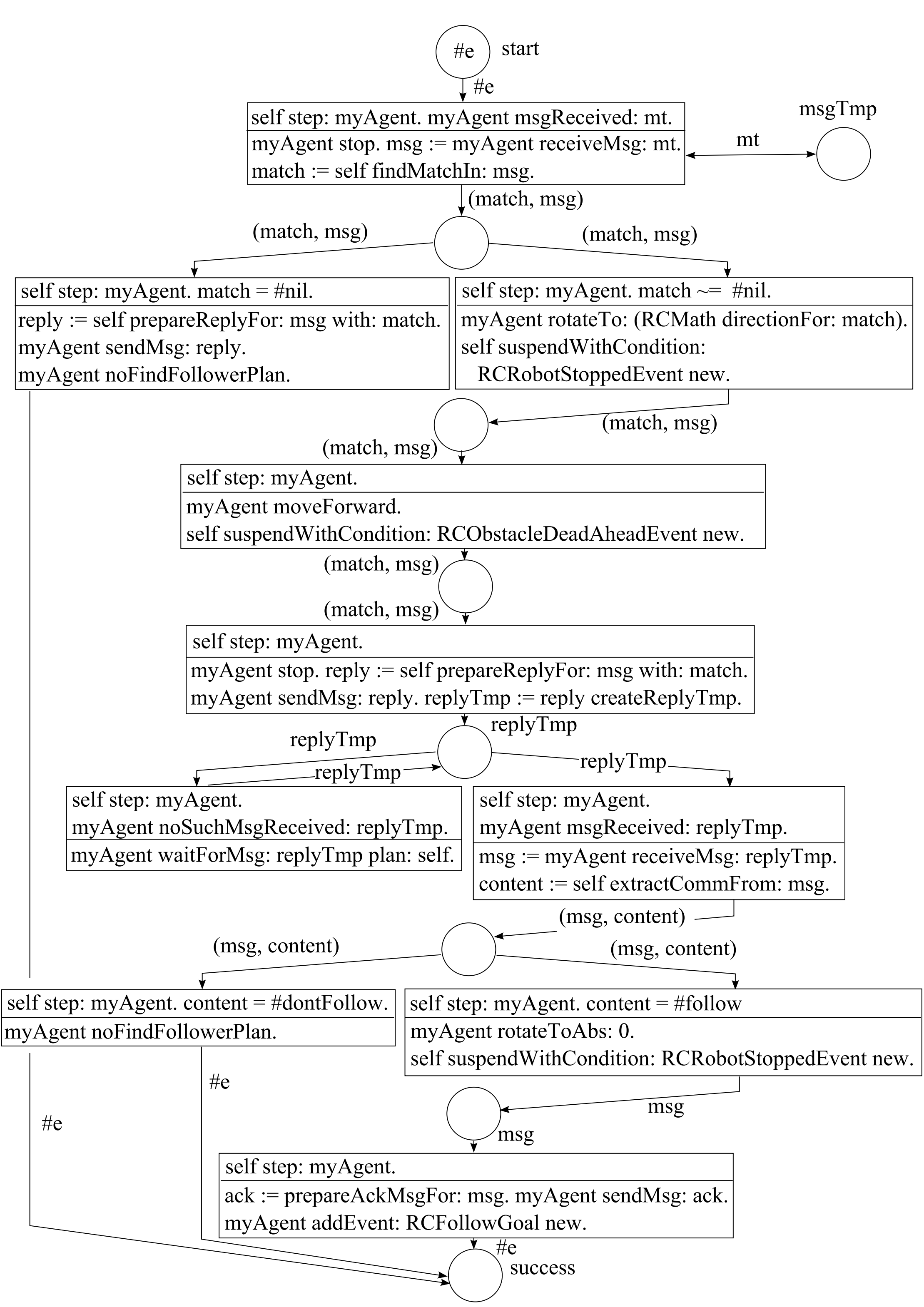
Poslední prezentovaný plán – *RCFollowPlan* je spuštěn na základě cíle *RCFollowGoal*, který je vytvořen v plánu *RCReplyFindFollowerPlan*, pokud je úspěšně vytvořena formace. Spočívá v přijímání zpráv s příkazy, jejich dekódování a provádění. V případě, že se vyskytne událost *RCObstacleDeadAhead* – tedy byla zpozorována překážka těsně před agentem, je odeslána zpráva o zrušení formace a plán je ukončen. Robot se poté dál pohybuje náhodně podle plánu *RCRandomMovePlan*, který byl po dobu pohybu ve formaci zastaven. Schéma plánu *RCFollowPlan* ukazuje obrázek 5.23.

Kromě těchto pěti základních plánů, které realizují vlastní funkčnost modelu, má k dispozici každý agent také několik plánů pro řešení různých selhání – například pro situaci, kdy robot narazil do překážky.

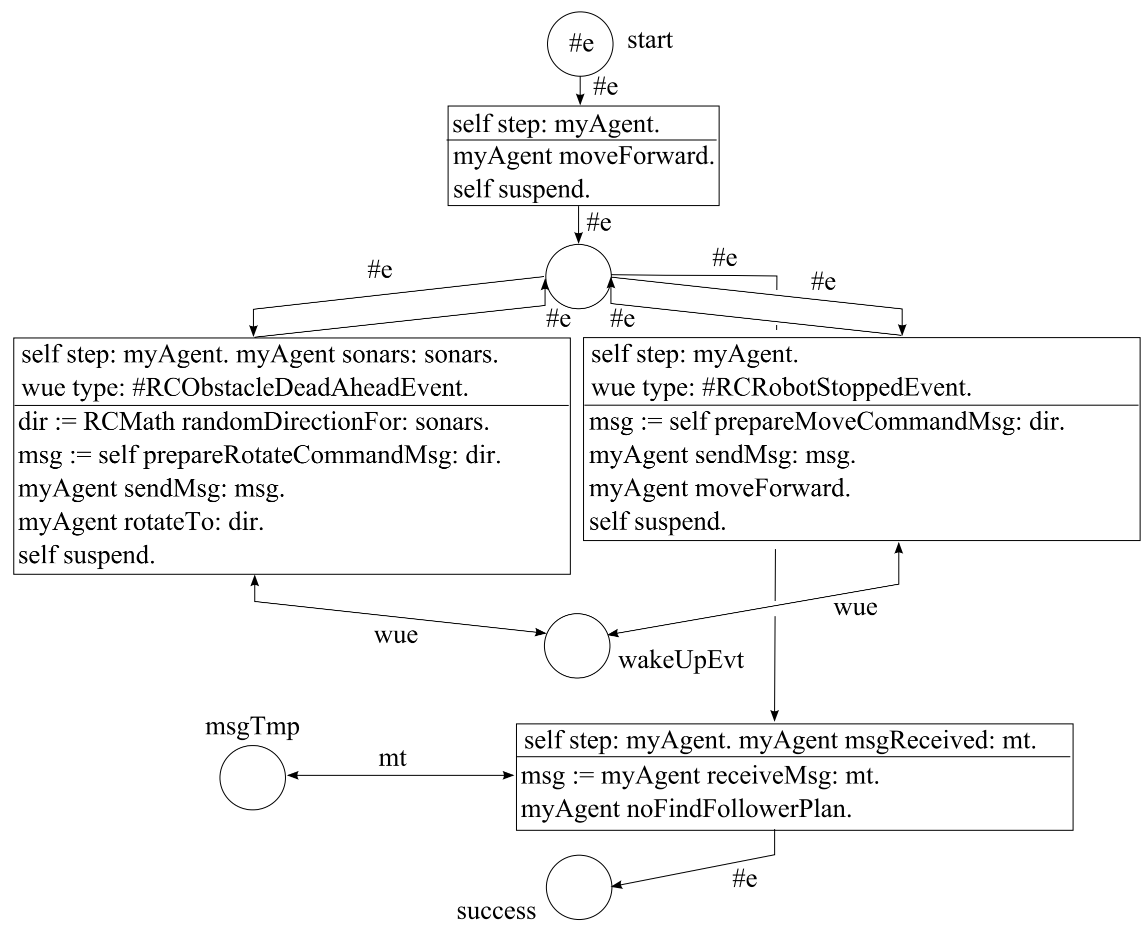


Obrázek 5.20: Plán *RCFindFollowerPlan* pro roli iniciátora protokolu tvorby formace

Pro výběr plánu k provádění je v modelu použita varianta interpretu řízená prioritami plánů, která je v některých případech doplněna explicitní synchronizací (v některých případech, jako například během čekání na příjem zprávy při protokolu tvorby formace, při kterém je daný plán uspaný, není žádoucí, aby agenta v tuto dobu řídily jiné plány s nižší prioritou – konkrétně plán pro náhodné procházení).



Obrázek 5.21: Plán *RCReplyFindFollowerPlan* zodpovědný za druhou roli protokolu tvorby formace

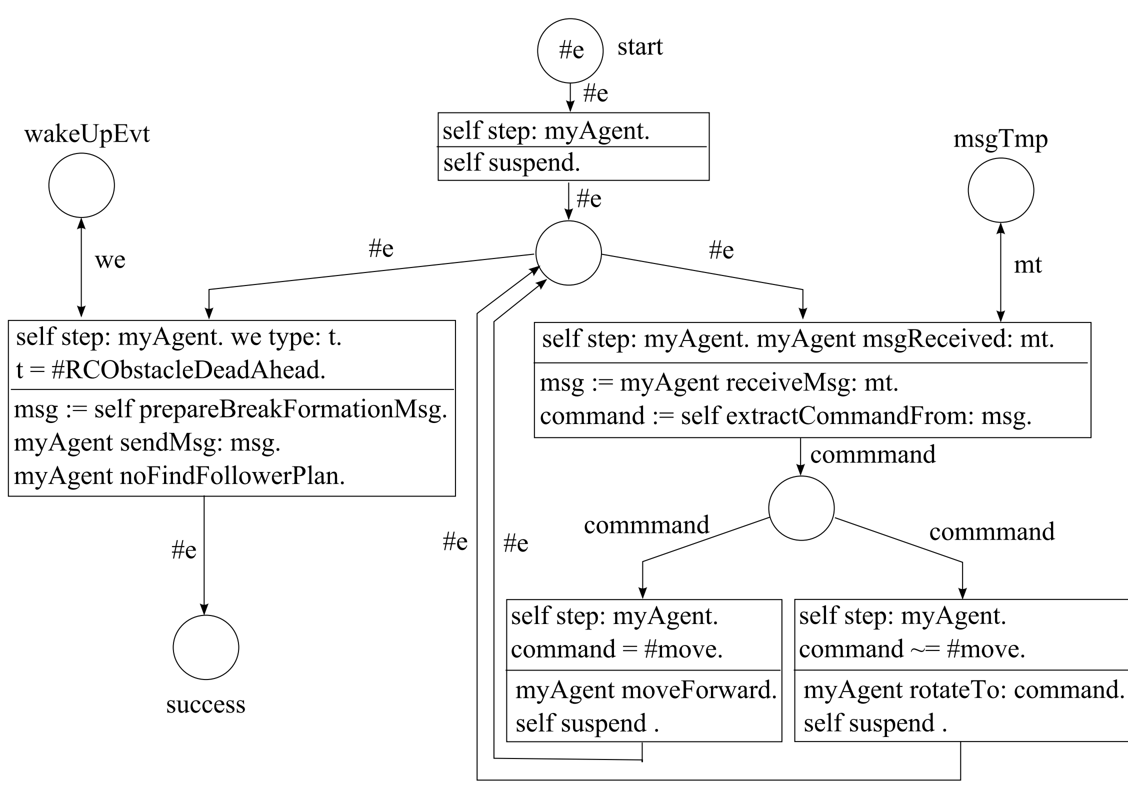


Obrázek 5.22: Plán *RCLeadPlan*, který kromě náhodného procházení přeposílá veškeré příkazy pomocí zpráv

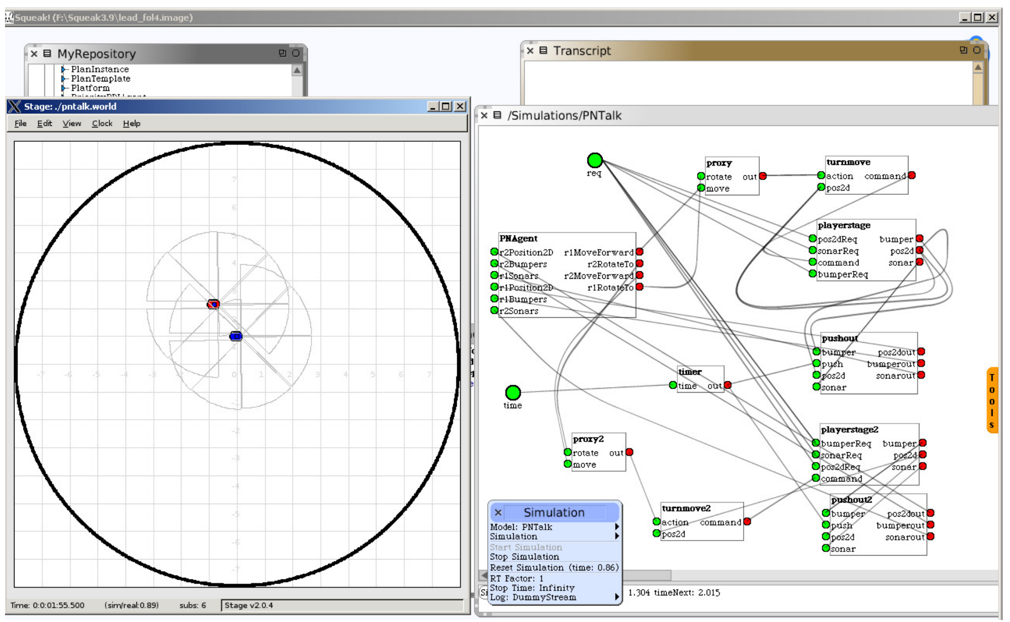
5.9.6 Zhodnocení

Hlavním cílem modelu bylo otestovat funkčnost a aplikovatelnost frameworku PNagent jako celku, zároveň s možnostmi propojení s ostatními technologiemi vyvíjenými v rámci skupiny Modelování a simulace na FIT VUT v Brně. Tento cíl byl splněn (snímek obrazovky průběhu experimentu ukazuje obrázek 5.24). Při návrhu modelu byla doladěna celá řada technických detailů v obou zmíněných oblastech. Hlavní problém, který se prozatím podařilo vyřešit jen částečně, je velmi vysoká náročnost systému PNtalk, ve kterém je framework PNagent implementován, na systémové zdroje. PNtalk se však stále nachází ve fázi prototypu a proto je možné provést celou řadu optimalizací (již v průběhu tvorby modelu a experimentů bylo autory PNtalku dosaženo značného zrychlení).

V porovnání s implementací rozhodovacího jádra agentů pomocí jednodušší reaktivní architektury (jak v původním článku skupiny prof. Zieglera, tak v rámci jiného týmu ze skupiny Modelování a simulace na FIT VUT v Brně, řešícího stejnou úlohu za použití čistého DEVSu, byla použita Subsumpční architektura) je zřejmě největším přínosem snadná rozšiřitelnost funkčnosti pomocí doplnění dalších plánů a využití báze znalostí a relativní intuitivnost popisu řešení úlohy pomocí plánů.



Obrázek 5.23: Plán *RCLeadPlan*, který přijímá zprávy od vedoucího formace a provádí je



Obrázek 5.24: Snímek obrazovky s výstupem simulátoru Player/Stage a modelu v prostředí SmallDEVS

Úspěšnost tvorby formace samozřejmě není stoprocentní, což vyplývá z poměrně dlouhé reakční doby systému a tím způsobené nepřesnosti informací, ze kterých agent vychází. Co se týče možných rozšíření realizovaného modelu, tak zřejmě nejzajímavější variantou je rozšíření na více než dva roboty – to však bude vyžadovat zrychlení systému PNtalk, případně distribuci jednotlivých agentů na více počítačů (a s tím související nutnost doplnit možnost zasílat zprávy mezi platformami v síťovém prostředí). Další drobná rozšíření by se mohla týkat samotného protokolu tvorby formace – zejména vylepšení tvaru formace po jejím vytvoření (po skončení protokolu se roboti nacházejí poměrně blízko sebe, takže formace se často záhy rozpadá) a pečlivějšího předzpracování hrubých dat ze senzorů při ověřování, zda se jedná o překážku či robota (poté co bylo ověřeno, že se o robota nejedná, tuto překážku nadále při vyhledávání robota ignorovat).

5.10 Jiné přístupy k modelování multiagentních systémů Petriho sítěmi

V následujícím textu budou stručně zmíněny jiné přístupy k modelování multiagentních systémů Petriho sítěmi. Tato část sice spadá spíše do teoretické části disertační práce, avšak byla umístěna záměrně až za prezentaci frameworku PNagent, aby mohlo provedeno jeho srovnání s přístupy zde uvedenými.

Jednou z prvních publikací, která se spojením Petriho sítí a agentních technologií zabývala, je článek [Hol95]. Využití Petriho sítí (konkrétně barvených Petriho sítí) je zde však popsáno poměrně vágně, stejně tak jako architektura použitých agentů. Na tuto práci navázala stejná skupina článkem [WH02], kde je barvená Petriho síť použita pro modelování konkrétního multiagentního systému, řešícího úlohu, kterou autoři nazvali *Packet-World*. V modelu se autoři zaměřují především na sociální interakce agentů, konstatují vhodnost modelování multiagentních systémů pomocí CPN. Výsledkem práce však není žádný znovupoužitelný systém, pouze samotný model.

Obecný framework pro modelování multiagentního systému pomocí barvených Petriho sítí popisuje prof. Moldt ve článku [MW97]. Formalismus barvených Petriho sítí je v něm rozšířen o koncepty objektové orientace, avšak poměrně konzervativním způsobem, tak aby bylo možné tyto sítě simulovat pomocí nástrojů pro CPN. Architektura modelovaného multiagentního systému vychází z již dříve zmíněného jazyka AOP [Sho93], tedy agentního systému založeného na teoretickém usuzování. Pro reprezentaci představ a schopností agenta je použito prostředků logického programování. Petriho sítě tak představují prostředek pro vyjádření infrastruktury frameworku (která je statická), zatímco vlastní programy jsou psány pomocí jazyka obdobného jazyku Prolog. Framework PNagent se s tímto přístupem tedy shoduje pouze v použití Petriho sítí rozšířených o objektovou orientaci; architektura agentů i způsob vyjádření agentních programů jsou zcela odlišné.

Dalším přístupem, modelujícím tentokrát agenty založené na architektuře BDI je jazyk nazvaný *ADLMAS* (Architecture Description Language for Multi-Agent Systems) [YC06]. Tento jazyk je založen na formalismu nazvaném OPN (Object-Oriented Petri nets). OPN představuje variantu Petriho sítí, která vychází z CPN a má s nimi mnoho podobností. Objektová rozšíření jsou zavedeny prakticky jen pro potřeby strukturování sítě, v důsledku toho je celý systém opět zcela statický – není možné během simulace přidávat či ubírat agenty, agenti jsou pevně propojeni hranami, po kterých jsou zasílány zprávy (reprezentované jako značky). Systém je výrazně zaměřen na statické ověřování vlastností modelu, dynamické simulaci je věnována malá pozornost. V důsledku toho je architektura i účel

tohoto systému zcela odlišná od frameworku PNagent.

Kromě těchto prací využívá řada autorů Petriho sítě jako prostředek pro vyjádření pouze některých částí systému, obvykle komunikace, případně synchronizace plánů. Příklady takových přístupů jsou popsány například ve článcích [Fer99] a [CCF+99].

Lze tedy říci, že to, co nejvíce odlišuje framework PNagent od všech projektů zmíněných výše je jeho dynamičnost, kterou se daleko více podobá systémům vytvořených v běžných programovacích jazycích. To je dáno zejména vlastnostmi použitých objektově orientovaných Petriho sítí. Framework PNagent tak představuje praktickou ukázku hlavní myšlenky a původního cíle OOPN – přiblížení matematicky definovaného formálního aparátu běžným programovacím jazykům.

Kapitola 6

Závěr

Předložená disertační práce se zabývá problematikou modelování inteligentních agentů pomocí prostředků objektově orientovaných Petriho sítí. Celá práce bude nyní shrnuta v několika krocích. Nejprve uvedu použitý přístup k řešení, dále dosažené výsledky a konečně naznačím možnosti dalšího výzkumu.

6.1 Zvolený přístup k řešení

Výzkum v oboru multiagentních systémů se zaměřuje na celou řadu oblastí, které se dají rozdělit do dvou hlavních skupin – hledání vhodné vnitřní architektury agentů a výzkum interakcí mezi kolekcemi agentů. Ve své práci jsem se rozhodl zaměřit na první oblast – návrh vnitřní architektury agentů. K tomuto návrhu existuje celá řada přístupů, které jsou diskutovány ve třetí kapitole. Jako jeden z nejperspektivnějších přístupů se jeví architektura BDI, kterou se poměrně podrobně zabývá kapitola čtvrtá. Současné systémy implementující architekturu BDI se obvykle snaží buď o sblížení s klasickým softwarovým inženýrstvím, anebo se zaměřují na hledání vhodného formálního popisu.

Použití objektově orientovaných Petriho sítí a nástroje PNtalk pro návrh a vývoj umožňuje vhodně zkombinovat oba přístupy. Mým hlavním cílem v této práci bylo navrhnout experimentální prototypový systém, který by umožnil vývoj agentů s architekturou BDI podle metodiky vývoje založeného na modelech, a zároveň bude sám snadno modifikovatelný, což by umožnilo experimenty se samotnou architekturou.

Vzhledem k tomu, že systém PNtalk je sám experimentální nástroj ve fázi vývoje, ke kterému existuje zatím poměrně málo materiálů týkajících se metodiky použití, promítly se některé myšlenky a zkušenosti získané při této práci zpětně do obecné metodiky modelování pomocí jazyka PNtalk.

6.2 Dosažené výsledky

Dosažené výsledky lze rozdělit do dvou oblastí. Hlavním výsledkem práce je návrh architektury frameworku *PNagent*, prezentovaný v páté kapitole. Framework PNagent umožňuje modelování inteligentních agentů založených na architektuře BDI pomocí objektově orientovaných Petriho sítí. Hlavním rysem tohoto frameworku je otevřenost – tvůrce konkrétního modelu má možnost snadno podle potřeby modifikovat chování samotného frameworku pomocí stejných grafických prostředků, kterými vytváří samotný model. To umožňuje využívat framework kromě tvorby modelů také pro experimenty se samotnou architekturou BDI.

Oproti většině existujících systémů je PNagent založen na formálním základu, poskytovaným Petriho sítěmi, což umožňuje mimo jiné vysokoúrovňovou formální analýzu. Další výhodou je intuitivnost grafického vyjádření modelu a přirozený popis paralelismu při zachování kompaktnosti celého systému.

Návrh frameworku byl ověřen prototypovou implementací pomocí nástroje PNtalk, včetně několika ukázkových modelů, které slouží pro demonstraci principů modelování při použití tohoto frameworku a také porovnání s existujícími systémy. Kromě těchto ukázkových modelů byla vytvořena také jedna rozsáhlejší aplikace z oblasti řízení mobilních robotů. Zde byl kromě funkčnosti a použitelnosti samotného frameworku ukázán také přístup k multiparadigmatickému modelování pomocí nástrojů vyvíjených v rámci skupiny Modelování a simulace na FIT VUT v Brně.

Druhou oblastí přínosu práce je prokázání a demonstrace vhodnosti použití objektově orientovaných Petriho sítí a systému PNtalk pro modelování inteligentních systémů a zároveň příspěvek k obecné metodice modelování pomocí OOPN a nástroje PNtalk, zejména zavedení konceptu negativních predikátů do jazyka PNtalk, diskutovaný ve druhé kapitole. Negativní predikáty umožňují v řadě případů použít prostředků Petriho sítí místo inskripčního jazyka, čímž mohou výrazně usnadnit modelování. Vzhledem k tomu, že představují jistý druh inhibičních hran, mohlo by jejich použití v modelech ztížit některé druhy analýzy založené na invariantech. Proto byl vytvořen algoritmus, který umožňuje model využívající negativní predikáty převést na ekvivalentní model obsahující pouze výrazy inskripčního jazyka.

6.3 Možnosti dalšího výzkumu

Na výzkum prezentovaný v této práci je možné navázat v několika oblastech. První z nich je využití frameworku PNagent pro experimenty se samotnou architekturou BDI, zejména v oblasti meta-plánování, tedy výběru relevantního plánu pro danou událost a volby záměru pro provádění ze struktury záměrů. Jako zajímavá možnost se jeví pro tento účel využití meta-úrovňové architektury nástroje PNtalk. Další experimenty se mohou týkat porovnání současné implementace interpretu s nezávislým zpracováním událostí s klasickým uzavřeným BDI-cyklem, podle několika provedených experimentů se zdá verze implementovaná PNagentem výhodnější. Rozšíření se mohou týkat také práce s bází představ, zejména v oblasti její automatické revize.

Další oblastí spíše implementačního charakteru je propracování komunikační infrastruktury, až do dosažení plné kompatibility s normami FIPA a tím interoperatibility s ostatními agentními systémy, zejména platformou JADE – implementované modely byly doposud spouštěny pouze lokálně.

Vzhledem k problémům s efektivitou prototypové implementace nástroje PNtalk, na které jsme narazili při tvorbě modelu řídícího mobilní roboty, se jeví jako zajímavá otázka možnosti transformace modelů do jiného formalismu, například do čistého DEVSu. PNagent by pak sloužil pro návrh a otestování rozhodovacího jádra agenta, které by pak pro samotný běh bylo přeloženo do nízkoúrovňového, efektivnějšího jazyka.

Posledním směrem je možnost vysokoúrovňové formální verifikace modelů, vytvořených pomocí frameworku PNagent – této problematice se věnuje jiná výzkumná skupina na FIT VUT v Brně.

Literatura

[AM04] K. A. Amin and A. R. Mikler. Agent-based Distance Vector Routing: A Resource Efficient and Scalable approach to Routing in Large Communication Networks. *Journal of Systems and Software*, 71(3):215–227, 2004.

[Aus75] J. Austin. *How to Do Things With Words (William James Lectures).* Harvard University Press, September 1975.

[BdOJB$^+$02] R. H. Bordini, R. de O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. AgentSpeak(XL): Efficient Intention Selection in BDI Agents via Decision-theoretic Task Scheduling. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002)*, pages 1294–1302. ACM Press, 2002.

[BF89] R. A. Brooks and A. M. Flynn. Fast, Cheap and Out of Control: A Robot Invasion of the Solar System. *Journal of the British Interplanetary Society*, 42:478–485, 1989.

[BH05] R. H. Bordini and J. F. Hübner. BDI Agent Programming in AgentSpeak Using *Jason* (Tutorial Paper). In *Proceedings of the Sixth International Workshop on Computational Logic in Multi-Agent Systems (CLIMA VI)*, pages 143–164, 2005.

[BIP91] M. E. Bratman, D. Israel, and M. Pollack. Plans and Resource–Bounded Practical Reasoning. In R. Cummins and J. L. Pollock, editors, *Philosophy and AI: Essays at the Interface*, pages 1–22. The MIT Press, Cambridge, Massachusetts, 1991.

[Bow96] J. P. Bowen. *Formal Specification and Documentation using Z: A Case Study Approach.* International Thomson Computer Press, 1996.

[BR01] F. Bellifemine and G. Rimassa. Developing Multi-agent Systems with a FIPA-compliant Agent Framework. *Software: Practice and Experience*, 31(2):103–128, 2001.

[Bra87] M. E. Bratman. *Intention, Plans, and Practical Reason.* Harvard University Press, Cambridge, MA, 1987.

[BRHL99] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents – Components for Intelligent Agents in Java, 1999. AgentLink News Letter.

[Bro86] R. A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[BS92] B. Burmeister and K. Sundermeyer. Cooperative Problem-Solving Guided by Intentions and Perception. In E. Werner and Y. Demazeau, editors, *Decentralized A.I. 3: Proceedings of the Third European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 77–92. North-Holland, Amsterdam, 1992.

[BS97] P. Bretier and M. D. Sadek. A Rational Agent as the Kernel of a Cooperative Spoken Dialogue System: Implementing a Logical Theory of Interaction. In *ECAI '96: Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, pages 189–203, London, UK, 1997. Springer-Verlag.

[CCF+99] R. S. Cost, Y. Chen, T. Finin, Y. K. Labrou, and Y. Peng. Modeling Agent Conversations with Colored Petri Nets. In *Working notes of the Autonomous Agents '99 Workshop on Specifying and Implementing Conversation Policies*, Seattle, Washington, May 1999.

[Cla87] K. L. Clark. Negation as Failure. In *Readings in nonmonotonic reasoning*, pages 311–325. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[Con89] J. Connell. A Colony Architecture for an Artificial Creature. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1989.

[DA92] R. David and H. Alla. *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Das08] M. Dastani. 2APL: a Practical Agent Programming Language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[Den87] D. C. Dennett. *The Intentional Stance.* The MIT Press, Cambridge, Massachusetts, 1987.

[dHL00] M. d'Inverno, K. V. Hindriks, and M. Luck. A Formal Architecture for the 3APL Agent Programming Language. In *ZB '00: Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 168–187, London, UK, 2000. Springer-Verlag.

[dLG+04] M. d'Inverno, M. Luck, M. P. Georgeff, D. Kinny, and M. Wooldridge. The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53, 2004.

[DW03] I. Dickinson and M. Wooldridge. Towards Practical Reasoning Agents for the Semantic Web. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 827–834, New York, NY, USA, 2003. ACM.

[Češ94] M. Češka. *Petriho sítě.* CERM, Brno, Česká republika, 1994.

[Fer92] I. A. Ferguson. TouringMachines: Autonomous Agents with Attitudes. *IEEE Computer*, 25(5):51–55, 1992.

[Fer99] J. Ferber. *Multi-Agent Systems.* Addision-Wesley, UK, 1999.

[FFMM94] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, MD, USA, 1994. ACM Press.

[FIP01] FIPA. *FIPA ACL Message Structure Specification.* FIPA, 2001.

[FIP02a] FIPA. *FIPA Abstract Architecture Specification.* FIPA, 2002.

[FIP02b] FIPA. *FIPA Contract Net Interaction Protocol Specification.* FIPA, 2002.

[FIP04] FIPA. *FIPA Agent Management Specification.* FIPA, 2004.

[Fis94] M. Fisher. A Survey of Concurrent METATEM – the Language and its Applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic - Proceedings of the First Intemational Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Heidelberg, Germany, 1994.

[FN71] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach in the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208, 1971. Reprint: Luger, Computation and Intelligence, 1995.

[GI89] M. P. Georgeff and F. F. Ingrand. Decision-making in an Embedded Reasoning System. Technical Report 04, Australian Artificial Intelligence Institute, Melbourne, Australia, November 1989.

[GL87] M. P. Georgeff and A. L. Lansky. Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, USA, July 1987. Morgan Kaufmann Publishers.

[GR96] M. P. Georgeff and A. S. Rao. A profile of the Australian Artificial Intelligence Institute. *IEEE Expert: Intelligent Systems and Their Applications*, 11(6):89–92, 1996.

[Gru93] T. R. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers.

[GVH03] B. P. Gerkey, R. T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

[Hin62] J. Hintikka. *Knowlegde and Belief.* Cornell University Press, Ithaca, NY, 1962.

[HM03] E. I. Hsu and D. L. Mcguinness. Wine Agent: Semantic Web Testbed Application. In *Proceedings of 2003 Description Logics (DL2003)*, pages 5–7. CEUR-WS.org, 2003.

[Hol95] T. Holvoet. Agents and Petri Nets. *Petri Net Newsletter*, 49, 1995.

[Hub99] M. J. Huber. JAM: A BDI-theoretic Mobile Agent Architecture. In *Proceedings of The Third International Conference on Autonomous Agents*, pages 236–243, Seattle, WA, 1999.

[HZ04] X. Hu and B. P. Zeigler. Model Continuity to Support Software Development for Distributed Robotic Systems: a Team Formation Example. In *Journal of Intelligent & Robotic Systems, Theory & Application, Special Issue: Multiple and Distributed Cooperating Robots*, pages 71–87, 2004.

[Jan98] V. Janoušek. *Modelování objektů Petriho sítěmi.* PhD thesis, VUT v Brně, Brno, CZ, 1998.

[Jen93] N. R. Jennings. Specification and Implementation of a Belief-Desire-Joint-Intention Architecture for Collaborative Problem Solving. *International Journal of Intelligent and Cooperative Information Systems*, 2(3):289–318, 1993.

[Jen97] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Springer, 1997.

[JK03] V. Janoušek and R. Kočí. PNtalk: Concurrent Language with MOP. In *Proceedings of the CS&P'2003 Workshop*, pages 271–282. Warsaw University, 2003.

[JK06] V. Janoušek and E. Kironský. Exploratory Modeling with SmallDEVS. In *Proc. of ESM 2006*, pages 122–126. EUROSIS, 2006.

[JK07] V. Janoušek and R. Kočí. Embedding Object-Oriented Petri Nets into a DEVS-based Simulation Framework. In *Proceedings of the 16th International Conference on System Science*, volume 1, pages 386–395. Wroclaw University of Technology, 2007.

[Kel94] J. Kelemen. *Strojovia a agenty.* Archa, Bratislava, SK, 1994.

[KHS97] H. Kargupta, I. Hamzaoglu, and B. Stafford. Scalable, Distributed Data Mining - An Agent Architecture. In *Proceedings the Third International Conference on the Knowledge Discovery and Data Mining*, pages 211–214, Menlo Park, California, 1997. AAAI Press.

[Kit98] H. Kitano, editor. *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Computer Science*. Springer, 1998.

[KMZJ07] R. Kočí, Z. Mazal, F. Zbořil, and V. Janoušek. Modeling Deliberative Agents using Object Oriented Petri Nets. In *Proceedings of the 7th ISDA*, pages 15–20. IEEE Computer Society, 2007.

[Koč04] R. Kočí. *Metody a nástroje pro implementaci otevřených simulačních systémů.* PhD thesis, VUT v Brně, Brno, CZ, 2004.

[Kou01] A. Koumpis. The European IST ExPlanTech project. *Ubiquity*, 2(36):1, 2001.

[Kub04] A. Kubík. *Inteligentní agenty.* Computer Press, Brno, Česká republika, 2004.

[LHDK94] J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny. UM-PRS: an Implementation of the Procedural Reasoning System for Multirobot Applications. In *Proceedings of the Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS'94)*, pages 842–849, Houston, Texas, USA, 1994.

[LTO03] V. Lesser, M. Tambe, and Ch. L. Ortiz, editors. *Distributed Sensor Networks: A Multiagent Perspective.* Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[MB02] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[MJK08] Z. Mazal, V. Janoušek, and R. Kočí. Enhancing the PNtalk Language with Negative Predicates. In *Proceedings of MOSIS '08*, pages 28–34. MARQ, 2008.

[MKJZ08a] Z. Mazal, R. Kočí, V. Janoušek, and F. Zbořil. Modelling Intelligent Agents for Autonomic Computing in PNagent Framework, *International Journal of Autonomic Computing (2008).* Submitted for review.

[MKJZ08b] Z. Mazal, R. Kočí, V. Janoušek, and F. Zbořil. PNagent: a Framework for Modelling BDI Agents using Object Oriented Petri Nets. In *Proceedings of the 8th ISDA.* IEEE Computer Society, (to be published November 2008).

[MP01] S. Moss and P.Davidsson, editors. *Multi-Agent-Based Simulation, Second International Workshop, MABS 2000, Boston, MA, USA, July, 2000, Revised and Additional Papers*, volume 1979 of *Lecture Notes in Computer Science.* Springer, 2001.

[Mul97] J. P. Muller. *The Design of Intelligent Agents: A Layered Approach.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[MVB03] Á. F. Moreira, R. Vieira, and R. H. Bordini. Extending the Operational Semantics of a BDI Agent-Oriented Programming Language for Introducing Speech-Act Based Communication. In *DALT 2003 : Declarative Agent Languages and Technologies*, pages 135–154. Springer, Berlin, Germany, 2003.

[MvdA05] N. Mulyar and W.M.P. van der Aalst. Patterns in Colored Petri Nets. BETA Working Paper Series WP 139, Eindhoven University of Technology, Eindhoven, 2005.

[MW97] D. Moldt and F. Wienberg. Multi-Agent-Systems Based on Coloured Petri Nets. In *Lecture Notes in Computer Science: 18th International Conference on Application and Theory of Petri Nets*, pages 82–101. Springer-Verlag, 1997.

[PBL03] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a BDI-Infrastructure for JADE Agents. *EXP – in Search of Innovation*, 3(3):76–85, 2003.

[Per04] J. Peregrín. *Logika a logiky.* Academia, Praha, Česká republika, 2004.

[Rao96] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In Rudy van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

[Rei01] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* MIT Press, Cambridge, MA, USA, 2001.

[RG92] A. S. Rao and M. P. Georgeff. An Abstract Architecture for Rational Agents. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 439–449, Cambridge, MA, USA, 1992. Morgan Kaufmann.

[RN02] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition).* Prentice Hall, December 2002.

[SBD+00] V. S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems.* MIT Press/AAAI Press, Cambridge, MA, USA, 2000.

[Sho90] Y. Shoham. Agent-oriented programming. Technical Report STAN-CS-1335-90, Computer Science Department, Stanford University, Stanford, CA, USA, 1990.

[Sho93] Y. Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993.

[WH02] D. Weyns and T. Holvoet. A Colored Petri Net for a Multi-Agent Application. In *Components, Objects and Agents, MOCA'02.* Computer Science Department, Aarhus University, 2002.

[Win05] M. Winikoff. JACK Intelligent Agents: An Industrial Strength Platform. In R.H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 175–193. Springer, 2005.

[WJ95] M. J. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[Woo01] M. J. Wooldridge. *Reasoning about Rational Agents.* MIT Press, Cambridge, MA, USA, 2001.

[Woo02] M. J. Wooldridge. *Introduction to MultiAgent Systems.* John Wiley and Sons, 2002.

[WOO07] D. Weyns, A. Omicini, and J. Odell. Environment as a First Class Abstraction in Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.

[YC06] Z. Yu and Y. Cai. Object-Oriented Petri nets Based Architecture Description Language for Multi-agent Systems. *International Journal of Computer Science and Network Security*, 6(1B):123–131, 2006.

[Zbo04] F. Zbořil. *Plánování a komunikace v multiagentních systémech.* PhD thesis, VUT v Brně, Brno, CZ, 2004.

[ZKP00] B. P. Zeigler, T. Gon Kim, and H. Praehofer. *Theory of Modeling and Simulation.* Academic Press, Inc., Orlando, FL, USA, 2000.

Přehled publikací autora

[MZ06] Z. Mazal, F. V. Zbořil. Modelling Sensor Networks using Multiagent Systems. In: *NETSS 2006*, pages 15-18, Ostrava, CZ, MARQ, 2006.

[Maz06] Z. Mazal. Multiagent System for Searching in FOAF Networks. In: *Proceedings of the 12th Conference and Competition STUDENT EEICT 2006 Volume 4*, pages 481-485, Brno, CZ, FIT VUT, 2006.

[MJZ07] Z. Mazal, P. Jurka, F. V. Zbořil. Framework for Intelligent Systems Education Support at FIT BUT. In: *ASIS 2007*, pages 173-176, Ostrava, CZ, MARQ, 2007.

[KMZJ07] R. Kočí, Z. Mazal, F. Zbořil, and V. Janoušek. Modeling Deliberative Agents using Object Oriented Petri Nets. In *Proceedings of the 7th ISDA*, pages 15–20. IEEE Computer Society, 2007.

[MJK08] Z. Mazal, V. Janoušek, and R. Kočí. Enhancing the PNtalk Language with Negative Predicates. In *Proceedings of MOSIS '08*, pages 28–34. MARQ, 2008.

[MKJZ08b] Z. Mazal, R. Kočí, V. Janoušek, and F. Zbořil. PNagent: a Framework for Modelling BDI Agents using Object Oriented Petri Nets. In *Proceedings of the 8th ISDA*. IEEE Computer Society, (to be published November 2008).

[MKJZ08a] Z. Mazal, R. Kočí, V. Janoušek, and F. Zbořil. Modelling Intelligent Agents for Autonomic Computing in PNagent Framework, *International Journal of Autonomic Computing (2008).* Submitted for review.

Příloha A

Syntax jazyka PNtalk

Textová verze jazyka PNtalk definuje kromě syntaxe inskripčního jazyka též syntax popisu struktury Objektově orientované Petriho sítě. Syntax popisů sítí, syntax struktury sítí a celého systému tříd formálně definujeme rozšířenou Backus-Naurovou formou. Zápis [...] znamená, že "..." se může vyskytnout nepovinně, [...]* znamená libovolně násobný nepovinný výskyt "...", [...]+ znamená výskyt "..." jednou nebo víckrát, ...—...[—...]* znamená výběr jedné z variant. Řetězce v uvozovkách odpovídají lexikálním symbolům. Výchozí symbol je classes.

```
classes: [class]* "main" id [class]*
class: "class" classhead [objectnet] [methodnet|constructor|sync|inhib]*
classhead: id "is_a" id

objectnet: "object" net
methodnet: "method" message net
constructor: "constructor" message net
sync: "sync" message [cond] [precond] [guard] [postcond]
inhib: "inhibitor" message [cond] [guard]
message: id | binsel id | [keysel id]+
net: [place|transition]*

place: "place" id"(" [initmarking] ")" [init]
init: "init" "{" initaction "}"
initmarking: multiset
initaction: [temps] exprs

transition: "trans" id [cond] [precond] [guard] [action] [postcond]
cond: "cond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
precond: "precond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
postcond: "postcond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
guard: "guard" "{" expr3 "}"
action: "action" "{" [temps] exprs "}"
arcexpr: multiset

multiset: [n "‘"] c ["," [n "‘"] c]*
```

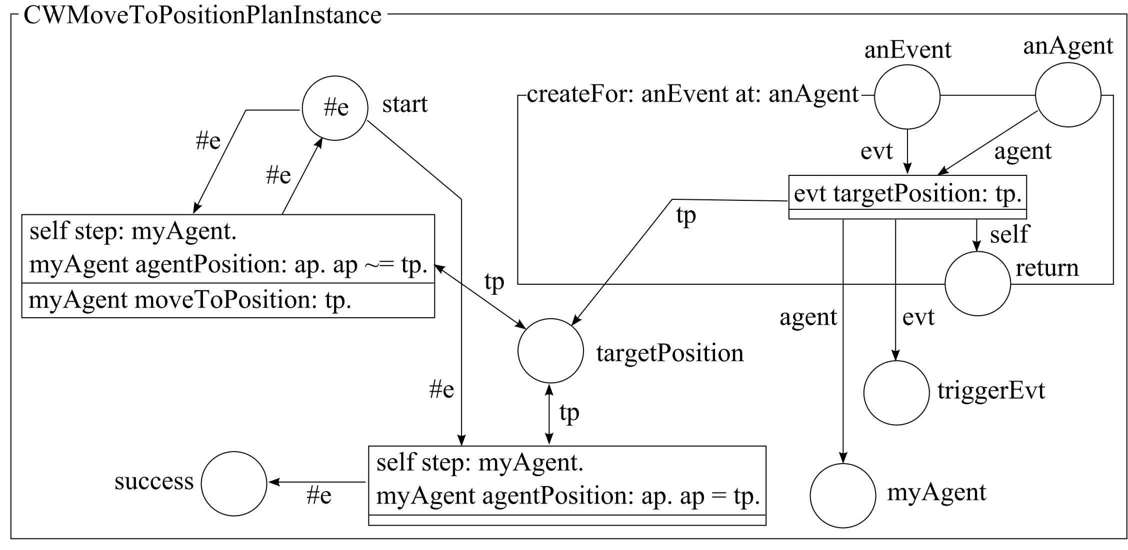
```
n: [dig]+ | id
c: literal | id | list
list: "(" [c ["," c]* ["|" [id | list] ]] ")"

temps: "|" [id]* "|"
unit: id | literal | "(" expr ")"
unaryexpr: unit [ id ]+
primary: unit | unaryexpr
exprs: [expr "."]* [expr]
expr: [id ":="]* expr2
expr2: primary | msgexpr [ ";" cascade ]*
expr3: primary | msgexpr
msgexpr: unaryexpr | binexpr | keyexpr
cascade: id | binmsg | keymsg
binexpr: primary binmsg [ binmsg ]*
binmsg: binsel primary
binsel: selchar[selchar]
keyexpr: keyexpr2 keymsg
keyexpr2: binexpr | primary
keymsg: [keysel expr2]+
keysel: id":"
literal: number | string | charconst | symconst | arrayconst
arrayconst: "#" array
array: "(" [number | string | symbol | array | charconst]* ")"
number: [-][[dig]+ "r"] [hexDig]+ ["." [hexDig]+] ["e"["-"][dig]+].
string: "’"[char]*"’"
charconst: "$"char
symconst: "#"symbol
symbol: id | binsel | keysel[keysel]* | string
id: letter[letter|dig]*
selchar: "+" | "-" | "*" | "/" | "~" | "|" | "," | "<" | ">" | selchar2
selchar2: "=" | "&" | "\" | "@" | "%" | "?" | "!"
hexDig: "0".."9" | "A".."F"
dig: "0".."9"
letter: "A".."Z" | "a".."z"
char: letter | dig | selchar | "[" | "]" | "{"|"}" | "(" | ")" | char2
char2:" "|"^"|";"|"$"|"#"|":"|"."| "-"|"‘"
```

Příloha B

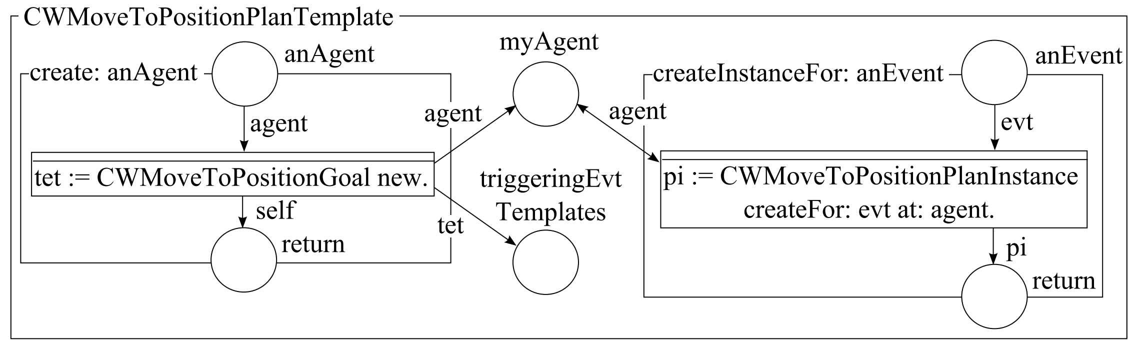
Model Cleaner world

Tato příloha ukazuje kompletní aplikačně specifickou část modelu *Cleaner world*, použitou v páté kapitole práce při popisu práce s bází představ a spouštění plánů. V textu byly ukázány jen klíčové části modelu, zde jsou doplněny o další souvislosti. V podobě prezentované níže není model napojen na reálné prostředí – předpokládá se, že z vnějšku jsou objektu agenta zasílány události představující objevení nového odpadu. Veškeré operace agent provádí jen nad svou bází představ. Model sestává z konkretizované třídy agenta, dvou plánů, jedné aplikačně specifické události a jednoho cíle. Pro běh modelu je používána základní verze interpretu bez použití priorit a meta-plánů. Cílem modelu je demonstrovat základní principy fungování agentů ve frameworku PNagent – práci s bází představ, unifikaci událostí, vytváření plánů, apod.

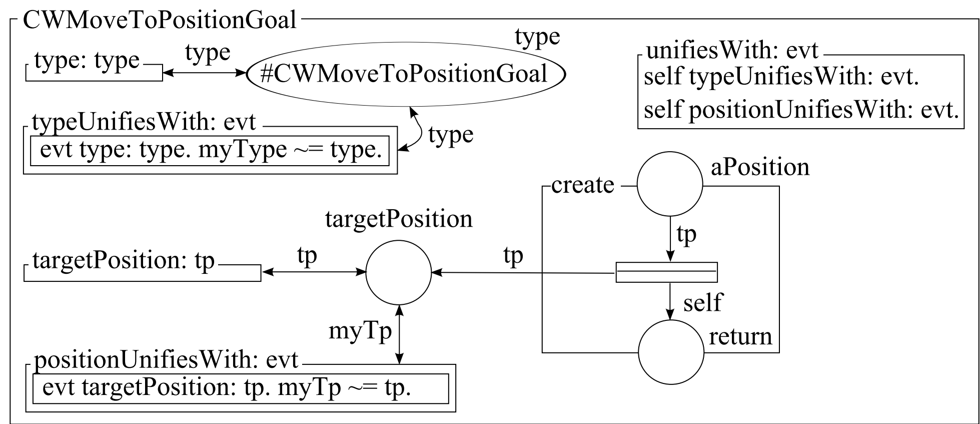
Prvním plánem je *CWMoveToPositionPlan*, jehož instanci ukazuje obrázek B.1 a šablonu obrázek B.2. Cílem tohoto plánu je přesunout agenta na místo, které je specifikováno v cíli *CWMoveToPositionGoal*, který představuje jeho spouštěcí událost. Strukturu třídy CWMoveToPositionGoal ukazuje obrázek B.3.



Obrázek B.1: Instance plánu CWMoveToPositionPlan



Obrázek B.2: Šablona plánu CWMoveToPositionPlan



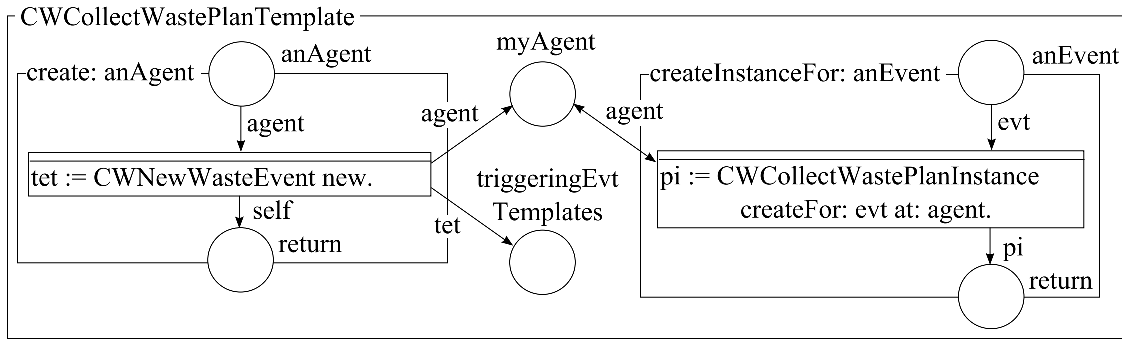
Obrázek B.3: Cíl CWMoveToPositionGoal

Druhým plánem je *CWCollectWastePlan*, který je spuštěn při nalezení nového odpadu, a který má za úkol přemístit agenta na místo odpadu (pomocí spuštění pomocného plánu CWMoveToPositionPlan), zvednutí odpadu, přenesení do popelnice a jeho upuštění. Instanci tohoto plánu ukazuje obrázek B.4, šablonu B.5, událost nalezení nového odpadu pak B.6.

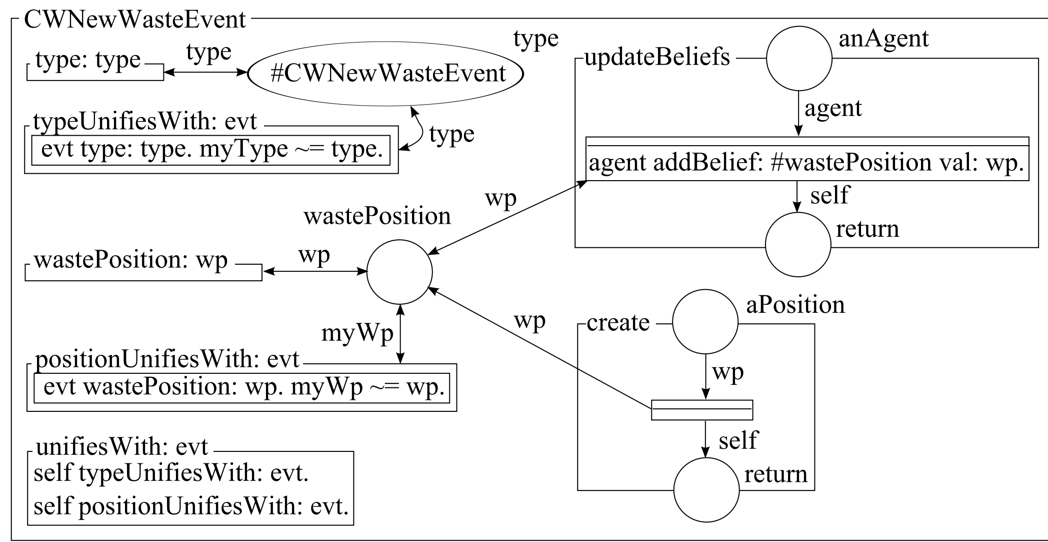
Vlastní třída agenta je ukázána na obrázcích B.7, B.8 a B.9. Obrázky B.7 a B.8 ukazují metody addBelief, resp. removeBelief sloužící k přidávání a odebírání představ z agentovy báze znalostí, které byly příliš rozsáhlé, aby se vešly do jednoho vyobrazení se zbytkem agenta. Ten je obsahem obrázku B.9 a ukazuje implementaci báze představ a agentových schopností.

CWCollectWastePlanInstance
start
#e
anEvent
anAgent
createFor: anEvent at: anAgent
evt
agent
evt wastePosition: wp.
wp
self
return
agent
evt
wastePosition
wp
triggerEvt
wp
myAgent
self step: myAgent.
myAgent agentPosition: ap.
myAgent notBinPosition: ap.
myAgent carryingWaste.
myAgent binPosition: bp.
goal := CWMoveToPositionGoal
create: bp.
self postSubgoal: goal.
self step: myAgent.
myAgent agentPosition: ap.
myAgent binPosition: ap.
myAgent carryingWaste.
myAgent dropWaste.
self step: myAgent.
myAgent notAgentPosition: wp.
myAgent notCarryingWaste.
goal := CWMoveToPositionGoal
create: wp.
self postSubgoal: goal.
self step: myAgent.
myAgent agentPosition: wp.
myAgent notCarryingWaste.
myAgent pickWaste.
self step: myAgent.
myAgent agentPosition: ap.
myAgent binPosition: ap.
myAgent carryingWaste.
myAgent dropWaste.
success
self step: myAgent.
myAgent agentPosition: ap.
myAgent notBinPosition: ap.
myAgent carryingWaste.
myAgent binPosition: bp.
goal := CWMoveToPositionGoal
create: bp.
self postSubgoal: goal.

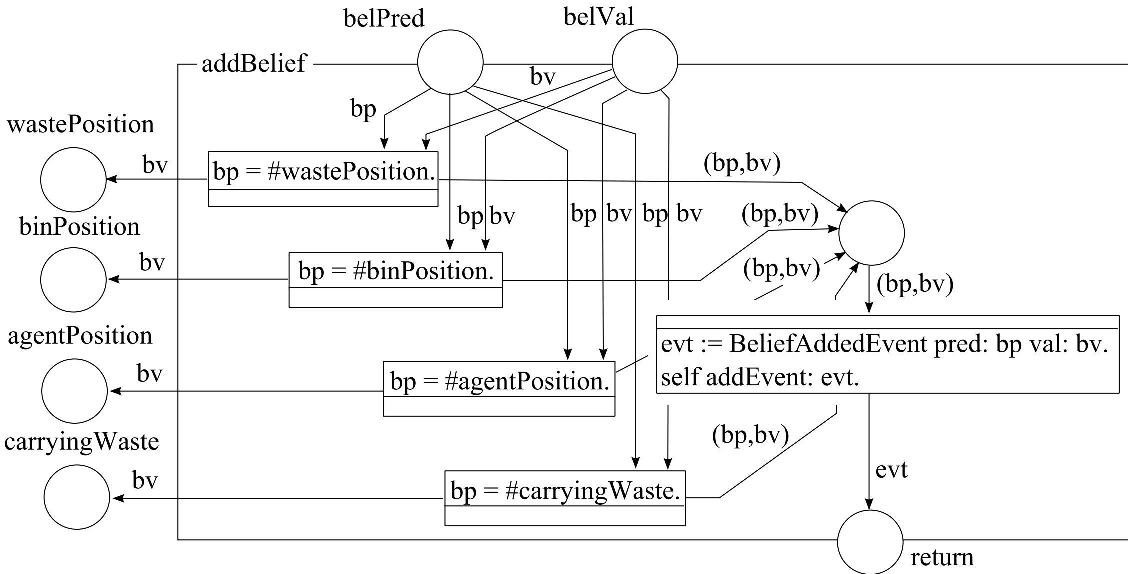
Obrázek B.4: Instance plánu CWCollectWastePlan



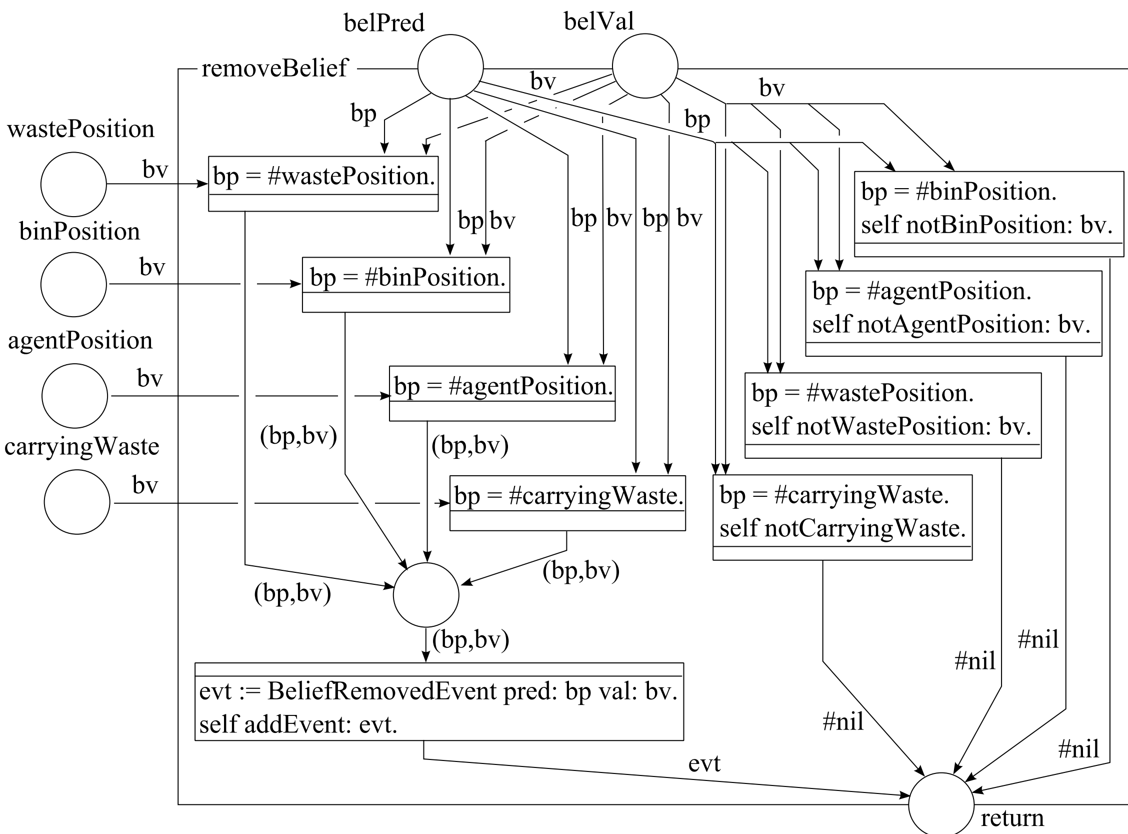
Obrázek B.5: Šablona plánu CWCollectWastePlan



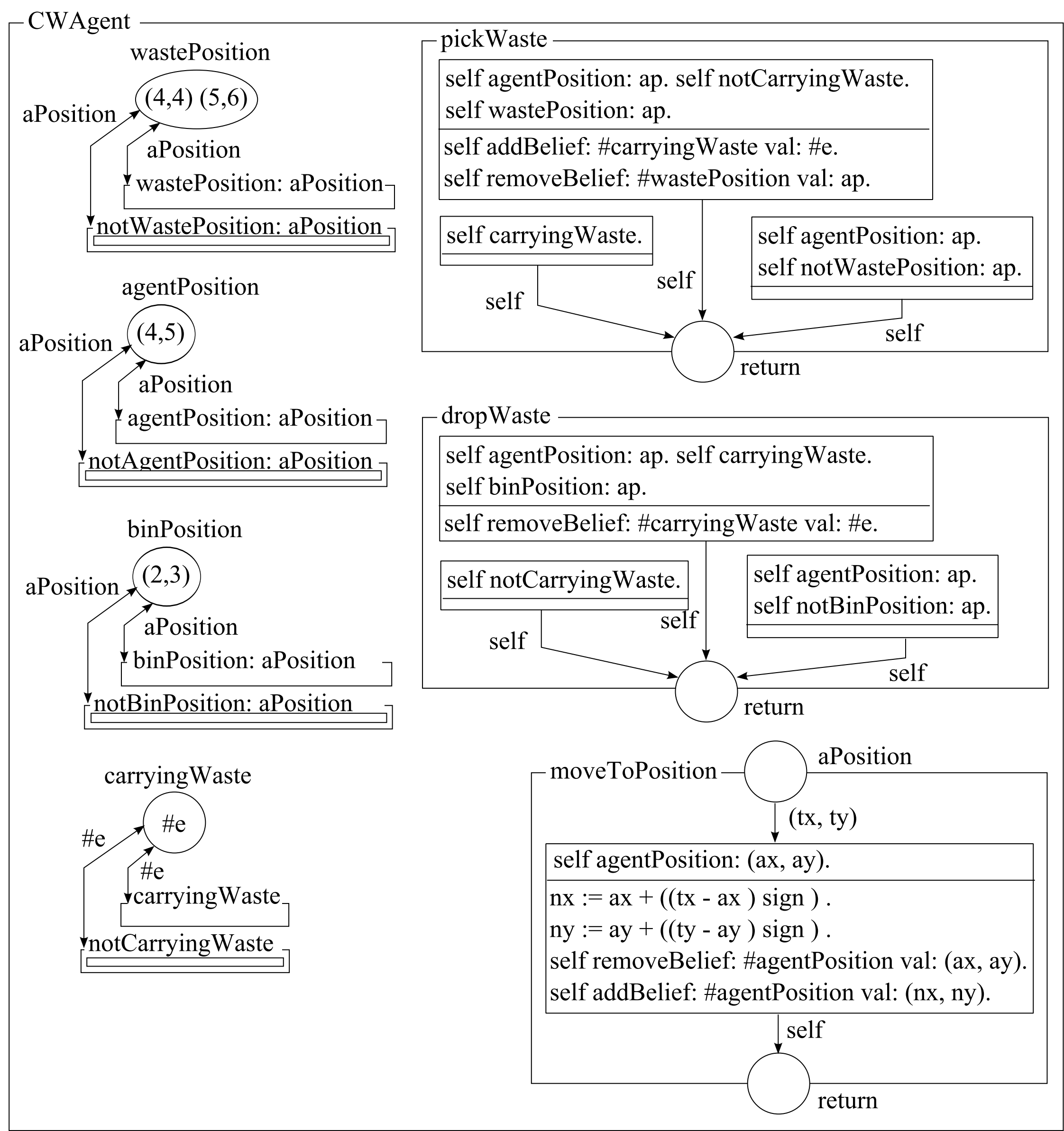
Obrázek B.6: Událost CWNewWasteEvent



Obrázek B.7: Metoda addBelief třídy CWAgent



Obrázek B.8: Metoda removeBelief třídy CWAgent

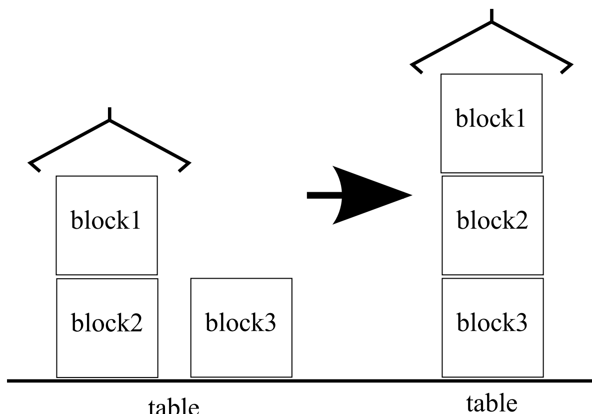


Obrázek B.9: Třída CWAgent, implementující bázi představ a agentovy schopnosti

Příloha C

Model Blocks world

Blocks world představuje jednu z klasických plánovacích úloh. Obsahuje tři kostky o stejné velikosti (block1, block2 a block3) a robotické rameno, které může zvednout jednu z kostek a přemístit ji na jinou kostku nebo na stůl (table). Cílem je přeskládat kostky z počátečního do cílového stavu. Příklad takového počátečního a cílového úlohy ukazuje obrázek C.1.

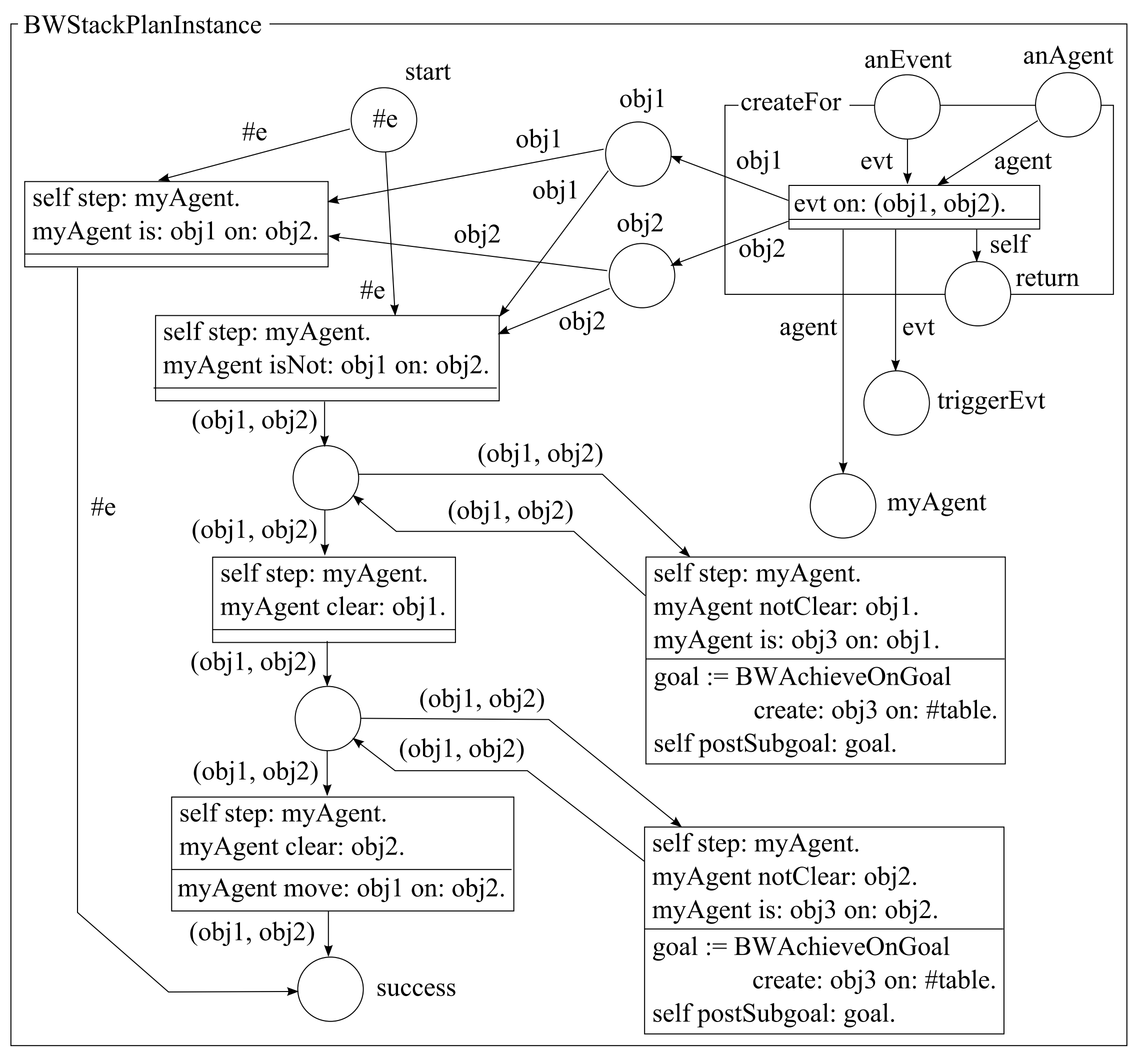


Obrázek C.1: Počáteční a cílový stav úlohy Blocks world

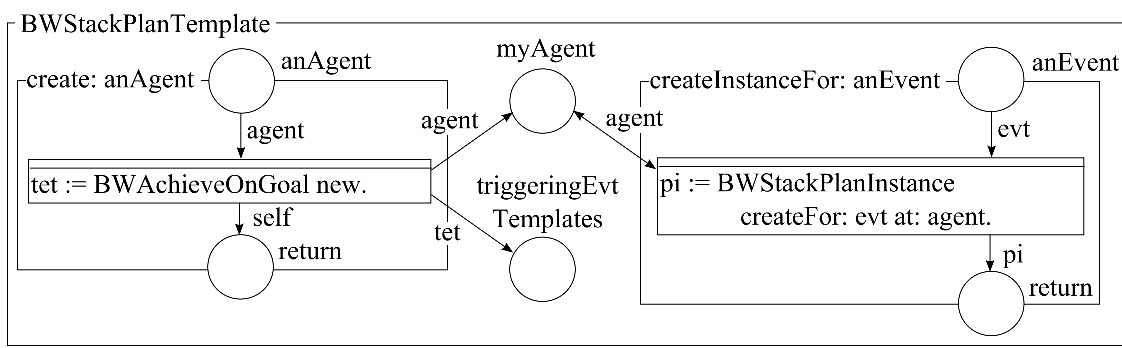
Pro systémy praktického usuzování není takováto úloha typická – agentovo prostředí je statické, takže problém je efektivněji vyřešen klasickými plánovacími algoritmy. Prezentovaný model však ukazuje jednu z možností, jak k danému problému přistoupit. Navíc umožňuje porovnání s dalšími BDI systémy, ve kterých byl tento model implementován, například JAM, prezentovaný v [Woo02]. Model prezentovaný dále je spuštěn speciální událostí poté vykonává veškeré přesuny pouze jako manipulaci se svojí bází představ – není tedy stejně jako v modelu Cleaner world napojen na okolní prostředí.

Hlavním plánem, který provádí vlastní přesuny kostek, je *BWStackPlan*. Jeho instanci ukazuje obrázek C.2, šablonu pak C.3. Plán je spouštěn cílem *BWAchieveOnGoal* (obrázek C.4), který specifikuje, která kostka se má přesunout kam.

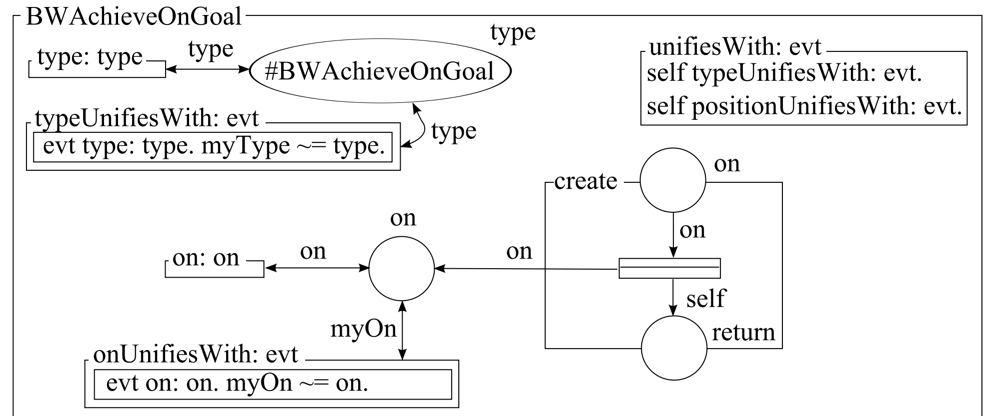
Druhým plánem, který specifikuje jednotlivé makro-kroky přesunu (tedy v postatě jak má vypadat výsledek a které kostky umístit nejdříve), je plán *BWTopLevelPlan* (obrázky C.5 a C.6). Tento plán je spouštěn událostí *BWStartEvent* (obrázek C.7), která kromě spuštění tohoto plánu provede inicializaci agentovy báze představ. Báze představ a agentova



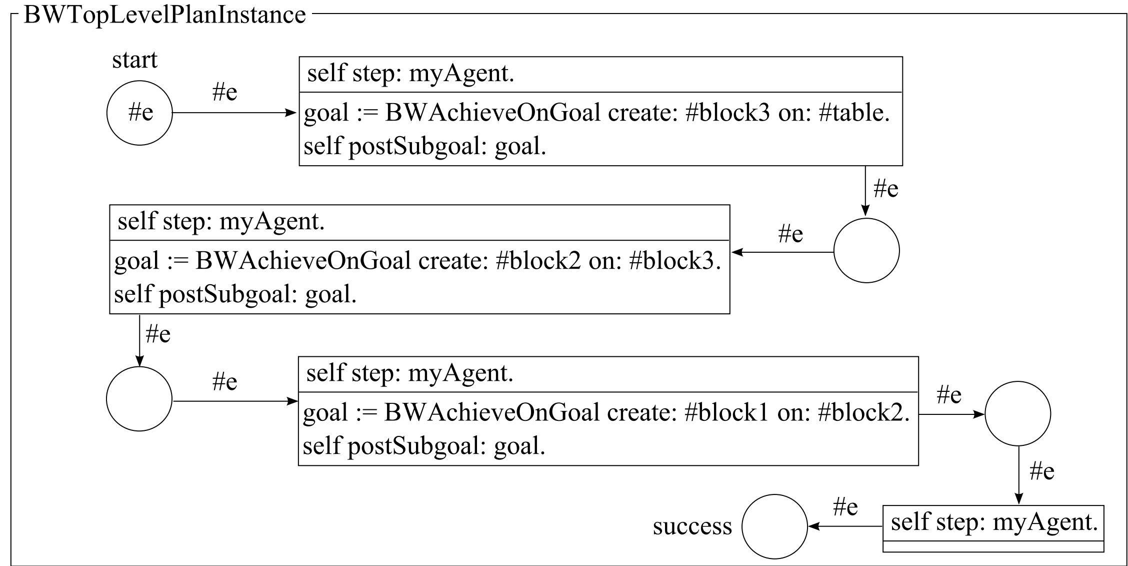
Obrázek C.2: Instance plánu BWStackPlan



Obrázek C.3: Šablona plánu BWStackPlan

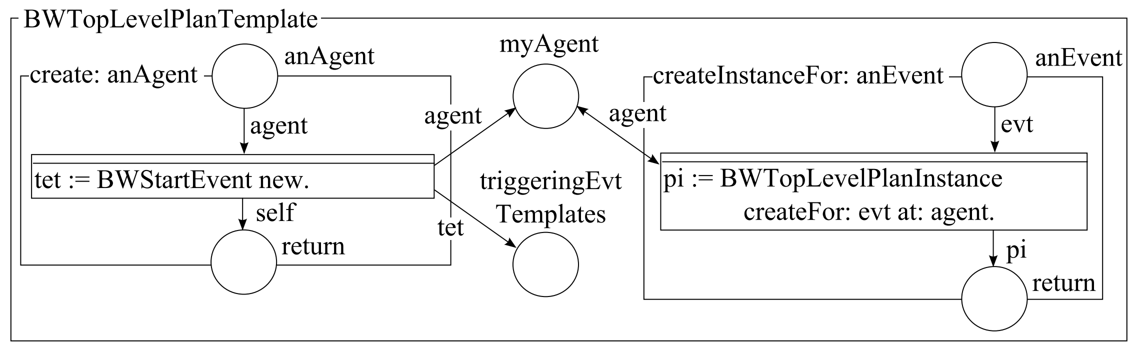


Obrázek C.4: Cíl BWAchieveOnGoal

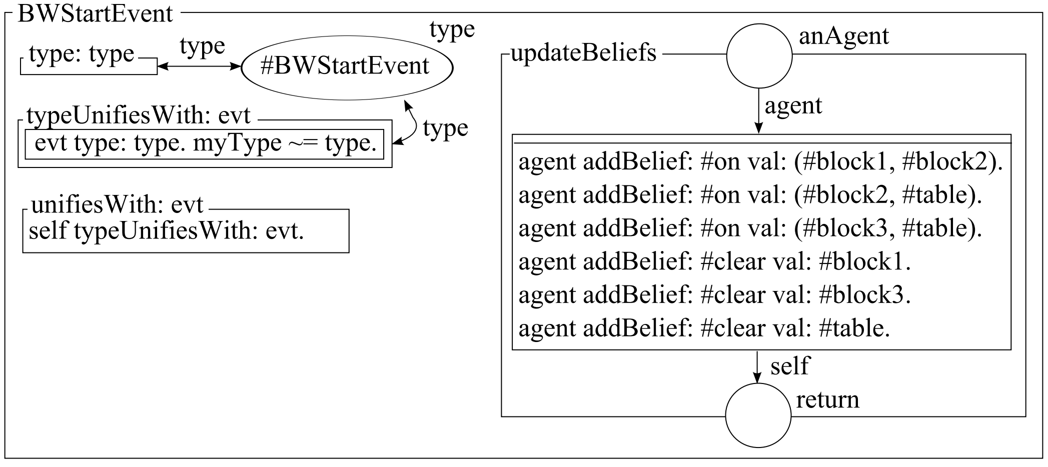


Obrázek C.5: Instance plánu BWTopLevelPlan

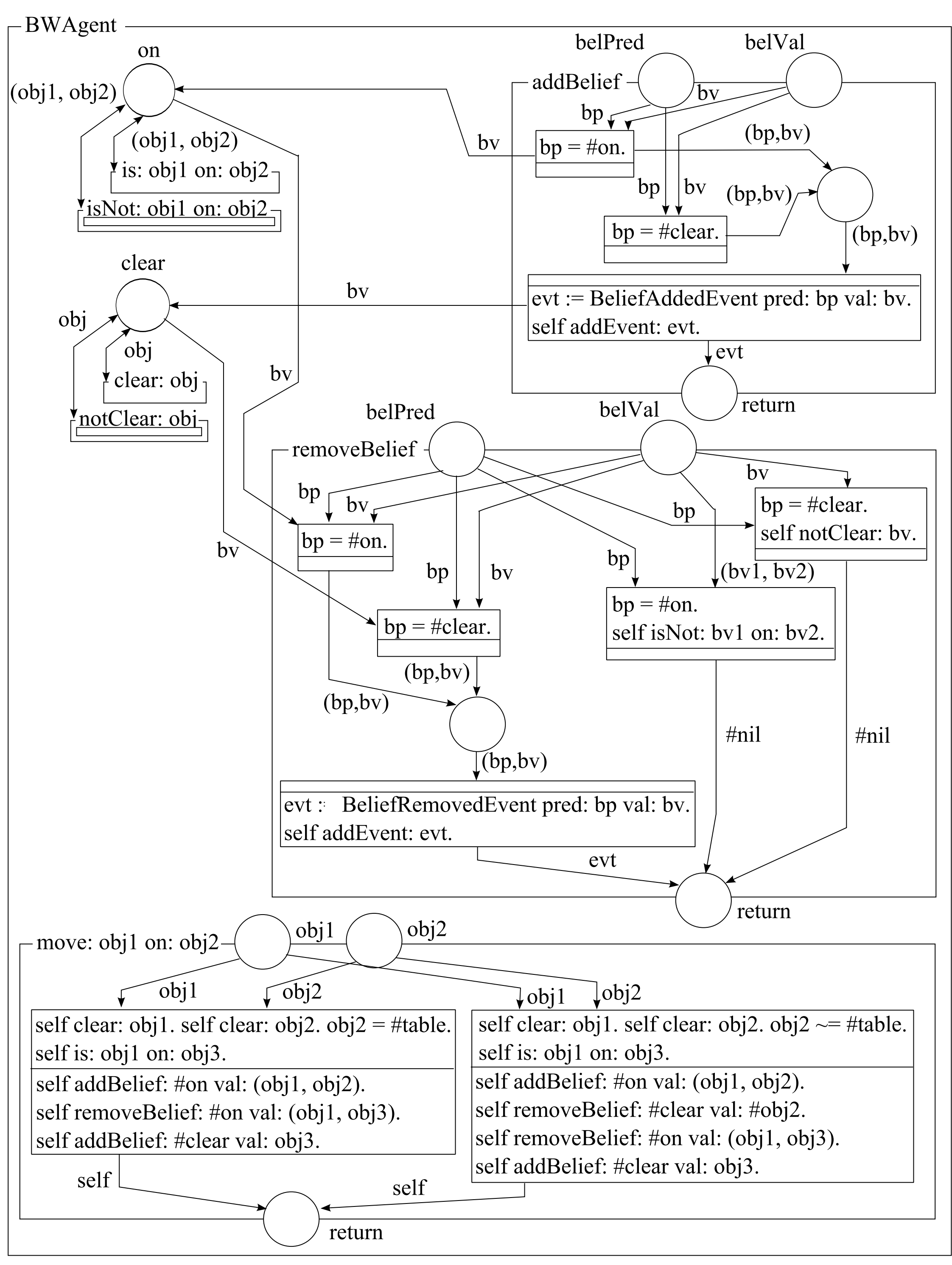
jediná schopnost – přesunovat kostky, jsou implementovány třídou *BWAgent*. Tu ukazuje obrázek C.4.



Obrázek C.6: Šablona plánu BWTopLevelPlan



Obrázek C.7: Událost BWStartEvent



Obrázek C.8: Aplikačně specifická část třídy BWAgent

Příloha D

Návod pro spuštění modelů na přiloženém CD

Framework PNagent byl prototypově implementován v systému PNtalk 2007, který je založen na prostředí Squeak Smalltalk. Vývojové prostředí a způsob distribuce se u Squeaku značně liší od tradičních programovacích jazyků. Distribuce se skládá z interpretu, který je určený pro konkrétní platformu, a „image“, což je zmrazený stav systému objektů, který je platformně nezávislý. Na přiloženém CD se nachází interpret fungující na platformě Windows, interpret pro ostatní platformy je možné stáhnout na stránkách Squeaku[1]. Pro spuštění je připraven dávkový soubor *run.bat*.

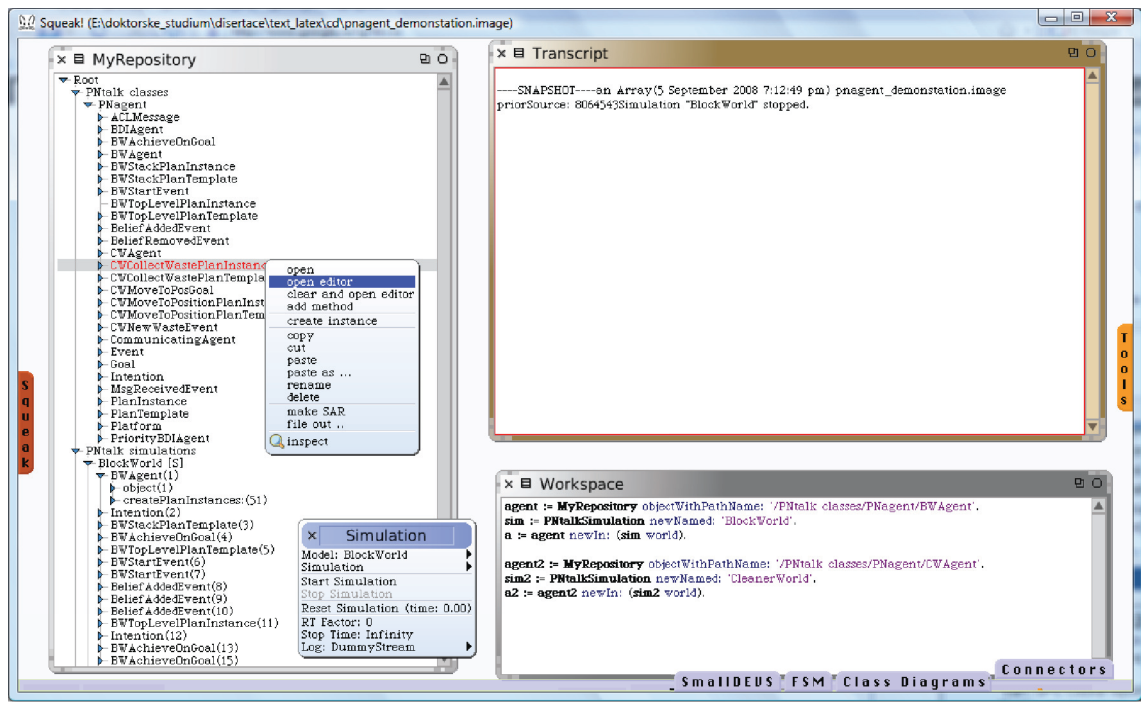
Obrázek D.1 ukazuje snímek obrazovky prostředí PNtalk. Vlevo se nachází tzv. „Repository“, což je místo, kde jsou uloženy v příslušných složkách všechny dostupné třídy frameworku PNagent a vytvořené simulace. Kliknutím pravým tlačítkem myši na název třídy či simulace se vyvolá kontextové menu. To umožňuje u tříd zobrazit jejich zdrojový kód v textové (položka *open*) i grafické (položka *open editor*) podobě, případně přidávat či odebírat metody, atd. U simulací umožňuje menu jejich spuštění, zastavení, smazání, apod. Kliknutím levým tlačítkem myši na modrý trojúhelník vlevo od názvu položky dojde k zobrazení podřazených položek, tedy u třídy jejích metod, u simulace objektů, které se v ní nacházejí.

Jelikož se projekt PNtalk se nachází ve stavu alfa verze, obsahuje celou řadu nedodělků a velmi omezenou podporu pro grafické zobrazování modelů a simulací – ve skutečnosti je možné pouze zobrazit objektovou síť či síť metody, která byla předtím zadána textově. PNtalk se také nestará o správné rozmístění grafických prvků na obrazovce – to je nutno provést ručně.

Vzhledem k relativně neintuitivnímu ovládání systému PNtalk jsem do ukázkového image zahrnul jen základní třídy frameworku PNagent a dva jednoduché modely, popsané v přílohách B a C. Model ovládání robotů, který vyžaduje také instalaci simulátoru Player/Stage pod Linuxem, jeho konfiguraci a celou řadu dalších nastavení jsem vynechal. Modely odpovídají těmto přílohám poměrně přesně, pouze obcházejí jisté nedodělky systému PNtalk v oblasti překladu (překladač dosud nepodporuje konstruktory a některé syntaktické obraty se seznamy).

Pro oba modely jsou předpřipraveny simulace, které lze spustit pomocí položky „Start Simulation“ z kontextového menu. Po spuštění simulace se vypisují agentovy akce do oblasti „Transcriptu“, což je v prostředí Squeaku obdoba konzole (na obrázku D.1 vpravo nahoře).

[1] http://www.squeak.org/



Obrázek D.1: Snímek obrazovky prostředí PNtalk s připravenými modely

Další simulace je možné vytvářet označením bloku kódu v oblasti nazvané „Workspace“ (na obrázku D.1 vpravo dole), vyvoláním kontextového menu pravým tlačítkem myši a volbou *Do It*.

U modelu Cleaner world je při inicializaci agent umístěn na pozici (1,1) a popelnice na pozici (2,1). Po inicializaci je vytvořen pouze jeden odpad, a to na pozici (5,5). V modelu Block world odpovídá počáteční a cílový stav situaci na obrázku C.1, prezentovaného v příloze C.