VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

INTERPRETACE AGENTNÍHO SYSTÉMU ŘÍZENÉHO ZÁMĚREM V JAZYCE PROLOG

INTENTION DRIVEN AGENT IN PROLOG

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. LADISLAV NĚMEC**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Ing. FRANTIŠEK ZBOŘIL, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Němec Ladislav, Bc.**
Program: Informační technologie Obor: Inteligentní systémy
Název: **Interpretace agentního systému řízeného záměrem v jazyce PROLOG**
Intention Driven Agent in PROLOG
Kategorie: Umělá inteligence
Zadání:

1. Nastudujte aktuální agentní systémy řízené záměrem, konkrétně s jazykem AgentSpeak(L) a systémy založenými na tomto jazyku. Dále se seznamte se systémem FRAg, který rozšiřuje flexibilitu rozhodování BDI agenta při interpretace AgentSpeak(L).
2. Implementujte v jazyce PROLOG interpret BDI agenta s rozšířeným prostředím podle specifikace FRAg.
3. Vytvořte alespoň jednoduché vývojové prostředí pro Vámi vytvořený interpet a jeho dokumentaci.
4. Navrhněte úlohy, u kterých je možné najít postupy, které vedou ke splnění více sílů současně. Zkoumejte na nich chování agenta řízeného klasickým interpretem jazyka AgentSpeak(L) v porovnání s chováním agenta ve Vašem interpretu.
5. Zhodnoťte chování agenta a diskutujte výhody a nevýhody obou řešení.

Literatura:

- Wooldridge, M.: An Introduction to MultiAgent Systems, 2nd Edition, Willey, 2009
- Rao, A., S.AgentSpeak(L): BDI agents speak out in a logical computable language, LNCS 1038, 1996
- Specifikace FRAg (bude poskytnuta vedoucím práce)

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Zbořil František, doc. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 4. června 2020

Abstrakt

Tato páce se zabývá realizací interpretu agentního systému řízeného záměrem implementovaného v jazyce PROLOG. Jako vzor byl využit interpret Jason implementovaný v jazyce Java, který interpretuje jazyk AgentSpeak(L). Byl vytvořen interpret a program na zpracování agentních systémů v jazyce AgentSpeak(L). Tento interpret dokáže pracovat s více agenty, dokáže realizovat systém s prostředím a při interpretaci využívat systému FRAg. Byli navrženy příklady agentních systémů v jazyce AgentSpeak(L) pro popis funkčnosti interpreta a následně byli popsány výhody a nevýhody systému FRAg.) jazyce.

Abstract

This lever deals with the realization of the iterpreter of an Driven Agent by the PROLOG implementation. The model was used by Jason implemented in Java that interprets the language of AgentSpeak(L). An interpreter and a program for processing agent systems in the language AgentSpeak (L) were created. This interpreter can work with multiple agents, can implement a system with an environment and use the FRAg system for interpretation. Examples of agent systems in AgentSpeak (L) were proposed to describe the functionality of the interpreter, and subsequently the advantages and disadvantages of the FRAg system were described.

Klíčová slova

BDI agent, Jason, PROLOG, AgentSpeak(L), interpret, FRAg, agentní systém

Keywords

BDI agent, Jason, PROLOG, AgentSpeak(L), interpreter, FRAg, agent system

Citace

NĚMEC, Ladislav. *Interpretace agentního systému řízeného záměrem v jazyce PROLOG.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. František Zbořil, Ph.D.

Interpretace agentního systému řízeného záměrem v jazyce PROLOG

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Ing. Zbořila Františka, Ph.D. Další informace mi poskytli... Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Ladislav Němec
5. června 2020

Poděkování

Tímto bych rád poděkoval Doc. Ing. Františkovi Zbořilovi, Ph.D. za jeho pomoc při konzultacích a za jeho nápady při tvorbě této diplomové práce.

Obsah

Kapitola 1

Úvod

V současnosti lze již tvrdit, že agentní a multiagentní systémy dosáhly stupně dospělosti, který je spojený s jejich nasazením v řadě oborů. Lze se s nimi setkat např. v astronautice při ovládání vesmírných lodí, zpracování zdravotních záznamů, plánování vojenských misí, správě síťových zdrojů, plánování řešení katastrof nebo třeba v aukcích[15].

Cílem této práce bylo navrhnout interpret agentního systému řízeného záměrem v jazyce PROLOG. Tento interpret je schopen realizovat program inspirovaný jazykem AgentSpeak(L), zprostředkovat agentům interakci s prostředím. Dále podporuje systémem FRAg, který rozšiřuje flexibilitu rozhodování BDI agenta při interpretaci AgentSpeak(L).

Práce je členěna do několika kapitol. Kapitola 2 krátce pojednává o interpretaci jazyka. V kapitole 3 nalezneme informace o agentních systémech řízených záměrem. Seznámíme se s BDI architekturou a předvedeme si jak může vypadat řídící cyklus agenta. Jazyku AgentSpeak(L) se věnuje kapitola 4. Zabývá se především sémantickou a syntaktickou stránkou tohoto jazyka. Také je zde představen model interpretace prostřednictvím interpretu Jason. Kapitola 5 pojednává o programovacím jazyku Prolog. Věnuje se jeho syntaktickým a sémantickým vlastnostem. Také srovnává jazyk Prolog a AgentSpeak(L). Získáme zde informace o interpretu tohoto jazyka SWI-Prolog. V kapitole 6 se seznámíme se systémem FRAg. Dozvíme se jeho výhody a jakým způsobem rozšiřují klasickou interpretaci agentních systémů. Kapitola 7 hovoří o mém přínosu v rámci této práce. Představuje návrh modelu mého interpreta a detailně se věnuje jeho implementaci. Posoudíme chování interpretu v kapitole 8, která je věnována testování. V kapitole 9 se zhodnocují výsledky práce, hodnotí se přínos systému FRAg v interpretaci agentních systémů a diskutuje se o navrženém interpretu a jeho vlastnostech.

Kapitola 2

Interpretace jazyka

V táto kapitole si vysvětlíme rozdíl mezi interpretem a kompilátorem, popíšeme si jejich výhody a nevýhody.

Interpret je v informatice speciální počítačový program, který přímo vykonává program ve formě zdrojového kódu napsaného v konkrétním programovacím jazyku, jinými slovy jej interpretuje[9]. Není nutné tedy program převádět do strojového kódu cílového procesoru, což je typické pro překladače. Program je pak snadno přenositelný mezi různými počítačovými platformami.

Podle toho zda je daný jazyk interpretovatelný se dělí na interpretované a kompilované. Teoreticky je možné vytvořit interpret i kompilátor pro jakýkoliv programovací jazyk, avšak interpretované jsou obvykle vyšší programovací jazyky. Některé jazyky vykonávají program kombinací těchto dvou přístupů. Mezi interpretovací jazyky se řadí i AgentSpeka(L).

Výhody interpretace jazyka:

- Nezávislost na počítačové platformě.
- Dynamické typování.
- Reflexe a reflektivní použití evaluátoru.
- Dynamické řízení oblasti platnosti identifikátorů.
- Se zdrojovým kódem lze manipulovat přímo, lze jej číst a kopírovat což může být více svobodné než u kompilovaných jazyků.

Nevýhody interpretace jazyka:

- Pomalejší běh programu v porovnání s rychlostí běhu strojového kódu nativního pro příslušný procesor je hlavní nevýhodou interpretace. Technikou just-in-time ovšem můžeme tento problém minimalizovat. Technika převádí části programu do strojového kódu před jejich prováděním.
- Při kompilaci se na rozdíl od interpretace provádí statické typové kontroly, což může zamezit chybám v programu a nevyžádanému chování. Ovšem typová kontrola lze u interpretovatelných jazyků provést dodatečnými nástroji, k tomu určenými.
- Interpretované jazyky mohou být více náchylné k útoku vložením části cizího kódu.

- Díky snadnému čtení interpretovaných jazyků se hůře bojuje proti záchovní duševního vlastnictví.

V našem případě bude interpret vykonávat kód, který je převeden z jazyka AgentSpeak(L). Jedná se vlastně o strukturu v jazyce Prolog, která se jazyku AgentSpeak(L) podobá. Jejich transformace je popsaná v kapitole 7.1. Převod je prováděn automaticky speciálním programem, který také vznikl v rámci této práce.

2.1 Syntaktická analýza a další akce interpretu

Jednou z důležitých částí zpracování programu, při jejich kompilaci a nebo interpretaci, je syntaktická analýza. Analýzou programu napsaném v určitém programovacím jazyku vzniknou datové struktury, reprezentující tento program, které interpret dokáže zpracovat a vykonat. Na rozdíl od toho kompilátor z těchto struktur generuje program v cílovém jazyce, který lze po dalších úpravách interpretovat hardwarovým nebo virtuálním procesorem.

Efektivní a rychlý syntaktický analyzátor lze vytvořit na základě deterministických bezkontextových gramatik, kterými bývají jednotlivé programovací jazyky specifikovány. Většinou bývá právě na základě deterministické bezkontextové gramatiky generován syntaktický analyzátor automaticky tzv. parser generátorem.

Proces analýzy:

- První fáze je lexikální analýza. Tento proces transformuje vstupní program v textové podobě, který lze charakterizovat jako řetězec znaků, na jednotlivé tokeny, zpravidla definované gramatikou regulárních výrazů. Jednotlivé tokeny nemusí být odděleny bílými znaky a proto tzv. Scanner obsahuje pravidla označují začátek nového tokenu. Tento proces předchází samotné syntaktické analýze.

- Další fází je samotná syntaktická analýza. Jako první se kontroluje, zda jednotlivé tokeny tvoří povolené výrazy. Pokut tomu tak není proces syntaktické analýzy skončí nezdarem. Na této úrovni je jazyk obvykle popsán bezkontextovou gramatikou, která rekurzivně definuje komponenty, z nichž lze složit výraz, a pořadí, ve kterém se musí objevit. Ovšem ne všechny pravidla definující programovací jazyky mohou být vyjádřena bezkontextovou gramatikou, například deklarace identifikátorů či typová kontrola se takto popsat nedá. K tomuto účelu můžou sloužit atributové gramatiky, které slouží k formálnímu popisu těchto pravidel. Z této gramatiky se vytvoří tabulka symbolů, která následně slouží k dodržování požadovaných pravidel.

- Poslední fází analýzy je sémantická analýza kdy se jednotlivé akce programu provádějí.

Syntaktický analyzátor má za úkol rozhodnout, zda je možné z počátečních symbolů gramatiky vygenerovat program předložený na vstupu interpreta. To lze provést dvěma způsoby.

Dva přístupy syntaktické analýzy:

- **Syntaktická analýza shora dolů** - Můžeme ji popsat jako hledání levé derivace vstupního řetězce. Lze ji realizovat jako metodu, kdy se snažíme v určitém bodě překladu aplikovat postupně jednotlivá pravidla gramatiky. Tento postup se nazývá „analýza s návraty“. Je pro syntaktickou analýzu programovacích jazyků nevhodný,

pro svou značnou neefektivnost, ale pozitivní je, že většina běžných konstrukcí v programovacích jazycích umožňuje přímočarou analýzu bez návratu. Další možností je použití deterministické analýzy, ve které při výběru pravidla využíváme více informací. Například se můžeme dívat dále do vstupní posloupnosti termů a řídit se tím. U metody analýzy shora dolů se ve většině případů používá deterministická analýza.

- **Syntaktická analýza zdola nahoru** - Při tomto přístupu se derivační strom sestavuje odspodu. Nejdříve se identifikují jednotlivé symboly, z nichž je složen vstupní text, a z nich se následně vytvářejí složitější struktury.

Pro náš interpret použijeme syntaktickou analýzu shora dolů. Provedeme transformaci jazyka AgentSpeak(L) na interní reprezentaci v jazyku Prolog.

Kapitola 3

Agentní systémy řízené záměrem

V této kapitole se budeme věnovat agentním systémům řízeným záměrem. Vysvětlíme si pojem záměr. Dále se budeme věnovat BDI architektuře a popíšeme si základní cyklus agenta v rámci této architektury. Tato kapitola čerpá převážně z [17].

Systém řízený záměrem, někdy také nazývaný intenční systém, je přístup, který se snaží nalézt způsob realizace racionálního agenta. Myšlenka vychází z filosofie a psychologie. Jde o snahu realizovat agenta, který by se rozhodoval na základě svých postojů ohledně svých přání, závazků, představ a podobně. Souhrnně jde tyto postoje nazvat mentální stavy. Hlavním mentálním stavem je právě záměr. Záměr je chápán jako odhodlání dosáhnout zvoleného cíle. Můžeme ho lépe vymezit pomocí několika bodů, které formuloval filosof Bratman[8]:

- Pro agenta je záměr zadáním problému a agent potřebuje stanovit cestu, jak jej dosáhnout.
- Záměr je mezí přípustnosti pro přijímání dalších záměrů.
- Agent si uchovává záznam o úspěšnosti svých pokusů o dosažení záměrů.
- Agent věří, že je možné splnit záměr.
- Agent nevěří, že se nikdy nemůže přiblížit dosažení záměru.
- Za určitých podmínek agent věří, že se přiblíží k dosažení záměru.
- Agent nemusí mít v úmyslu všechny vedlejší účinky, které nastanou při dosahování záměru.

U systémů řízených záměrem má praktické rozhodování dvě fáze:

1. **Zvažování** - výběr cíle, kterého chce agent dosáhnout a zavázání se, že se pokusí cíle dosáhnout.
2. **Plánování** - sestavování plánu, jakým způsobem dosáhnout daného záměru.

Pokud má agent plán k dosažení záměru a zvolil si jej jako záměr s nejvyšší prioritou, pak tento záměr následuje. Naopak záměr nenásleduje, pokud se rozhodl dosáhnout jiného dříve vytvořeného záměru, nebo nezná způsob, jakým by jej momentálně dosáhl. Agent však věří, že může nastat situace, kdy bude záměr následován.

3.1 BDI architektura

BDI architektura je přístup k tvorbě agentních a multiagentních systémů[27][16]. BDI agent je zvláštní druh deliberativního či racionálního agenta, jehož jednání vychází z třech konkrétních mentálních stavů, jejich první písmena anglické podoby tvoří zkratky BDI [17]:

- **Beliefs** (**Představy**) - Jsou to informace, které agent má. Jedná se o představy o stavu světa, ve kterém se agent nachází. Tyto představy ovšem nemusí být nutně pravdivé a mohou být proměnlivé.
- **Desires** (**Přání**, touhy) - Určují, čeho by agent chtěl dosáhnout. Stavy světa, o kterých by agent chtěl, aby nastaly. Mohou být jak krátkodobé, tak i dlouhodobé. Dlouhodobým přáním se často říká cíle (Goals). Nemusí být dosažitelné, aby agent všechna přání vyplnil, některé se dokonce mohou vzájemně vylučovat.
- **Intentions** (**Záměry**, instance) - Představují, co se agent může rozhodnou udělat. Jde o záměry konání, které mohou vést ke splnění přání a cílů. Záměry jsou přání, ke kterým má agent nějaký druh závazku. Intence mohou být i společné pro více agentů usilujících o společný cíl.

Důležitou součástí BDI agenta je plánovač. Ten na základě přání, záměrů a představ sestavuje plán, jak cílů dosáhnout. Plány ale mohou být i dopředu implementovány pro konkrétní záměry a při běhu agenta jsou pouze vybírány z této sady plánů.

3.2 Řídící cyklus agenta

Jednou z důležitých součástí BID architektury je řídící cyklus agenta.V tomto cyklu mění agent svoje cíle na základě informací získaných z prostředí nebo od jiných agentů. Pseudokód cyklu by mohl vypadat například tak, jako v algoritmu 1.

V uvedeném algoritmu představují proměnné B, D, I představy, přání a záměry agenta. Funkce f slouží k získání informací z prostředí pomocí senzorů. Funkce $options()$ slouží k aktualizaci přání na základě představ a záměrů. Pro výběr konkrétních přání, z kterých se stanou záměry, slouží funkce $filter()$. Sestavení plánu pro dosažení svých nově aktualizovaných záměrů slouží funkce $plan()$.

Jeden cyklus končí tehdy, je-li seznam kroků plánu prázdný (funkce $empty()$) a pokud je vyhodnoceno, že bylo docíleno záměru (funkce $succeeded()$) nebo záměru vůbec docílit nelze (funkce $impossible()$).

Výběr prvního kroku plánu realizuje funkce $first\_element\_of()$. Pro vykonání jednoho kroku plánu slouží funkce $execute()$. Zbytek plánu je přiřazen do proměnné π (funkce $tail\_of()$). Rozhodnutí, zda mají být přehodnoceny záměry zajišťuje funkce $reconsider()$. Další funkce algoritmu $sound()$ slouží k ověření, zda aktuální plán vyhovuje záměrům a představám agenta. Pokud ne, agent vytvoří nový plán činnosti agenta.

```
B ← B_0 /* Počáteční představy agenta. */
I ← I_0 /* Počáteční záměry agenta. */
while true do
    senzory ziskej dalsi informaci p o prostredi
    B ← f(B, p)
    D ← option(B, I)
    B ← filter(B, D, I)
    π ← plan(B, I, Ac) /*Ac je množina akcí agenta. */
    while not(empty(π)) or succeeded(I, B) or impossible(I, B) do
        α ← first_element_of(π)
        execute(α)
        π ← tail_of(π)
        ziskej dalsi informaci p o prostredi
        B ← f(B, p)
        if reconsider(I, B) then
            D ← options(B, I)
            I ← filter(B, D, I)
        end
        if sound(π, I, B) then
            π ← plan(B, I, Ac)
        end
    end
end
```

Algoritmus 1: Řídící cyklus agenta.

Kapitola 4

AgentSpeak(L)

Tato kapitola popisuje jazyk AgentSpeak(L), který je založený na logickém programování a BDI architektuře pro autonomní agenty. V roce 1996 jej vyvinul Anand Rao a následně i další autoři přispěli k jeho rozvoji [2, 6]. Jeho účelem bylo ukázat konkrétní realizaci, která by vycházela z dřívějších formálních předpokladů. Jazyk patří k jednomu z nejpopulárnějších agentově orientovaných jazyků díky vývoji platformy Jason [21]. Primárně nás bude zajímat syntaxe a sémantika tohoto jazyka. Tato kapitola čerpá z těchto zdrojů[3, 5].

Syntaxe jazyka vychází z níže uvedené BNF gramatiky (viz 4.1), kterou Jason přijal. V gramatice je <ATOM> identifikátorem začínající malým písmenem nebo "_", <NUMBER> je libovolné celé číslo nebo číslo s plovoucí desetinnou čárkou a < STRING> je libovolný řetězec uzavřený ve dvojitých uvozovkách.

V následujících podkapitolách budou vysvětleny a popsány jednotlivé části jazyka a jejich sémantický význam.

4.1 Představy

Každý agent si uchovává představy o prostředí, v kterém se vyskytuje. Tyto agentovy informace o prostředí budeme dále nazývat **množinou představ**. Tuto množinu představují jednotlivé predikáty, které určují co agent zná.

Například takto můžeme zapsat, že pozice střelce je na a5:

position(bishop, a, 5).

Jednotlivé představy můžeme jednak inicializovat na začátku běhu agenta a to právě daným predikátem, nebo je může agent získat při běhu systému z prostředí prostřednictvím senzorů. Jednotlivé představy mohou vyjadřovat vlastnost nějakého objektu a nebo i vztah mezi více objekty. Agent ovšem pouze věří, že uvedené představy o prostředí jsou pravdivé, což díky vlivu různých podmínek nemusí být pravda.

4.1.1 Anotace

V platformě Jason existují oproti klasické specifikaci AgentSpeak(L) tzv. anotace. Ty nezvyšují vyjadřovací schopnost jazyka, ale pomáhají přehlednosti při programování a mohou zjednodušit jednotlivé představy. Jednou z anotací, která má konkrétní význam je **source**. Ta vyjadřuje zdroj informace. Může to být prostředí a u multiagentních systémů, také jiný agent. Programátor anotacím může dát i vlastní význam. Anotace se mohou v případě potřeby vnořovat jedna do druhé. Příkladem anotace může být: **thick(Karol)[source(a1)].**

agent	->	(init_bels | init_goals)* plans
nit_bels	->	beliefs rules
beliefs	->	(literal ".")*
rules	->	(literal ":-" log_expr ".")*
init_goals	->	("!" literal ".")*
plans	->	(plan)*
plan	->	["@" atomic_formula] triggering_event
		[":" context] ["<-" body] "."
triggering_event	->	("+" | "-") ["!" | "?"] literal
literal	->	["~"] atomic_formula
context	->	log_expr | "true"
log_expr	->	simple_log_expr
		| "not" log_expr
		| log_expr "&" log_expr
		| log_expr "|" log_expr
		| "(" log_expr ")"
simple_log_expr	->	(literal | rel_expr | <VAR>)
body	->	body_formula (";" body_forumula)* | "true"
body_formula	->	("!" | "?" | "+" | "-" | "-+") literal
		| atomic_formula
		| <VAR>
		| rel_expr
atomic_formula	->	(<ATOM> | <VAR>) ["(" list_of_terms ")"]
		["[" list_of_terms "]"]
list_of_terms	->	term ("," term)*
term	->	literal | list | arithm_expr | <VAR> | <STRING>
list	->	"[" [term ("," term)* ["|" (list | <VAR>)]] "]"
rel_expr	->	rel_term ("<"|"<="|">"|">="|"=="|"\\=="|"=") rel_term
rel_term	->	(literal | arithm_expr)
arithm_expr	->	arithm_term [("+" | "-") arithm_expr]
arithm_term	->	arithm_factor [("*" | "/" | "div" | "mod") arithm_term]
arithm_factor	->	arithm_simple ["**" arithm_factor]
arithm_simple	->	<NUMBER> | <VAR> | "-" arithm_simple | "(" arithm_expr ")"

Tabulka 4.1: BNF gramatika AgentSpeak(L)[3].

Tato představa může znázorňovat víru, že Karol je tlustý a anotace v tomto případě znamená, že informaci nám poskytl agent a1.

4.1.2 Pravidla

Jednotlivé představy mohou být vyjádřeny také pravidly. Funkčnost pravidel si můžeme ukázat na příkladu:

parent(A,B) :- mother(A,B).
parent(A,B) :- father(A,B).

Následující pravidla se vyhodnocují postupně, tak jak jdou za sebou. Pokud by představa *mother(A,B)* znamenala, že A je matkou B a představa *father(A,B)*, že A je otcem B, pak pravidlo pro *parent(A,B)* bude značit, že A je rodičem B. První pravidlo je pravdivé pokud A je matkou B. Pokud by tomu tak nebylo, začne se vyhodnocovat pravidlo druhé, které bude pravdivé právě tehdy, pokud je A otcem B. Pokud ani toto pravidlo neuspěje, vyhodnotí se celý predikát za nepravdivý.

4.2 Cíle

Jednou z velmi důležitých součástí jazyka AgentSpeak(L) jsou cíle. Cíle určují, čeho má agent dosáhnout. Máme dva druhy cílů:

- **Dosahované cíle** - Jedná se o cíle, kterých agent hodlá dosáhnout. V tomto případě bude před literálem znak "!". Příkladem může být:

  !move(cube).

  Cílem je v tomto případě posunutí kostky.

- **Testovací cíle** - Tyto cíle slouží k získání informací z představ daného agenta. Před literál bude tentokrát vložen znak "?". Příklad:

  ?smart(X).

  Tento cíl může sloužit k získání seznamu chytrých osob.

4.3 Plány

Plány v AgentSpeak(L) slouží k dosažení jednotlivých cílů agenta. Konkrétní plán má následující obecnou strukturu:

spouštěcí událost : kontext <- tělo.

Tyto jednotlivé části si popíšeme v následujících podkapitolách.

4.3.1 Spouštěcí událost

Tato sekce plánu slouží k detekci, kdy se má plán spustit. Při běhu agenta může nastat několik druhů událostí, které mohou zapříčinit spuštění některého z plánů. Tyto události se týkají přidání a odebírání představ a cílů. Jejich seznam je uveden v tabulce 4.2.

Odebírání a přidávání představ se provádí při aktualizaci množiny představ. To se může dít jednak při provádění těla plánu, tak i při přijímání podnětů z prostředí.

Události přijímání a odebírání cílů nastává především při realizaci jiných plánů, ale může nastat například i na základě přijetí zprávy od jiného agenta.

Syntaxe	Význam
`+l`	Přidání představy
`-l`	Odebrání představy
`+!g`	Přidání dosahovaného cíle
`-!g`	Odebrání dosahovaného cíle
`+?g`	Přidání testovacího cíle
`-?g`	Odebrání testovacího cíle

Tabulka 4.2: Tabulka typů spouštěcích událostí

4.3.2 Kontext

Kontext slouží jako podmínka pro vykonání plánu. Jde o logický výraz, který je vyhodnocený na základě představ agenta. Logické výrazy, jak je zvykem, mohou obsahovat závorky, logické spojky and (`"&"`) a or (`"|"`), negace a další. AgentSpeak(L) realizovaný platformou Jason obsahuje několik typů negací. Jejich typy naleznete v tabulce 4.3. Tato část plánu je nepovinná.

Syntaxe	Význam
`l`	Agent věří, že literál l je pravdivý
`~l`	Agent věří, že literál l je nepravdivý
`not l`	Agent nevěří, že literál l je pravdivý
`not~ l`	Agent nevěří, že literál l je nepravdivý

Tabulka 4.3: Tabulka typů negací v platformě Jason

4.3.3 Tělo

Tělo plánu obsahuje sekvenci formulí, které se postupně provádí. Jednotlivé formule jsou odděleny znakem ";". Tělo plánu se začne vykonávat jen tehdy, zda-li nastala událost uvedená v sekci spouštěcí události a byl splněn logický výraz v kontextu plánu. Formule můžou být různých typů:

- **Externí akce** - Tyto akce jsou nadefinovány programátorem. Agent jejich prostřednictvím mění prostředí. Literál neobsahuje na začátku žádný speciální znak. Například: *go(left);*.
- **Interní akce** - Na rozdíl od externích akcí, nemění prostředí. Programátor si je opět může nadefinovat a před literálem mají znak ".". Příklady některých interních akcí, které jsou již předdefinovány:
  - **.send** - slouží ke komunikaci mezi agenty.
  - **.random(X)** - unifikuje `X` s náhodným číslem mezi 0 a 1.
  - **.time(HH,MM,SS)** - unifikuje HH, MM a SS s aktuální hodnotou hodin, minut a sekund.
  - **.println** - slouží k výpisu zprávy do konzole a na konec vloží konec řádku.

– **.wait(T,E)** - pozastaví vykonávání záměru na dobu T (v milisekundách) a nebo čeká na splnění události E. Parametry se mohou zadat samostatně, nebo je i kombinovat.

Interních akcí je násobně víc. Pro více informací navštivte dokumentaci Jason[1].

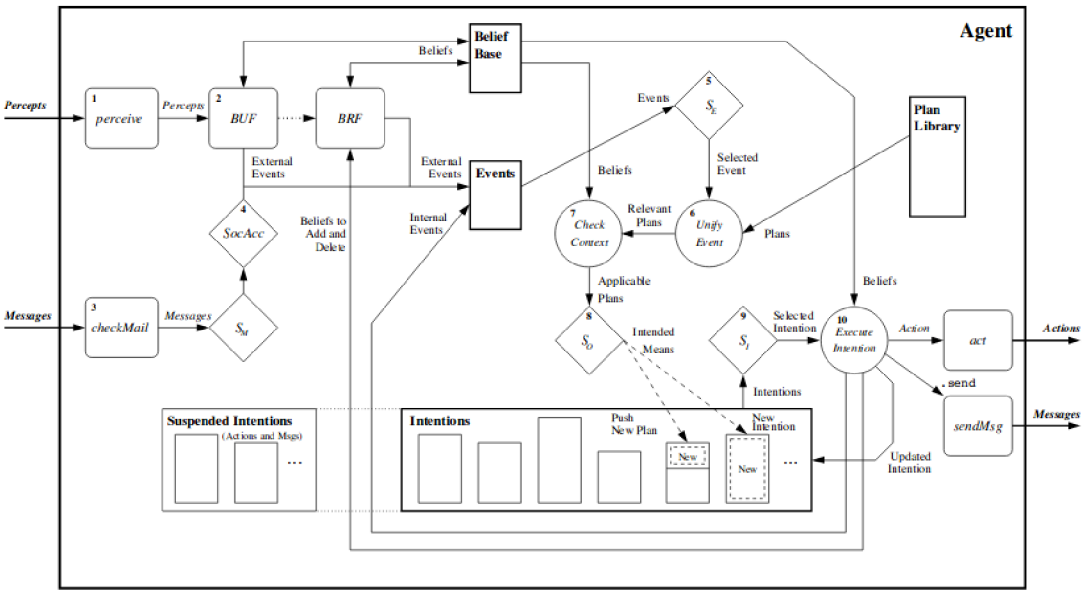
- **Dosahovaný cíl** - Pokud je nadefinovaná operace pro přidání tohoto cíle, je přidán do seznamu cílů a musí být splněn pro dokončení cíle aktuálního. Před literálem je znak "!". Například: *!arrest(burglar);*.
- **Testovací cíl** - Jak již bylo zmíněno výše, slouží k získání informace z představy od agenta. Může buď přidat hodnoty jednotlivým proměnným a nebo být vyhodnocen jako celek za pravdivou, či nikoliv. Může také spustit jiný plán. Před literálem je znak "?". Například: *?woman(X);* vrátí seznam žen v představách agenta.
- **Vlastní záznamy** - Přidávají nové představy do množiny představ, ale mohou je i odebírat. Před literálem je znak "+"pro přidání a "-"pro odebrání představy. Například: *+woman(Anna);* přidá Annu do představ jako ženu a naopak *-woman(Kate);* odebere Kate z představ jako ženu. Tyto operace lze i kombinovat. V tom případě před literálem budou znaky "-+". Například: *-+pos(last,X,Y);* odebere z představ aktuální pozici objektu last a přidá novou pozici X Y.
- **Logické výrazy** - Každá formule v sekvenci musí mít nějakou pravdivostní hodnotu. Pokud je výsledná hodnota *true*, provede se další formule v pořadí a pokud se vyhodnotí celý plán za neúspěšný, začne se provádět jiný. Proto jako formule mohou být i logické výrazy. $X > 5$;.

4.4 Interpretační cyklus agenta

Na obrázku 4.1 je uveden cyklus interpretace pro platformu Jason. Jednotlivé akce jsou na obrázku označeny čísly. Popis jednotlivých akcí:

1. Vnímání prostředí
2. Aktualizace množiny přesvědčení
3. Příjem komunikace od jiných agentů
4. Výběr "sociálně přijatelných"zpráv
5. Výběr události
6. Načtení všech relevantních plánů
7. Určení platných plánů
8. Výběr jednoho platného plánu
9. Výběr záměru pro další provedení
10. Provedení jednoho kroku záměru

[1] http://jason.sourceforge.net/api/jason/asSemantics/DefaultInternalAction.html



Obrázek 4.1: Interpretační cyklus AgentSpeak(L) [2]

Kapitola 5

Prolog

V této kapitole si popíšeme programovací jazyk Prolog a podíváme se i na podobnosti mezi ním a AgentSpeak(L). Tato kapitola vychází převážně z knihy *Programming in Prolog: Using the ISO standard* [10], *ISO Prolog z roku 1993* [12] a článku *Improving the ISO Prolog Standard by Analyzing Compliance Test Results* [23].

5.1 Programovací jazyk Prolog

Vznik programovacího jazyka se pojí s rokem 1972. Autory jsou Alain Colmermauer a Robert Kowalski. Název je odvozen z anglického **Pro**gramming in **Log**ic.

Jazyk Prolog můžeme zařadit mezi tzv. neprocedurální logické programovací jazyky. Tzv. neprocedurální programovací jazyky jsou založeny na myšlence programování aplikací pomocí definic. Definujeme, co se má udělat, ale nedefinujeme způsob jakým toho má být dosaženo, toto je ponecháno na interpretu daného jazyka. Logické programovací jazyky se vyznačují využitím matematické logiky jako prostředku pro programování. Počátky logických programovacích jazyků byli položeny J. McCarthnem [19]. Prolog se snaží vyjádření logických vztahů a faktů abstraktně s potlačením implementační složky. Uplatnění nalézá v mnoha moderních oborech, jako je například umělá inteligence.

5.1.1 Syntaxe

Syntaxe Prologu vychází z predikátové logiky, konkrétně z Hornových klauzulí[14]. Jedná se o speciální druh klauzule, která obsahuje nejvýše jeden pozitivní literál. Lze ji obecně zapsat jako implikaci ve formě:

$u \Leftarrow (p \wedge q \wedge ... \wedge t)$

Prolog využívá vlastní upravené syntaxe. Předešlá klauzule pak v syntaxí prologu vypadá následovně:

$u :- p, q, ..., t.$

Místo implikace píšeme v Prologu ":-", konjunkci nahrazujeme "," a každá klauzule je zakončena tečkou.

5.1.2 Datové typy

V jazyce Prolog rozlišujeme několik datových typů:

- **Atomy** - Můžeme je také nazvat konstanty. Jde o celá čísla (řada implementací pracuje již i s reálnými čísly), kde je posloupnost znaků uzavřených v apostrofech nebo posloupnost malých písmen a podtržítek. Příklady: mother, 'head22', 21.
- **Proměnné** - Musí začínat velkým písmenem nebo podtržítkem, potom následuje posloupnost písmen, číslic a podtržítek. Proměnná tvořená pouze jedním podtržítkem má speciální význam. Používá se v případě, kdy nám nezáleží na její hodnotě. Proměnné jsou v Prologu pouze lokální. Stejně jako máme v predikátové logice volné a vázané proměnné, používáme tohoto principu i v Prologu. Volná proměnná není vázaná na konkrétní hodnotu, zatím co vázaná ano. Příklady: A, Beta, K3, _.
- **Struktury** - Jsou tvořeny z funktorů a argumentů. Počet argumentů udává aritu struktury. Některé operátory mohou být používány také v infixovém tvaru. Strukturou tedy mohou být i klauzule, kde se jako funktor používá infixový operátor :- Struktury jsou vždy ukončeny tečkou. Příklady: date(2,20,2019)., grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
- **Seznamy** - Jde o strukturovaný datový typ, který je vždy započatý i ukončený hranatou závorkou, mezi kterými se nachází prvky oddělené čárkami. Prvky seznamu mohou být atomy, proměnné či jiné seznamy. Příklady: [], [1,B], [[1,2,3],4,5,6].

5.1.3 Unifikace

Unifikace je proces hledání substituce za proměnné vyskytující se ve dvou termech tak, aby oba výrazy byli totožné. Pokud je substituci možné najít, výsledkem je právě nejobecnější substituce, jinak unifikace neexistuje. Unifikace v Prologu se značí znakem "=" mezi dvěma termy. Provádí se taky při interpretaci konkrétní struktury. Pro jasnější představu si uvedeme příklad. Pokusíme se unifikovat různé termy. Substituci termu A za term B budeme značit A/B:

X = [1,2,3].	-	*X/[1,2,3]*
f(X) = f(f(a)).	-	*X/f(a)*
f(X) = f(f(a),X).	-	*unifikace neexistuje*
[1,2,3] = [H|T].	-	*H/1, T/[2,3]*

5.1.4 SLD rezoluce

Pro vyhodnocování dotazů v Prologu se užívá tzv. SLD rezoluce[1]. Jedná se o princip vyhodnocování do hloubky. Pokud tedy chceme vyhodnotit nějaký podcíl cíle, dojde k prohledání hlaviček klauzulí v programu shora dolů, dokud není možné některou hlavičku unifikovat s podcílem. Pokud k unifikaci došlo, je tělo dané klauzule vyhodnocováno zleva doprava, kde každý term je vyhodnocen stejným způsobem. Pokud jsou všechny termy na pravé straně klauzule vyhodnoceny jako kladné, daný podcíl uspěl.

Pokud nastane situace, že na pravé straně bude některý z termů vyhodnocený jako nepravdivý či ho nelze unifikovat, nastává selhání cíle. V tomto případě pak nastane proces navracení, kdy se interpret vrací do místa prohledávání a snaží se unifikovat s jinou hlavičkou. Pokud se unifikace nepovede, selhává i původní cíl.

5.1.5 Důležité operace Prologu

Jazyk Prolog nabízí řadu možností. Již podle ISO Prolog[12] je definováno mnoho operací. Zde uvedu jen ty nejdůležitější operace pro realizaci mého interpretu.

- **assert(Clause)** - Tato operace vloží do databáze klauzuli (fakt nebo pravidlo). Konkrétně *assert/1* a *assertz/1* vkládá klauzule na konec a *asserta/1* jej vloží na začátek. Díky této operaci je možné vkládat nové klauzule přímo za běhu programu. Pokud je klauzule již nadefinována, operace selže.
- **retract(Clause)** - Operace odstraní z databáze klauzuli (fakt nebo pravidlo). Díky této operaci je možné odebírat klauzule přímo za běhu programu. Pokud daná klauzule neexistuje, operace skončí neúspěchem.
- **Řez - !** - Tento predikát se značí '!'. Slouží k zamezení zpětného navracení. Lze jej použít k zajištění determinismu. Pouze pokud se nepodaří nalézt řešení uvnitř řezu, je predikát opuštěn jako neúspěšný a řez je odvolán. Rozlišujeme dva základní druhy řezů. Zelený řez má vliv pouze na efektivnost programu, zatím co červený má vliv na jeho funkčnost. Algoritmus 2 je ukázkou červeného řezu. Pokud by zde řez nebyl, tak v případě, že `X>Y` by se postupně aplikovali obě pravidla a tudíž by predikát nefungoval správně. V druhém příkladě 3 je ukázka zeleného řezu. Pokud se úspěšně aplikuje první pravidlo, řez zabrání zbytečnému aplikování dalších podmínek a jeho absence by na funkčnost vliv neměla.

```
min1(X,Y,Y):- X>Y,!.
min1(X,Y,X).
```

Algoritmus 2: Příklad červeného řezu.

```
min2(X,Y,Y):- X>Y,!.
min2(X,Y,X):- Y>=X.
```

Algoritmus 3: Příklad zeleného řezu.

5.1.6 Interpret SWI-Prolog

SWI-Prolog nabízí komplexní bezplatné prostředí Prologu[25]. Od svého založení v roce 1987 byl vývoj SWI-Prologu řízen potřebami reálných aplikací. SWI-Prolog je široce používán ve výzkumu a vzdělávání, stejně jako v komerčních aplikacích. Vestavěné predikáty splňují normu ISO. Velkou výhodou tohoto interpreta je jeho rychlost, malá velikost jádra a především skutečnost, že je multiplatformní. Dále také obsahuje nízkoúrovňové rozhraní pro jazyk C, které je základním rozhraním pro podporu jiných jazyků, jako je C++, Java, C#, Python a další.

Tento interpret také obsahuje i rozsáhlý rámec webového serveru (HTTP), který může být použit jak pro poskytování služeb (REST), tak pro aplikace koncových uživatelů založených na HTML5 + CSS + JavaScript. Další výhodou je, že dokáže pracovat s více vlákny[24].

5.2 Srovnání Prologu a AgentSpeak(L)

Pokud si prohlédneme program napsaný v AgentSpeak(L) a Prologu, zjistíme, že jsou si nápadně podobné. I když pominu velmi podobnou syntaxi, mají toho mnoho společného, ale v jistých věcech se naopak liší. V této kapitole sepíši výčet společných vlastností a jejich rozdílů.

Společné vlastnosti AgentSpeak(L) a Prologu:

- Oba patří mezi logické programovací jazyky.
- Oba tyto jazyky jsou interpretované. AgentSpeak(L) je interpretován například Jasonem a Prolog například SWI-Prologem.
- Oba jazyky realizují svůj program na základě unifikace hlaviček a realizují tělo zleva doprava. U Prologu je myšleno tělo klauzule a u AgentSpeaku(L) tělo plánu.

Rozdílné vlastnosti AgentSpeak(L) a Prologu:

- Především je to jejich zaměření. Agentspeak(L) je úzce zaměřen na implementaci BDI agentů, zatímco Prolog je využíván na větší škálu úloh v umělé inteligenci, zpracování jazyka, rozhodování a řešení deklarativně zadaných úloh. Často se využívá jako vložené využití ve větších aplikacích či deduktivní databáze[26][7][11].
- Můžeme vidět podobnost mezi plány AgentSpeak(L) a klauzulemi Prologu. U AgentSpeak(L) máme navíc kontext, který v Prologu chybí.
- Pokud bychom chtěli srovnávat interní databázi pravdivých klauzulí Prologu a představy v AgentSpeaku(L) je zde jeden zásadní rozdíl. Seznam představ má každý agent vlastní, zatímco interní databáze klauzulí je globální.
- S rozšířeními AgentSpeaku(L) o Jason je odděleno teoretické a praktické uvažování.
- Zaměření AgentSpeaku(L) na BDI architektura umožňuje:
  - Existenci dlouhodobých cílů.
  - Změny a reagování na dynamické prostředí.
  - Manipulace s více ohnisky pozornosti.
- Díky zaměření AgentSpeaku(L) na BDI architektura má také jazyk jednotlivé operace zaměřené přímo na práci s představami, cíli a realizaci plánů jednotlivých agentů. Také je přizpůsoben pro práci s prostředím a komunikaci agentů.
- AgentSpeak(L) nedisponuje schopností zpětného navracení. Pokud dojde k neúspěchu podcíle, nedochází zde k unifikaci například jinou představou a vyhodnocením pro jiné hodnoty proměnných.

Většina těchto rozdílů jsou důsledkem zaměření se AgentSpeaku(L) pro popis BDI agenturního systému. To dělá z Prologu více účelový nástroj, ale AgentSpeak(L) je naopak mnohem přizpůsobenější na danou problematiku.

AgentSpeak(L) je také více svázaný s konkretním interpretem. Pro Prolog jsou vytvořeny desítky různých interpretů a je také více rozšířen, Díky tomu má i větší podporu a existuje pro něj více rozšíření. Existuje pro něj i ISO standard dokument [23], který lépe popisuje jeho vlastnosti.

Kapitola 6

FRAg

Architektura FRAg, což je zkratka Flexibly Reasoning BDI Agent, v překlad Flexibilně uvažující BDI agent, je další možností interpretace AgentSpeak(L). Tento koncept byl navržen v [28], dále byl představen v roce 2003 na konferenci v Košicích [18]. Jako takový interpretuje systém obsahující určité představy, plány, události a záměry. Cíle mohou být ve formě testů nebo dosahování cílů, jak je běžné v téměř každém BDI systému [20]. Události jsou opět vyjádřeny formou predikátů, záměry jsou ve formě zásobníku plánů a plány jsou struktury obsahující spouštěcí událost, mohou obsahovat podmínku aplikace a nakonec obsahují tělo. Některé systémy, jako Jason 2.1 [4] nebo 2APL [13], umožňují definovat představy jako některá pravidla Prologu. Ale tuto možnost v systému FRAg uvolňujeme. Předpokládáme, že základ představy agenta sestává ze souboru přesvědčení ve formě predikátů. Odlišuje se také v tom, že FRAg nedodržuje přesně interpretační specifikaci předloženou v originálním dokumentu. FRAg vznikl v rámci snahy nalézt způsob, jak zlepšit proces výběru cílů, plánů nebo záměrů v BDI systémech. Agent BDI, který uvažuje o prostředcích vhodných pro událost, si vybere ze skupiny možností. Tyto možnosti jsou reprezentovány některými plány, které jsou vhodné pro dosažení cíle nebo dílčího cíle. To znamená, že pro jeden plán můžeme najít jeden nebo více unifikátorů, které sjednocují spouštěcí událost se zbytkem plánu, a pak můžeme najít jiné unifikátory, které sjednocují spouštěcí podmínky plánu se základem představy agenta. V nynějších realizacích, kdy je plán ze sady voleb vybrán jako zamýšlený význam pro událost, jsou některé substituce aplikovány bezprostředně před přidáním plánu do odpovídajícího zásobníku záměrů.

Podle FRAg může záměr mít více než jednu možnost, jak jej dosáhnout. Konkrétněji předpokládáme, že každý plán v rámci záměru může obsahovat kontext plánu, který lze chápat jako dočasnou paměť vztahující se ke každému plánu, který byl vybrán pro tento záměr. Tyto kontexty obsahují všechny dostupné substituce, které souzní s původní událostí, pro kterou byl plán vybrán v daném stavu základu představy agenta. Pak jeden plán může představovat více než jen jediné chování agenta. Obecně může agent provádět určité výpočty, například může otestovat svůj základ představy nebo stanovit cíl úspěchu dříve, než se musí rozhodnout, kterou konkrétní vnitřní nebo vnější činnost má učinit. Agent obvykle provádí akci, která vyžaduje určité konkrétní zdroje, nebo se zabývá konkrétním směrem a tyto zdroje jsou určeny uvnitř akčního predikátu. Pokud například agent uvažuje o tom, jak se dostat na místo, kde může dostat nějaký Kool-Aid, pak se musí rozhodnout, zda jde pěšky, autobusem, na kole nebo autem. Pokud chce koupit pivo, neměl by myslet na auto a jízdní kola jako na dopravní prostředek a také místa, kam by chtěl jet, se mohou lišit od míst, kde je nabízen Kool-Aid.

Takže systému FRAg odkládá rozhodování o konkrétních variabilních substitucích, dokud to není nutné. Tímto způsobem můžeme vzít jeden plán z báze plánu agenta a jeho struktury těla spolu se souborem možných substitucí představujících možné způsoby chování. Říkáme, že udržujeme slabé případy plánu, kterými rozumíme, že plán může obsahovat některé volné proměnné a k těmto proměnným je přidruženo více možných substitucí.

6.1 Motivační příklady

Pro lepší pochopení zde uvedu dva příklady, na kterých bude patrnější jak systém FRAg funguje. Tyto příklady jsou převzaty od Doc. Ing. Františka Zbořila, Ph.D.

V prvním příkladu máme chlapce Adama, který vlastní sbírku karet. Chce ovšem získat nějaké peníze a tak se pokusí některou kartu prodat. V první řadě ovšem musí určit, které z karet nabídne k prodeji. Můžeme tedy napsat plán AgentSpeak(L), který odpovídá Adamovu záměru PA (algoritmus4).

@PA: +!getMoney(Amount) <- !selectCard(C),!sellCards(C,Amount).

Algoritmus 4: Plán PA k 1. příkladu.

Plán pro výběr karet lze realizovat poměrně snadným způsobem. Adam jen reviduje svou sbírku karet a pak ví, které karty bude prodávat. Po procesu revize se v jeho základu představy objevují představy s kartami, které má dvakrát nebo vícekrát. Odpovídající plán by pak měl reagovat na událost `+!SelectCard(C)` (algoritmus 5) a jako výsledek by měl být atom omezen na proměnnou C. Vytvořme operaci `reviseCollection`, která mění Adamovi představy, podle předpokladu a v jeho základu představy se tudíž objevují představy `cardToSell(Card)` karet, které má Adam dvakrát.

@PB: +!selectCard(Card) <- reviseCollection; ?cardToSell(Card).

Algoritmus 5: Plán PB k 1. příkladu.

Následně chce najít způsob, jak je prodat. Může existovat několik způsobů, jak karty prodat. Například nabídnout je přes internet, navštívit setkání obchodníků s kartami atd. Ale mimo jiné, tento druh karet také sbírá jeho přítelkyně Betty. Rozhodl se tedy navštívit Betty a zeptat se jí, zda má zájem některou z karet koupit. Výsledkem takového procesu je částka, kterou Adam tímto obchodem vydělá. Tento scénář může být reprezentován například plánem PC (algoritmus 6).

@PC: +!sellCards(Cadr,Amount) <- dealWith(betty, Card); ?bettyWants(Card, Amount).

Algoritmus 6: Plán PC k 1. příkladu.

Takto navržený program selže jak v systému JASON, tak v systémech APL2, pokud karta, kterou si Adam vybere po revizi své sbírky, se nehodí pro Betty, ale když si vybere jinou a Betty souhlasí s jejím nákupem. Důvodem je, že v průběhu praktické fáze uvažování je vybrán první použitelný a relevantní plán a jsou na něj aplikovány první vhodné substituce. Kdyby si Adam vzpomněl na všechny karty, které by chtěl prodat dohodou s Betty, bylo by možné uspět.

Druhý příklad demonstruje situaci, kdy agent Klára sleduje více než jeden záměr. Zpočátku je Klára v situaci, kdy musí koupit dárek svému kamarádovi Danovi a má také v plánu užít si pěkný den. Klářin plán na hezký den podmiňuje výlet na hrad. Program pro Kláru může zahrnovat následující dva cíle a sedm přesvědčení (algoritmus 7).

```
!buyPresent.
!makeTrip.
presentSells(supermarket, book).
presentSells(pernstejn, figure).
PresentSells(pernstejn, thimble).
presentSells(bouzov, postcard).

castle(pernstejn).
castle(bouzov).
myPosition(home).
```

Algoritmus 7: Představy a cíle k 2. příkladu.

Chování Kláry je tvořeno několika plány. Může přijmout dva záměry. Za prvé koupit dárek a za druhé udělat si výlet. Můžeme mít tyto čtyři plány (algoritmus 8).

```
@P1: +!goTo(X):myPosition(X) <- T.
@P2: +!goTo(X) <- ?demandsMean(X,Y), allocateMean(Y), goToBy(X.Y).
@P3: +!buyPresent <- ?presentSells(X,Y); !goTo(X); ...
@P4: +!makeTrip <- ?castle(X); !goTo(X); ...
```

Algoritmus 8: Plány k 2. příkladu.

Pokud je tento program interpretován obvyklým způsobem, pak první záměr bude obsahovat plán P3, který nakonec vyvolá událost `+!GoTo(supermarket)`, protože testovací cíle budou první unifikovat představu `presentSells(supermarket, kniha)`. Druhý záměr bude vytvořen pro druhý cíl nejvyšší úrovně `!MakeTrip`, který bude vykonán prostřednictvím plánu P4. V tomto případě by agent šel na hrad Pernštejn.

Ale proč Klára nekupuje dárek na zámku Pernštejn? Je to proto, že uspořádání plánů determinovalo výběr substituce ve vybraném zamýšleném záměru.

Těmito příklady bylo prokázáno, že úspěšné provedení programu AgentSpeak(L) závisí na způsobu, jakým je zamýšlený záměr události vybrán z některých dostupných možností. Pokud by agent Adam vybral jinou kartu, například kartu X, pak by tato karta byla prodána Betty. A kdyby Klára spojila oba své úmysly dohromady, ušetřila by čas a úsilí.

6.2 Realizace FRAg pomocí Prologu

Programovací jazyk Prolog má jisté vlastnosti, které jsou pro realizaci systému FRAg ideální.

Při vyhodnocování se vždy prohledává databáze od shora dolů a na hladinách, kde existují alternativy, se pak pokračuje zleva doprava. O splnění cílů rozhoduje unifikace. Srovnává se vždy aktuální cíl s faktem či hlavou pravidla v databázi. Součástí úspěšného srovnání je i případný vznik vazby proměnné na hodnotu. Pokud nedojde k splnění cíle, může program v Prologu použít tzv. navracení (anglicky backtracking), při kterém dojde ke zrušení

vazeb proměnných na hodnoty a hledání jiného řešení. Po nalezení řešení může navracení vyvolat i uživatel a to stisknutím klávesy středník nebo Tab.

Pokud tedy budeme mít zadefinovanou databázi jako v (algoritmus 9) na otázku `man(P)` dostaneme odpověď uvedenou v algoritmu 10.

```
man(karel).
man(josef).
man(M):- child(M), not(girl(M)).
child(jana).
child(jiri).
girl(jana).
girl(lucie).
```

Algoritmus 9: Definice databáze v Prologu.

```
P = karel ;
P = josef ;
P = jiri.
```

Algoritmus 10: Odpověď na otázku man(P).

V našem případě se tedy nejprve Prolog pokusí porovnat `man(P)` a `man(karel)`. Proměnná `P` byla nespecifikována, mohla tak vzniknout vazba na atom `karel`. Cíl je splněn, Prolog vypíše první odpověď. Pokud máme zájem o vyhledání alternativního řešení, požádáme Prolog o znovu splnění cíle. Tedy použijeme klávesu středník. Prolog zruší vazbu na atom `karel`, označí si již splněný fakt a pokouší se najít další řešení. Protože databáze se prohledává směrem dolů, dalším porovnáním bude `man(P)` a `man(josef)`. Vzniká nová vazba proměnné `P` a to na atom `josef`. Cíl je opět splněn, vypíše se druhá odpověď. Po stisku klávesy středník dojde ke zrušení vazby proměnné `P`, označení použitého faktu a hledání dalšího cíle. Při hledání další odpovědi pracuje Prolog s pravidlem a postupuje následovně:

Nespecifikované proměnné `P` a `M` se stanou sdílenými, tzn. obě proměnné mohou mít vazbu na jedinou hodnotu. Vyhledá se první dítě, což je v našem případě `jana` a na proměnné `P` i `M` se naváže hodnota `jana`. Poté hledá predikát `girl(jana)`. Toto hledání je úspěšné, proto bude výraz `not(girl(jana))` vyhodnocen jako nepravdivý a proto se použije navrácení. Prolog se vrátí k predikátu `child(M)`, zruší vazbu na proměnnou `jana` a snaží se najít jinou vazbu, což v našem případě bude atom `jiri`. Nyní tedy program hledá vazbu na predikát `man(jiri)`, tento predikát nenajde, čili `not(man(jiri))` je splněno, čímž splní cíl a je vypsána odpověď `jiri`.

Další hledání probíhá podobně a je neúspěšné. Z výše napsaného vyplývá, že Prolog prohledává strom programu (derivační strom) shora dolů a na hladinách. Tam, kde existují alternativy, pak zleva doprava.

Díky predikátu `fail` můžeme dosáhnout toho, že se provedou veškeré možné unifikace a postupným navracením se provedou všechny možné variace programu. Jako jednoduchý příklad uvedu program na vypsání všech mužů z databáze (algoritmus 11).

Tento přístup nám dovolí splnit předpoklad systému FRAg. Pokud použijeme databázi Prologu k evidenci představy agenta, můžeme tímto principem splnit plán agenta pro

```
write_all_men:- man(M), write(M),nl,fail.
write_all_men.
```

Algoritmus 11: Příklad programu pro výpis všech mužů z databáze.

všechny možné substituce. Dále díky tomuto principu dokážeme vyřešit problém, kdy nedospějeme ke správnému řešení jen proto, že se záměry plnily v jiném pořadí. Jednoduše vykonáme veškeré možné permutace pořadí.

Kapitola 7

Návrh a implementace interpreta agentního systému řízeného záměrem v jazyce Prolog

Tato kapitola popisuje mé osobní přínosy v této práci. Konkrétně jde o návrh a implementaci programu v jazyce C++ pro převod agentního systému popsaného v jazyce AgentSpeak(L) na interní reprezentaci v Prologu a interpretu, který interpretuje systém řízený záměrem v jazyce Prolog s podporou chování FRAg systému. Můj interpret byl inspirován interpretem jazyka AgentSpeak(L) Jason. Navržený interpret dokáže interpretovat systém jednak klasickým způsobem a nebo využít při interpretaci systém FRAg, kdy interpretace probíhá stejným způsobem jako je tomu u Jason, ale použije systém zpětného nahrazení. Tato funkce je volitelná.

7.1 Návrh automatického převodníku z jazyka AgentSpeak(L) na interní reprezentaci v Prologu

Před samotnou implementací je nutno převést agentní systém popsaný v jazyce AgentSpeak(L) na interní reprezentaci.

Vnitřní struktura programu v Prologu je seznam představ, cílů a plánů pro jednotlivé agenty a jeho BNF gramatiku můžete vidět v následující tabulce 7.1. <ATOM> zde značí posloupnost znaků, začínající malým písmenem a <TERM> je struktura, jak je popsaná v kapitole 5.1.2. Samotný program je seznam (`list_of_agents`) jednotlivých menších programů. Každý tento program patří jednomu agentovi. Programy konkrétních agentů jsou záznamy (`agent`), které tvoří čtyři prvky. První je jméno daného agenta (`agent_name`). Druhý je seznam přesvědčení, které agent zná při startu programu. Jako přesvědčení či představa může být buď term a nebo pravidlo. Představy musí mít v této sekci jasně definovanou podobu. Pokud je to term, bude to `bel(belief, owner_name, source)`, kde `belief` je daná představa, což může být jakýkoliv term, `owner_name` je jméno agenta, který tuto představu má a `source` je zdroj představy. V tomto případě to bude `self`. Pokud je představa udaná pravidlem, může to být jakékoliv pravidlo, ale jednotlivé představy zde musí mít opět stejný tvar, jako když jsou zadány termem. Pravidlo může vypadat například takto: `bel(same_pos(),agent1,_) :- bel(my_pos(X,Y),agent1,_), bel(your_pos(X,Y)).` Třetí položka u plánu daného agenta je seznam cílů (`list_of_goals`). Jedná se o cíle, které se bude snažit daný agent splnit. Každý cíl je reprezentován jedním termem. Poslední

čtvrtou položkou každého programu daného agenta je seznam plánů pro realizaci cílů (`list_of_plans`). Každý jednotlivý plán (`plan`) je opět struktura o pěti položkách. První je typ plánu (`plan_type`). Význam je stejný jako v AgentSpeak(L) a je uveden v tabulce 4.2. Druhý parametr je atomicita (`atomicity`). Pokud je atomicita `'atomic'`, nebude se měnit aktuální záměr, dokud nebude celý proveden. Dalším parametrem je hlavička plánu (`head_of_plan`). Předposledním parametrem je seznam podmínek pro spuštění daného plánu (`list_of_contexts`) a posledním parametrem je seznam operací (`list_of_operations`), které slouží k realizaci daného plánu. Každá operace je struktura s typem operace (`operation_type`) a samotnou operací, což je term. Typ operace může být buď dosahovaný cíl (`'!'`), testovací cíl (`'?'`), přidání nebo odebrání představy (`'+'` nebo `'-'`) a nebo vlastní operace (`'op'`).

```
program            ->   list_of_agents
list_of_agents     ->   "[" [ agent ("," agent)* ] "]"
agent              ->   "(" agent_name "," list_of_beliefs ","
                        list_of_goals "," list_of_plans  ")"
agent_name         ->   <ATOM>
list_of_beliefs    ->   "[" [ belief ("," belief)* ] ] "]"
belief             ->   "(" (<TERM>|complex_belief) ")"
complex-belief     ->   <TERM> ":-" list_of_terms "."
list_of_terms      ->   <TERM> ("," <TERM>)*
list_of_goals      ->   "[" [ goal ("," goal)* ] ] "]"
goal               ->   <TERM>
list_of_plans      ->   "[" [ plan ("," plan)* ] "]"
plan               ->   "(" plan_type "," atomicity ","
                        head_of_plan "," list_of_contexts "," list_of_operations ")"
plan_type          ->   ("'+'"|"'-'"|"'+!'"|"'-!'"|"'+?'"|"'-?'"))
atomicity          ->   ("'atomic'"| "'noatomic'")
head_of_plan       ->   <TERM>
list_of_contexts   ->   "[" [ context ("," context)* ] "]"
context            ->   <TERM>
list_of_operations ->   "[" [ operation ("," operation)* ] "]"
operation          ->   "(" operation_type "," <TERM> ")"|
operation_type     ->   ("'op'"|"'!'"|"'?'"|"'+'"|"'-'")
```

Tabulka 7.1: BNF gramatika interní reprezentace programu.

Pro jasnější představu zde uvedu na porovnání dva krátké programy. Jeden je napsán v AgenntSpek(L) (algoritmus 12) a druhý má totožné chování, ale je napsán v interní reprezentaci srozumitelné Prologu (algoritmus 13). Jak vidíte, obě podoby programu jsou podobné a intuitivně převoditelné. U verze pro Prolog je pouze náročnější dodržovat přesnou strukturu proměnné `Program`.

7.2 Návrh interpretu

Můj návrh interpretu vychází z modelu znázorněného na obrázku 4.1. Vizualizovaný návrh běhu mého interpretu naleznete na schématu 7.1. Systém se sestává z několika procesů. Fáze interpretace probíhají ve stejném pořadí nezávisle na tom, zda jsou interpretovány po způsobu Jason a nebo jako FRAg systém. Rozdíl je v povolení zpětného navracení u systému

```
/*beliefs*/
card(adam,card1,3).
card(adam,card2,1).
card(adam,card3,2).
card(adam,card4,3).
card(adam,card6,4).
bettyWants(card4,500).
bettyWants(card2,300).
bettyWants(card5,100).
bettyWants(card6,100).

/*goals*/
getMoney(Amount).

/*plans*/
+!getMoney(Amount) <- !selectCard(C); !sellCards(C,Amount).
+!selectCard(Card) <- reviseCollection(); ?cardToSell(Card).
+!sellCards(Card,Amount) <- ealWith(betty, Card); ?bettyWants(Card, Amount).
```

Algoritmus 12: Příklad reprezentace programu v AgentSpeak(L).

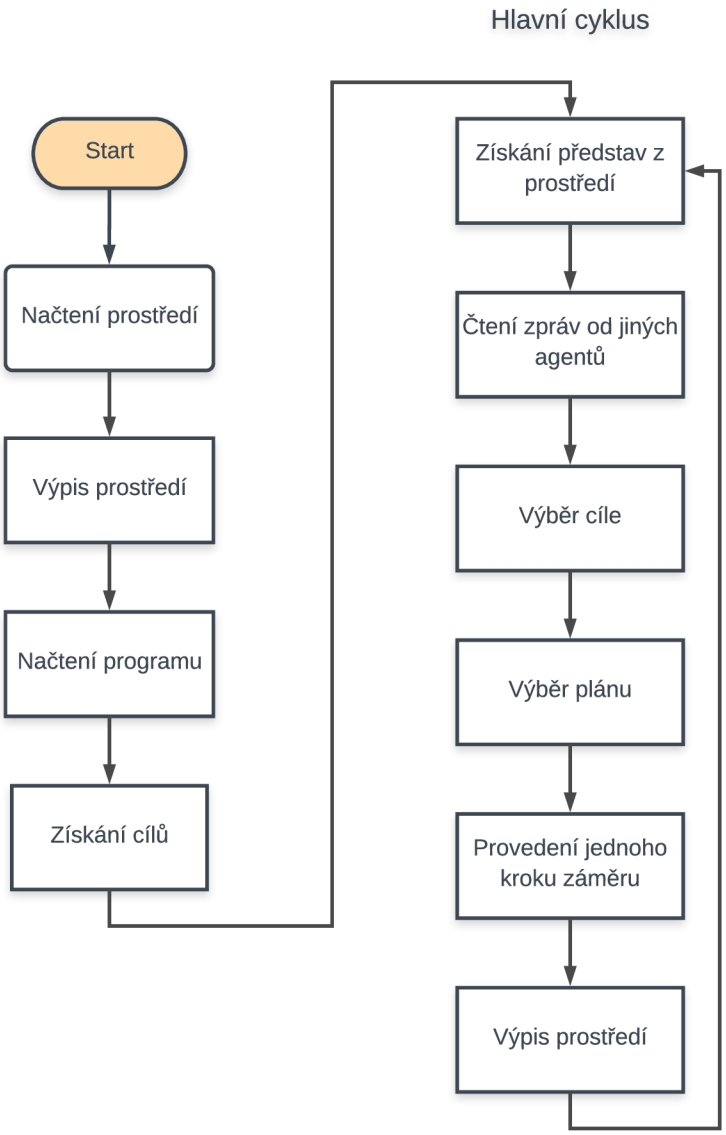
FRAg. V každém cyklu interpretace se provedou dané úkoly pro všechny agenty. To znamená, že jeden cyklus interpretace je zároveň jeden cyklus pro každého agenta v systému.

Nyní si popíšeme jednotlivé části systému:

- **Načtení prostředí** - Tato komponenta slouží k načtení struktury prostředí, včetně její počáteční inicializace. Není nutné, aby interpret s prostředím zacházel. Pokud je interpret spuštěn v režimu bez interakce s prostředím, je tato fáze vynechána.
- **Výpis prostředí** - Zobrazí aktuální stav prostředí. Opět je tato fáze vynechána, pokud je agentní systém navržen bez prostředí.
- **Načtení programu** - Chování každého agenta je charakterizováno programem, který je sestaven přímo v interní reprezentaci pro daného interpreta. Zde si interpret tuto strukturu načítá, aby ji mohl interpretovat. Aktuálně neexistuje automatická konverze z jazyku AgentSpeak(L) do této formy. Tudíž pokud by uživatel chtěl vytvořit transformaci z AgentSpeak(L) na interní reprezentaci, musí ji vytvořit ručně.
- **Získání cílů** - Z programu jsou načteny počáteční cíle pro každého agenta a přidány do množiny cílů.
- **Získání představ z prostředí** - Z prostředí jsou načteny představy a jsou přidány do množiny představ pro konkrétní agenty. Pokud na základě prostředí vznikají nové cíle jsou přidány do množiny cílů. Tato sekce je také provedena pouze pokud má agentní systém prostředí.
- **Čtení zpráv od jiných agentů** - Zde dochází k načtení komunikace od jiných agentů, výběr "sociálně přijatelných" zpráv a přidání dalších případných cílů do množiny cílů.

```
Program = [
%first agent name
(adam,
    %beliefs
    [
        %example beliefs
        bel(card(adam,card1,3),adam,self),
        bel(card(adam,card2,1),adam,self),
        bel(card(adam,card3,2),adam,self),
        bel(card(adam,card4,3),adam,self),
        bel(card(adam,card6,4),adam,self),
        bel(bettyWants(card4,500),adam,self),
        bel(bettyWants(card2,300),adam,self),
        bel(bettyWants(card5,100),adam,self),
        bel(bettyWants(card6,100),adam,self)
    ],
    %goals
    [
        %goals
        getMoney(Amount)
    ],
    %plans
    [
        ('+!','noatomic',
            getMoney(Amount),
                [],[
                    ('!',selectCard(C)),
                    ('!',sellCards(C,Amount))
                ]
        )
        ('+!','noatomic',
            selectCard(Card),
                [],[
                    ('op',reviseCollection()),
                    ('?',cardToSell(Card))
                ]
        )
        ('+!','noatomic',
            sellCards(Card,Amount),
                [],[
                    ('op',dealWith(betty, Card)),
                    ('?',bettyWants(Card, Amount))
                ]
        )
    ]
    )
].
```

Algoritmus 13: Příklad interní reprezentace programu pro následnou interpretaci v Prologu.



Obrázek 7.1: Model interpretu

- **Výběr cíle** - Z množiny událostí je vybrán cíl, který se bude v tomto cyklu realizovat. Pokud je interpret v základním režimu, vybírá vždy první událost z množiny, pokud jde o interpretaci jako FRAg, slouží tento bod jako křižovatka. Prvně se vybere první a při zpětném navracení se vybere další, tak se postupně provedou všechny variace.
- **Výběr plánu** - Jsou načteny všechny relevantní plány. Dále je ověřena jejich platnost a nakonec je vybrán jeden platný plán, který se bude dále realizovat.
- **Provedení jednoho kroku záměru** - Je vybrán jeden záměr z plánu a ten je následně vykonán. Může se jednat o přidání nového cíle, poslání zprávy, akce, která ovlivní prostředí, dotaz na představy a podobně. Také je aktualizována množina plánů. I tato část se bude chovat odlišně pokud je systém v režimu FRAg. Opět se povolí možnost zpětného navracení. Pokud je například jako operace čtení představ agenta a ta může substituovat proměnné více způsoby, vybere se prvně jedna varianta a při zpětném navrácení další. Tento postup se opakuje, dokud se neprovedou všechny varianty.

Interpret podporuje také speciální funkce, které vyžadují speciální popis:

- **Atomicita** - Jednou z vlastností plánu je jeho atomicita. Pokud je plán atomický, tak po dobu jeho plnění se nesmí plnit žádné jiné záměry daného agenta. Pokud atomický není, je možné při výběru události, vybrat i jinou.
- **Systém s prostředím** - Agentní systém může obsahovat prostředí, ale také nemusí. Pokud jej obsahuje, vyžaduje to větší režii ze strany tvůrce agentního systému. Musí nadefinovat jeho vzhled, operace nad ním, funkci pro jeho výpis a funkci pro detekci nových událostí z prostředí.
- **Systém FRAg** - Interpret podporuje sytém FRAg. Pokud je interpret spuštěný v režimu podpory tohoto systému, dokáže provádět veškeré substituce při operacích a nemusí vždy brát jen první nabídnutou možnost. Dále provádí permutace nad výběrem aktuálně prováděných cílů. Díky těmto dvou vlastnostem interpret nenabízí pouze jedno řešení dosažení cílů, ale může jich nabídnout hned několik.

7.3 Implementace převodníku z jazyka AgentSpeak(L) na interní reprezentaci v prologu

První akce, která je ještě před samotnou interpretací notná, je převod agentního systému popsaného jazykem AgentSpeak(L), převést na interní reprezentaci, tak aby jej mohl interpret jednoduše začít interpretovat.

Pro tento účel byl sestaven jednoduchý překladač v jazyce C++, který tento převod automaticky provede. Programu je předložen soubor s příponou **mas2j**. Ten obsahuje jednoduchou strukturu, kde je jsou uveden seznam agentů systému a jejich počet- Tento formát využívá pro své projekty i interpret Jason, ale pro náš případ je zjednodušen. Pro systém o třech agentech, kdy dva agenti budou stejného typu může tento soubor vypadat takto:

```
MAS name_system {
  agents: agent1;
          agent2 #2;
}
```

Po tom co získáme jednotlivé jména agentů a jejich počet, jsou nalezeny v adresáři tohoto souboru, nebo jeho podadresářích, soubory s programem jednotlivých agentů. Název souboru musí být sestaven ze jména agenta a přípony **asl**. Pro ukázkový systém by bylo chování agenta agent1 popsáno v souboru `agent1.asl` a chovaní agenta agent2 v v souboru `agent2.asl`.

V těchto souborech již je standardní popis chování systému v jazyce AgentSpeak(L), tak jak byl popsán v návrhu.

Takto vypadá jeho spuštění:

```
./languageTransform <mas2_file>
```

7.4 Implementace Interpretu

Po automatickém převodu z AgentSpeaku(L) dostaneme interní reprezentaci agentního systému v Prologu. Ten předložíme samotnému interpretu. Tato kapitola popisuje implementaci modelu, který byl předveden v předchozí kapitole. Systém byl implementován v jazyce Prolog, který je interpretovatelný v SWI-prologu.

7.4.1 Hlavní cyklus

Nyní si uvedeme, jak vypadá hlavní cyklus programu a operace, které mu předcházejí. Také si popíšeme význam jednotlivých funkcí. V algoritmu 14 můžeme vidět hlavní klauzuli, kterou se interpret spouští. Při implementaci nebylo na rozdíl od návrhu realizovaná komunikace mezi agenty-

```
interpreter():-     get_environment_starter(E),
                    print_e_starter(E),
                    get_program(P),
                    remove_old_beliefs(),
                    insert_beliefs(P),
                    get_goals(P,G),
                    get_plans(G,Pl),
                    get_empty_atom_list(P,A),
                    get_empty_atom_list(P,L),
                    go_all(P,Pl,E,M,A,L).

go_all(P,Pl,E,M,A,L):-
                    go_all_loop(P,Pl,E,M,A,L).
go_all(_,_,_,_,_,_):-
                    frag(0), find_plan(0),
                    log('INFO',['Plan not found']),!.
go_all(_,_,_,_,_,_).

go_all_loop(P,Pl,E,M,A,L):-
                    main_loop(P,Pl,E,M,A,1,L),
                    fail.
```

Algoritmus 14: Spouštěcí klauzule.

- **Predikát *get_environment_starter(E)*** - Agenti se pohybují v prostředí, které je potřeba si nadefinovat. Tento predikát vrací strukturu reprezentující toto prostředí. Pokud je systém, který je interpretován bez prostředí, vrací prázdný seznam.
- **Predikát *print_e_starter(E)*** - Tento predikát slouží k výpisu aktuálního stavu prostředí. Jeho podoba se bude lišit na základě druhu prostředí. Pokud je systém bez prostředí, tato operace nic nevypisuje.
- **Predikát *get_program(E)*** - V předchozí kapitole jsme si uvedli, jak vypadá program v interní reprezentaci. Tento predikát vrátí v proměnné E právě takto vypadající strukturu. Je to jediná funkce, kterou je nutno nadefinovat v každém systému.
- **Predikát *remove_old_beliefs()*** - Jednotlivé představy agenta jsou reprezentovány v interpretu fakty, které jsou uloženy přímo v databázi Prologu. Pro zamezení konfliktů s jinými programy jsou veškeré představy vymazány.
- **Predikát *insert_beliefs(P)*** - Tento predikát vloží do databáze nové představy, které byly inicializovány přímo v programu.
- **Predikát *get_goals(P,G)*** - Další predikát získá z programu inicializační cíle.

- **Predikát *get_plans(G,Pl)*** - Cíle si převedeme na strukturu, v které uchováváme jednotlivé plány.
- **Predikát *go_all(P,Pl,E,M,A,L)*** - Slouží ke spuštění hlavního cyklu programu, dále kontroluje a zpracovává neúspěšné hledání cílů. Pokud je interpret v režimu FRAg, zajišťuje funkce realizaci zpětného navracení. Pokud interpret není v režimu FRAg a nepodařilo se nalézt řešení, tato funkce o tom informuje.
- **Predikát *go_all_loop(P,Pl,E,M,A,L)*** - Vykonává zpětné navracení, které je nutné, pokud je interpret spuštěn v režimu FRAg.
- **Predikát *main_loop(P,Pl,E,M,A,1,L)*** - Jedná se o predikát, který spouští hlavní cyklus interpreta. Na vstupu je P, což je program řídící jednotlivé agenty, proměnná Pl, která reprezentuje aktuální cíle agentů. Proměnná E, která obsahuje prostředí systému. Čtvrtá proměnná reprezentuje příchozí zprávy od jiných agentů. Proměnná A obsahuje aktuální událost, pokud je program v módu ATOMIC. Předposlední proměnou je číslo cyklu běhu interpreta. Poslední proměnná je seznam všech vykonaných operací od začátku běhu systému.

Když nyní známe veškeré operace, které se musí vykonat, před samotným hlavním cyklem, můžeme si jej popsat. Vysvětlíme si jeho běh přímo na jeho algoritmu 15.

```
main_loop(P,Pl,E,M,A,C,L) :-
    get_beliefs_from_enviroment_starter(P,E,New_bel),
    print_all_believs(),
    write_new_bel(P,New_bel),
    add_new_beliefs_to_plan(New_bel,Pl,Pl_with_bel),
    select_plan(Pl_with_bel, A, Select_plan, Pl2, ADetect),
    performing_one_step(Select_plan, ADetect, E,
        P, New_plan, ADetect2, E2, L, L2, B),
    control_fail(B,Pl2,R2),
    R2 = 1,
    merge_plans(Pl2,New_plan,Pl3,ADetect2,New_A),
    print_e_starter(E2),
    C2 is C + 1,
    control_end(Pl3,New_A,L2),
    main_loop(P,Pl3,E2,M,New_A,C2,L2).
main_loop(_,_,_,_,_,_,_).
```

Algoritmus 15: Hlavní cyklus interpretu.

I nyní si objasníme významy jednotlivých predikátů. Zde již budou některé operace složitější:

- **Predikát *get_beliefs_from_enviroment_starter(E, New_bel)*** - Podle aktuálního stavu prostředí je nutno vygenerovat nové představy. Ty jsou v prvé řadě přidány do databáze představ, ale také se vrací v proměnné New_bel, aby mohl být vznik nových představ zahrnut do cílů.
- **Predikát *print_all_believs()*** - Vypíše všechny aktuální představy agenta. Pro výpis je nutné mít úroveň výpisu nastavený na 3.

- **Predikát *add_new_beliefs_to_plan(New_bel,Pl, Pl_with_bel)*** - Přidání představ prostředí do cílů.

- **Predikát *select_plan(Pl_with_bel, A, Select_plan, Pl2, ADetect)*** - Tento predikát je určen k výběru jedné události z cílů a následně i konkrétního plánu, který se bude vykonávat. Obsahuje i kontrolu platnosti plánu. Jedná se o důležitý predikát, a proto si jej představíme detailněji. Vstupní proměnná *Pl_with_bel* obsahuje aktuální cíle obohacené o nové cíle z prostředí. Je vybrána jedna událost z množiny cílů. V proměnné *A* je aktuální událost, pokud je program v ATOMIC režimu. Pro daný cíl se vybere konkrétní platný plán, ten je vrácen v proměnné *Select_plan*. Tento plán je vyjmut z původních cílů. Takto upravené cíle jsou vráceny v proměnné *Pl*2. Poslední parametr *ADetect* slouží k detekci atomicity.

- **Predikát *performing_one_step(Select_plan, ADetect, E, P, New_plan, ADetect2, E2, L, L2, B)*** - Tento predikát slouží k provedení jednoho kroku plánu. Opět jde o podstatnou část implementace. V proměnné *Select_plan* je aktuální plán pro provedení. Vybere se z něj první operace a ta se vykoná. Může se jednat o operaci nad prostředím (tyto operace je nutno naprogramovat v Prologu pro dané prostředí), může to být poslání zprávy (zatím neimplementováno), přidání představy, či cíle atd. Tyto operace mohou ovlivnit prostředí, seznam přijatých zpráv, množinu představ a vytvořit nové cíle. Nová operace je vložena do seznamu vykonaných operací. Funkce také detekuje zdar, či nezdar operace.

- **Predikát *control_fail(B,Pl2,R2)*** - Pokud se operace nezdařila, dojde zde k odstranění dalších nežádoucích operací a navíc, pokud dojede ke kombinaci neúspěšného plánu a absence dalších operací v něm, detekuje operace neúspěch cíle.

- **Predikát *merge_plans(Pl2,New_plan,Pl3,ADetect2,New_A)*** - Sestavení nových plánů pro další cyklus.

- **Predikát *end_test()*** - Testuje vstup z klávesnice. Zastaví program dokud není stisknuta klávesa ENTER, v tom případě pokračuje dalším cyklem a nebo klávesa q, v tomto případě ukončuje interpretaci. Proběhne i kontrola, zda nejsou již všechny cíle vykonány.

7.4.2 Vývojové prostředí

Pro zjednodušení práce s interpretem bylo vyvinuto jednoduché vývojové prostředí, které pomůže tvůrci agentních systémů s interpretem pracovat a vytvořit si vlastní program. Toto prostředí je pouze konzolové.

Zde je popis parametrů interpretu:

```
./interpreter.pl {-V|--version|-h|--help}
./interpreter.pl {-p | --program} <file> [-edf] {-l | -log} <log_level>
./interpreter.pl {-n | --new} <file> [-e] {-l | -log} <log_level>

  -h, --help            Vypíše nápovědu
  -V, --version         Vypíše verzi
  -p, --program <file>  Zadání souboru s~programem agentního systému
  -n, --new <file>      Vytvoření souboru se vzorovým
```

```
                        programem agentního systému
  -l, --log <log_level> Nastavení úrovně výpisů
                        (ERROR=1, INFO=2,DEBUG=3)
  -e, --enviroment      Program bude pracovat s~prostředím
  -f, --frag            Program bude pracovat v~režimu FRAGg
  -d, --debug           Program bude pracovat v~debug módu
```

Samotný interpret je oddělen od funkcí, které se musí vytvořit při tvorbě agentního systému. Tyto uživatelem nadefinované funkce musí být součástí vstupního souboru. Tento soubor se přidá za parametr `-p`. Pro vytvoření šablony programu je možné tento soubor vytvořit. Vytvoří se velmi základní operace s nápovědou, jak má systém vypadat. Docílíte toho pomocí parametru `-n`. Pro vytvoření šablony pro agentní systém s prostředím je nutné kombinovat jej s parametrem `-e`. Pro procházení programu po cyklech slouží parametr `-d`. V tomto režimu můžete procházet krok za krokem chování agenta a budou se vypisovat základní informace o jeho představách, prováděných operacích a případném prostření. Pro spuštění interpretace s prostředím, musíte využít parametr `-e`. Pro jednotný výstup byla definována funkce pro výpis na standardní výstup `log(LogLevel,ListTerm)`, kde `ListTerm` je seznam hodnot pro výpis a `LogLevel` značí typ zprávy. Podle typu se budou vypisovat jednotlivé zprávy při konkrétní úrovni výpisů, která se přidá za parametr `-l`. Defaultně je nastaven na hodnotu 2. To znamená, že vypisuje zprávy s typem 'INFO' a 'FAIL'. V režimu debug je implicitně nastaven log na stupeň 3 a proto se vypisují i zprávy 'DEBUG'. Ve výpisu je aktuální čas, typ zprávy a text samotné zprávy. Funkce `log` lze použít i v operacích definovaných v rámci agentního systímu. Volání může vypadat například tekto:

```
log('INFO',['Name agent ', N ,' move on position ',X,',',Y]).
```

Pro zapnutí režimu FRAg slouží parametr `-f`.

Každý agentní systém musí mít ve vstupním souboru nadefinované tyto funkce:

```
%get_program(--Program)
% - Parametr Program definuje program agentního systému
get_program(Program)
```

Tato funkce vrací program agentního systému. Právě tento program je následně interpretován. Jde o jedinou funkci, pokud se jedná o systém bez prostředí, která musí být vždy definována.

```
%get_environment(--Enviroment)
% - Parametr Envidoment je vámi definované prostředí.

get_environment(Enviroment)
```

V této funkci si uživatel definuje strukturu prostředí agentního systému. Struktura nemá danou konkrétní podobu. Uživatel si prostředí nadefinuje podle vlastních potřeb.

```
%print_e(+Enviroment)
% - Parametr Enviroment je vámi definovaní prostředí.
print_e(Enviroment)
```

Funkce vypisuje ve vámi definované podobě prostředí v aktuálním stavu. Volá se v debug režimu v každém cyklu.

```
%get_beliefs_from_enviroment(+Enviroment,--ListBeliefs)
% - Parametr Enviroment je vámi definované prostředí.
% - Parametr ListBeliefs je seznam udávající nové představy vycházející z prostředí.
get_beliefs_from_enviroment(Enviroment, ListBeliefs)
```

Touto funkcí se definuje, zda na základě změn v prostředí vznikají nové představy agenta. Nové představy se zde mohou přidat (pomocí funkce Prologu `assert`) nebo odebrat (pomocí funkce Prologu `retract`). Lze také aby na základě těchto akcí, vznikla nová událost. Stačí je přidat do proměnné `ListBeliefs`. Nový prvek seznamu získáte pomocí funkce `new_event(+O,+B,--E)`, kde `O` je '+' při přidání představy nebo '-' při odebrání, `B` je představa a `E` je nová událost. Seznam `ListBeliefs` musí mít stejnou délku jako je počet agentů v systému a pro každého agenta je pak definován vlastní seznam událostí.

Pokud v běhu některého agenta využíváte vámi nadefinované operace, je nutné je také mít ve vstupním souboru. Dále je nutné přidat dva parametry k fumkci, pokud se interpret spouští v režimu, kdy pracuje s prostředím. Tyto poslední dva parametry dané funkce budou právě pro práci s prostředím. Předposlední parametr bude tedy sloužit jako vstup původního prostředí a poslední jako výstup nově upraveného prostředí. Pokud se prostředí nemění, tak se pouze převede vstupní proměnná i na výstup.

V operacích, které si sami definujete, můžete využívat veškerých výhod programování v Prologu a také všech jeho vestavěných funkcí.

Příklad operace bez prostředí:

```
operation_plus(X,Y,Z):- Z~is Y + Y.
```

Příklad operace s prostředím:

```
operation_plus(X,Y,Z,E,E2):- Z~is Y + Y, E2 = append(E,[Z],E2).
```

Kapitola 8

Testování

Testování interpretu proběhlo pod systémem 64-bit Ubuntu 16.04 LTS. Hardwarová konfigirace notebooku použitého pro testování byla 8GB operační paměti a procesor Intel® Core™ i7-4700MQ CPU @ 2.40GHz × 8.

Funkcionalita interpretu byla realizována na ilustračních příkladech. V této kapitole tedy vždy uvedu příklad na jehož základě posoudím funkčnost interpretu. Všechny zde uvedené příklady jsou dodány jako příklady k implementaci interpretu.

8.1 Příklad - Prodej karet

Příklad lze nalézt na přiloženém médiu ve složce `DP/examples/sale_of_cards/`.

Tento příklad již byl popsán v kapitole 6.1. Agent Adam se snaží vydělat prodejem karet. Zde uvedu celý krátký program v jazyce AgentSpeak(L), který tento problém simuluje:

```
%beliefs
card(adam,card1,3).
card(adam,card2,1).
card(adam,card3,2).
card(adam,card4,3).
card(adam,card6,4).
bettyWants(card4,500).
bettyWants(card2,300).
bettyWants(card5,100).
bettyWants(card6,100).

%goal
!getMoney(Amount).

%plans
+!getMoney(Amount) <- !selectCard(C); !sellCards(C,Amount).
+!selectCard(Card) <- reviseCollection(); ?cardToSell(Card).
+!sellCards(Card,Amount) <- dealWith(betty, Card); ?bettyWants(Card, Amount).
```

Základní cíl agenta je získat peníze za karty. Tuto snahu realizuje plánem, kdy nejprve vybere kartu k prodeji a až následně se ji pokusí prodat. Kartu ovšem může prodat jen v případě, že ji Adam má více než jednou a Betty ji chce koupit. V našem případě Adam

vlastní pět karet a z toho čtyři více než jednou (card1, card3, card4, card6) a Betty je ochotna si koupit čtyři karty (card2, card4, card5, card6). Seznam vhodných karet k prodeji u Adama a Betty se schoduje pouze u dvou karet a to karet card4 a card6.

Pokud interpret spustíme bez FRAg režimu, Adam kartu neprodá. Je zde totiž problém v chování klasického interpretu. Ten bere karty z databáze, tak jak jdou za sebou. Proto vezme první kartu k prodeji, kterou Adam vlastní více jak jednou, což je v našem případě karta card1. Tuto kartu se následně pokusí prodat Betty, ale ta o kartu nemá zájem. Interpret proto prohlásí plán za nesplněný a cíl nebude splněn. Výsledek interpretace můžete vidět na obrázku 8.1.

Pokud ovšem spustíme interpret v režimu FRAg, výsledek bude zcela jiný. I když interpret z počátku pracuje stejně, při neúspěšném prodeji se navrací k výběru karet a volí jinou kartu a tak pokračuje dokud se mu kartu nepodaří prodat a nebo nevyčerpá všechny možnosti. Dokonce, i když se mu podaří splnit cíl, pokračuje dál a nabídne veškeré možné scénáře prodeje. Výsledek interpretace je na obrázku 8.2.

```
eva ~/diplomka2020/examples/sale_of_cards> ./run_nofrag
| 00:43::3.033441544 | INFO  | Card for sell: card1
| 00:43::3.033551455 | INFO  | Card for sell: card3
| 00:43::3.033582211 | INFO  | Card for sell: card4
| 00:43::3.033608913 | INFO  | Card for sell: card6
| 00:43::3.033881664 | INFO  | Plan not found
```

Obrázek 8.1: Interpretace příkladu prodej karet bez FRAg režimu.

```
eva ~/diplomka2020/examples/sale_of_cards> ./run_frag
| 00:43::7.128056526 | INFO  | Card for sell: card1
| 00:43::7.128139973 | INFO  | Card for sell: card3
| 00:43::7.128162622 | INFO  | Card for sell: card4
| 00:43::7.128181696 | INFO  | Card for sell: card6
| 00:43::7.128672361 | INFO  | List of plans: [[(!,getMoney(500)),(!,selectCard(
card4)),(op,reviseCollection()),((?),cardToSell(card4)),(!,sellCards(card4,500))
,(op,dealWith(betty,card4)),((?),bettyWants(card4,500)),[]]]
| 00:43::7.128888369 | INFO  | List of plans: [[(!,getMoney(100)),(!,selectCard(
card6)),(op,reviseCollection()),((?),cardToSell(card6)),(!,sellCards(card6,100))
,(op,dealWith(betty,card6)),((?),bettyWants(card6,100)),[]]]
```

Obrázek 8.2: Interpretace příkladu prodej karet ve FRAg režimu.

Pokud tedy použijeme FRAg. nalezneme hned dvě řešení daného problému. Kdežto použitím klasického způsobu, který je stejný jako v interpretu Jason nenalezneme řešení žádné.

8.2 Příklad - Hledání cesty čtvercovým polem

Příklad lze nalézt na přiloženém médiu ve složce `DP/examples/ways/`.

Další ilustrační příklad je hledání cesty ve čtvercovém poli. Úloha spočívá v nalezení možné cesty čtvercovým polem o straně velikosti n polí, kdy agent v počátečním stavu začíná v levém horním rohu se souřadnicemi $(1, 1)$ a jeho cíl je dostat se do pravého dolního rohu se souřadnicemi $(n.n)$. Má ovšem v každém okamžiku na výběr jen ze dvou směrů pohybu. Buď provede pohyb směrem dolů a nebo směrem vpravo.

Samotný program v jazyku AgentSpeak(L) pro `n=10` vypadá následně:

```
%beliefs
len(10).
direction(down).
direction(right).
home(L,L) :- len(L).
start(1,1).

%goal
!way_home().

%plans
+!way_home() <- ?start(X,Y); !go(X,Y).
+!go(X,Y) : home(X,Y).
+!go(X,Y) <- ?direction(D); step(D,X,Y,X2,Y2); !go(X2,Y2).
```

Opět je chování interpretu v režimu FRAg a mimo něj je odlišné. Pokud, režim FRAg není aktivní, interpret vždy substituuje směr, kterým se má agent v daném kroku pohnout, za první z databáze, což je v tomto případě down, čili dolů. Agent tedy v poli o velikosti n dospěje $n-1$ posuny dolů na pozici $(1, n)$ a pokusí se znovu o posun dolů, ale díky hranici pole operace selže a agent opět nenalezne ani jedno řešení.

Pokud použijeme systém FRAg, agent vždy provede substituci znovu, a to při úspěšném i neúspěšném projití polem, Nakonec vypíše všechny možné varianty. Pro představu na obrázku 8.3 je vyobrazena jedna cesta, kterou interpret vypsal.

```
| 21:51::11.93471193 | INFO  | List of plans: [[(!,way_home()),((?),start(1,1)),
(!,go(1,1)),((?),direction(right)),(op,step(right,1,1,2,1)),(!,go(2,1)),((?),dir
ection(right)),(op,step(right,2,1,3,1)),(!,go(3,1)),((?),direction(right)),(op,s
tep(right,3,1,4,1)),(!,go(4,1)),((?),direction(right)),(op,step(right,4,1,5,1)),
(!,go(5,1)),((?),direction(right)),(op,step(right,5,1,6,1)),(!,go(6,1)),((?),dir
ection(right)),(op,step(right,6,1,7,1)),(!,go(7,1)),((?),direction(right)),(op,s
tep(right,7,1,8,1)),(!,go(8,1)),((?),direction(right)),(op,step(right,8,1,9,1)),
(!,go(9,1)),((?),direction(right)),(op,step(right,9,1,10,1)),(!,go(10,1)),((?),d
irection(down)),(op,step(down,10,1,10,2)),(!,go(10,2)),((?),direction(down)),(op
,step(down,10,2,10,3)),(!,go(10,3)),((?),direction(down)),(op,step(down,10,3,10,
4)),(!,go(10,4)),((?),direction(down)),(op,step(down,10,4,10,5)),(!,go(10,5)),((
?),direction(down)),(op,step(down,10,5,10,6)),(!,go(10,6)),((?),direction(down))
,(op,step(down,10,6,10,7)),(!,go(10,7)),((?),direction(down)),(op,step(down,10,7
,10,8)),(!,go(10,8)),((?),direction(down)),(op,step(down,10,8,10,9)),(!,go(10,9)
),((?),direction(down)),(op,step(down,10,9,10,10)),(!,go(10,10)),[]]]
```

Obrázek 8.3: Interpretace příkladu cesta čtvercovým polem ve FRAg režimu. Ukázka jedné nalezené cesty polem $n = 10$.

Počet cest ve čtvercovém poli vypočítáme následovně:

$$\frac{(2(n-1))!}{((n-1)!)^2}$$

Pokud tedy použijeme pole o straně deset, měli bychom dostat 48620 možných průchodů polem. O tom, že přesně tolik cest interpret nalezl, se můžete přesvědčit na obrázku 8.4.

I pro tento případ je tedy systém s FRAg účinějším nástrojem.

```
eva ~/diplomka2020/examples/ways> w=$(../../interpreter.pl -p ways.pl -p ways_op
er.pl -f | wc -l); echo found $w plans
found 48620 plans
eva ~/diplomka2020/examples/ways>
```

Obrázek 8.4: Interpretace příkladu cesta čtvercovým polem ve FRAg režimu. Ukázka nalezení všech cest polem $n = 10$.

8.3 Příklad - Nákup dárku na výletě

Příklad lze nalézt na přiloženém médiu ve složce `DP/examples/castle1/`.

Tento příklad byl již také uveden v kapitole 6.1. Pro tuto ukázku byl ovšem zjednodušen. Agent zde plní dva cíle. Jeden cíl je vypravit se na hrad a druhý koupit dárek.

Nejprve uvedu program:

```
presentSells(supermarket, book).
presentSells(pernstejn, figure).
presentSells(pernstejn, thimble).
presentSells(bouzov, postcard).
castle(pernstejn).
castle(bouzov).

!buyPresent().
!makeTrip().

+!goTo(X).
+!buyPresent() <- ?presentSells(X,_); !goTo(X).
+!makeTrip() <- ?castle(X); !goTo(X).
```

Chování obou srovnávaných přístupů je opět odlišné. Pokud použijeme systém bez FRAg, bude v tomto případě nalezena jedna cesta. Konkrétně bude dárek zakoupen v supermarketu a na výlet půjde agent na Perštejn a to díky tomu, že jsou tyto místa opět první v seznamu. Ukázka činnosti interpreta je na obrázku 8.5.

Pokud je systém zpracováván systémem FRAg, budou nalezeny veškeré možné realizace plánů. Ukázku některých výsledků můžete vidět na obrázku 8.6. Pro nalezení ideálních plánů, což jsou plány, kde jdeme na výlet a kupujeme dárek na stejném místě, by jsme museli výsledky přetřídit. I když bychom vzali jen místa, kde je splněna tato podmínka, nedostali by jsme jen tři plány, které bychom čekali, ale plánů více, lišící se pořadím prováděných kroků v plnění dvou plánů najednou. Řešení, které nalezl interpret bez podpory FRAg, toto podmínku jak je patrné nesplňuje.

```
eva ~/diplomka2020/examples/castle1> ./run_nofrag
| 04:29::45.16149282 | INFO  | List of plans: [[(!,buyPresent()),(!,makeTrip()),
((?),presentSells(supermarket,book)),(!,goTo(supermarket)),[],((?),castle(pernst
ejn)),(!,goTo(pernstejn)),[]]]
```

Obrázek 8.5: Interpretace příkladu nákup dárku na výletě bez FRAg.

```
| 03:17::24.78184938 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(perns
tejn)),(!,goTo(pernstejn)),[],[],(!,buyPresent()),((?),presentSells(supermarket,
book)),(!,goTo(supermarket)),(op,printPos(supermarket)),[]]]
| 03:17::24.78199363 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(perns
tejn)),(!,goTo(pernstejn)),[],[],(!,buyPresent()),((?),presentSells(pernstejn,fi
gure)),(!,goTo(pernstejn)),(op,printPos(pernstejn)),[]]]
| 03:17::24.78213549 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(perns
tejn)),(!,goTo(pernstejn)),[],[],(!,buyPresent()),((?),presentSells(pernstejn,th
imble)),(!,goTo(pernstejn)),(op,printPos(pernstejn)),[]]]
| 03:17::24.78228354 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(perns
tejn)),(!,goTo(pernstejn)),[],[],(!,buyPresent()),((?),presentSells(bouzov,postc
ard)),(!,goTo(bouzov)),(op,printPos(bouzov)),[]]]
| 03:17::24.78259706 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(bouzo
v)),(!,goTo(bouzov)),(op,printPos(bouzov)),[],(!,buyPresent()),((?),presentSells
(supermarket,book)),(!,goTo(supermarket)),(op,printPos(supermarket)),[]]]
| 03:17::24.78274584 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(bouzo
v)),(!,goTo(bouzov)),(op,printPos(bouzov)),[],(!,buyPresent()),((?),presentSells
(pernstejn,figure)),(!,goTo(pernstejn)),(op,printPos(pernstejn)),[]]]
| 03:17::24.78290296 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(bouzo
v)),(!,goTo(bouzov)),(op,printPos(bouzov)),[],(!,buyPresent()),((?),presentSells
(pernstejn,thimble)),(!,goTo(pernstejn)),(op,printPos(pernstejn)),[]]]
| 03:17::24.78306890 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(bouzo
v)),(!,goTo(bouzov)),(op,printPos(bouzov)),[],(!,buyPresent()),((?),presentSells
(bouzov,postcard)),(!,goTo(bouzov)),(op,printPos(bouzov)),[]]]
| 03:17::24.78332877 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(bouzo
v)),(!,goTo(bouzov)),[],[],(!,buyPresent()),((?),presentSells(supermarket,book))
,(!,goTo(supermarket)),(op,printPos(supermarket)),[]]]
| 03:17::24.78347063 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(bouzo
v)),(!,goTo(bouzov)),[],[],(!,buyPresent()),((?),presentSells(pernstejn,figure))
,(!,goTo(pernstejn)),(op,printPos(pernstejn)),[]]]
| 03:17::24.78363967 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(bouzo
v)),(!,goTo(bouzov)),[],[],(!,buyPresent()),((?),presentSells(pernstejn,thimble)
),(!,goTo(pernstejn)),(op,printPos(pernstejn)),[]]]
| 03:17::24.78378201 | INFO  | List of plans: [[(!,makeTrip()),((?),castle(bouzo
v)),(!,goTo(bouzov)),[],[],(!,buyPresent()),((?),presentSells(bouzov,postcard)),
(!,goTo(bouzov)),(op,printPos(bouzov)),[]]]
```

Obrázek 8.6: Interpretace příkladu nákup dárku na výletě ve FRAg režimu. Ukázka pár nalezených plánů.

8.4 Příklad - Čistící roboti na Marsu

Příklad lze nalézt na přiloženém médiu ve složce `DP/examples/cleaning_robots/`.

Poslední příklad jsem si vypůjčil z příkladů interpretu Jason. Jedná se o příklad s prostředím. Jelikož je jeho jednomu agentovi.chování komplikovanější a program má větší rozsah, uvedu zde jen základní princip. Máme dva agenty. První je robot průzkumník, který se pohybuje po ploše Marsu a pokud narazí na odpad, uchopí jej a donese do středu plochy, kde jej druhý robot spálí. První agent se poté vrátí na místo, odkud odebral odpad a pokračuje v prohledávání. Příklad přesně kopíruje předlohu a nemá přirozené ukončení. Proto je vhodné jej spouštět v debug módu. Tento příklad uvádím jako ilustraci toho, že interpret zvládá bez problému i práci s prostředím a obecně náročnější systémy. Ukázka výstupu příkladu o čistících robotech je na obrázku 8.7. Výpis prostředí je pouze znakový a je implementován a jazyce Prolog, ale pro ilustrační účely stačí. Jde i o pěknou ukázku programu v debug módu, kdy je možné procházet běh agentního systému krok za krokem. Symbolem X je reprezentován první agent průzkumník a symbolem 0 druhý agent, který odpad pálí. Číslicemi je pak uvedeno kolik odpadu je aktuálně v daném poli. Interpret také vypisuje aktuálně prováděnou akci a seznam představ jednotlivých agentů.

8.5 Nalezené nedostatky interpreta

Při testování interpretu jsem narazil na jisté nedostatky interpreta. Některé funkcionality, které jazyk AgentSpeak(L) a jeho interpret Jason dokáže vykonávat, nejsou u mého interpreta možné. Byla by nutné je implementovat.

Jedním z těchto funkci je komunikace mezi agenty. AgentSpeak(L) v podobě pro Jason obsahuje internl funkce, z nichž jedna dokáže zprostředkovávat komunikaci mezi agenty. Konkrétně realizuje přidávání či odebírání představ ostatním agentům a také se na tyto představy doptávat. Mimo to dokáže klást jinému agentovi cíle a podobně. Tato funkcionalita u mého interpreta chybí a dělá jej omezenější. Bylo by teoreticky možné sestavit operaci, která by pracovala s představy ostatních agentů a to jako externí operaci v Prologu, která se dá následně zahrnout do plánu. S cíli ovšem takto pracovat nelze, protože jsou součástí agentova kontextu a nejsou uloženy v interní databázi interpreta Prologu. Interní operace tu chybí obecně. Většina lze nahradit operacemi externími, které si uživatel naprogramuje.

Dále jsem objevil chybu v návrhu systému, která se projeví v agentním systému, který je spuštěn v režimu FRAg a pracuje aktivně s představami. Konkrétně je ne jen čte, ale také přidává a odebírá. Při zpětném navracení se sice vrací celý kontext agenta, který obsahuje jednotlivé cíle a plány, ale ne představy. Ty jsou uloženy v interní databázi interpreta Prologu a při zpětném navrácení zůstávají stejné. Toto je vážný nedostatek interpreta a bylo by vhodné představy agenta také zahrnout do kontextu agenta, který si nese v proměnné základním cyklem interpretace.

```
| 04:43::16.69620895 | DEBUG | ------------------------------------------------
------------------------------------------------
| 04:43::16.69624591 | DEBUG | Cyklus: 1
| 04:43::16.69626236 | DEBUG | Actual plans list: [[([[(!,check(slots))]],[])],[
]]
| 04:43::16.69644070 | DEBUG | after get bel.
| 04:43::16.69645882 | DEBUG | Actual belief:
| 04:43::16.69647169 | DEBUG | ------------------------------
| 04:43::16.69648600 | DEBUG | bel(at(r1), r1, _9476)
| 04:43::16.69650507 | DEBUG | bel(pos(r1,1,1), r1, percept)
| 04:43::16.69652176 | DEBUG | bel(pos(r1,1,1), r2, percept)
| 04:43::16.69654536 | DEBUG | bel(pos(r2,4,4), r1, percept)
| 04:43::16.69656134 | DEBUG | bel(pos(r2,4,4), r2, percept)
| 04:43::16.69657707 | DEBUG | ------------------------------
| 04:43::16.69658971 | DEBUG | after print all bel
| 04:43::16.69660282 | DEBUG | New beliefs from environment
| 04:43::16.69661593 | DEBUG | Agent: r1
| 04:43::16.69662881 | DEBUG | No new beliefs
| 04:43::16.69664216 | DEBUG | New beliefs from environment
| 04:43::16.69665527 | DEBUG | Agent: r2
| 04:43::16.69666767 | DEBUG | No new beliefs
| 04:43::16.69668031 | DEBUG | afretr write bel.
| 04:43::16.69669390 | DEBUG | add new bwl.
| 04:43::16.69670701 | DEBUG | Begin select plan.
| 04:43::16.69678569 | DEBUG | After select plan.
| 04:43::16.69680834 | DEBUG | ************************************
| 04:43::16.69682169 | DEBUG | 0: !,check(slots)
| 04:43::16.69683552 | DEBUG | ************************************
| 04:43::16.69689107 | DEBUG | Print enviroment:
-------------------------------------------------------
|(X, , )|( , , )|( , , )|( , ,1)|( , , )|( , , )|( , ,1)|
-------------------------------------------------------
|( , , )|( , , )|( , , )|( , , )|( , , )|( , , )|( , , )|
-------------------------------------------------------
|( , , )|( , ,1)|( , , )|( , , )|( , , )|( , , )|( , , )|
-------------------------------------------------------
|( , , )|( , , )|( , , )|( ,0, )|( , , )|( , , )|( , , )|
-------------------------------------------------------
|( , , )|( , , )|( , , )|( , , )|( , , )|( , , )|( , , )|
-------------------------------------------------------
|( , ,1)|( , , )|( , , )|( , , )|( , , )|( , , )|( , , )|
-------------------------------------------------------
|( , , )|( , , )|( , , )|( , , )|( , , )|( , , )|( , ,1)|
-------------------------------------------------------

|:
```

Obrázek 8.7: Interpretace příkladu čistící roboti na Marsu.

Kapitola 9

Diskuze

V této kapitole popíši rozdíly mezi interpretací agentního systému řízeného záměrem s použitím FRAg režimu a bez něj. Vycházet budu především z předchozí kapitoly 8, kde jsme testovali chování systému FRAg na různých příkladech. Dále zhodnotím stav interpreta a jeho použitelnost.

Na prvním příkladu 8.1, kdy jsme hledali vhodné karty k prodeji, jsem mohli vidět, že interpret založený na FRAg systému byl mnohem vhodnějším k dosažení cíle. Na rozdíl od běžného přístupu nalezl vhodné řešení. Dokonce na základě zvoleného cíle nalezl hned dvě řešení. To s klasickým přístupem není možné. FRAg systém je tedy vhodný pro příklady, kdy standardní interpret se stejným chováním jako je například interpret Jason neuspěje. Také je vhodný pokud chceme zjistit veškerá možná řešení.

Druhý příklad, kde agent procházel čtvercovým polem 8.2, je v zásadě podobný předchozímu příkladu. Interpret v režimu FRAg nalezl všechna možná řešení, čili všechny možné cesty agenta čtvercovým polem, zatím co bez něj opět nebyl úspěšný. Tento příklad byl sestavený, tak aby ukázal schopnost interpreta s FRAg procházet veškeré možné realizace plánu. Dokonce lze použít k zjištění počtu možných řešení daného problému. Dá se tedy použít k prohledávání stavového prostoru a získání všech možných řešení. Jistě by se tento systém dal použít k řešení podobných problémů jako je například známý problému osmi dam [22], kde je úkolem rozmístit n dam na šachovnici o rozměrech $n \times n$ tak, aby se vzájemně neohrožovaly.

Třetí příklad, kdy má agent stanovené dva cíle 8.3, nám názorně ukázal, že FRAg systém může vhodně sloužit i k hledání variací řešení se stejným výsledkem. V tomto případě by mohla být úloha lépe navržena. Kdyby například agent pracoval s aktuální pozicí na úrovni představ a plány byli jinak sestaveny, mohli bychom pomocí systému FRAg dosáhnout třech řešení, která se nabízí a FRAg systém by si poradil i s více cíli agenta. Ovšem příklad je navržený tak, aby nám ukázal i negativa systému FRAg. Může být složitější navrhnou agentní systém představující pouze ty plány, které opravdu vyžadujeme. Při plnění více cílů provádí agent interpretovaný systémem FRAg jednotlivé kroky v různém pořadí. Tedy i když by se agentovy podařilo koupit dárek na hradě, kde si zároveň udělá výlet, systém FRAg by vypsal i jednotlivé variace v závislosti na pořadí vykonávaných kroků plánu, což nemusí být vždy žádaný efekt. Jde tomu ovšem zabránit dobrým návrhem agentního systému. Pokud bychom například určili pořadí vykonávání cílů tím, že bychom vytvořili cíl nový a jako plán bychom mu dali vykonání těchto podcílů, můžeme opravdu získat pouze tři správné plány k splnění tohoto cíle. Tuto variantu řešení je možné nalézt na přiloženém médiu ve složce `DP/example/scastle2/`.

čtvrtý příklad slouží spíše k ukázce, co vše implementovaný interpret dokáže. Zároveň je to vhodná ukázka agetného systému, pro který interpretace pomocí FRAg není vhodná. Pokud tento systém spustíme v režimu FRAg, bude nacházet řešení, která nejsou žádoucí.

Můžeme tedy shrnout vlastnost interpretace systémem FARgu do těchto jednoduchých bodů:

- Interpret v režimu FRAg dokáže nalézt díky systému zpětného navracení řešení, i když při klasické interpretaci by se plán nalézt nepovedlo.
- Interpret v režimu FRAg dokáže nalézt všechny možné plány pro realizaci cílů agenta. Lze jej tak použít na úlohy prohledávání stavového prostoru.
- Pro agentní systém řízený pomocí FRAg systému může být náročnější sestavit plány tak, aby dával jen požadovaná řešení.
- Systém FRAg je vhodný jen pro některé typy úloh, konkretně pro ty, kde je žádoucí využít jeho speciálních vlastností. Naopak pro úlohy, kde chceme dojít k jednomu plánu i přes velké množství možných řešení je systém FRAg nevhodný.

Interpret, který byl interpretován v rámci této práce je funkčním řešením, které dokáže interpretovat jednoduché úkoly, má však i své nedostatky. Ty nepramení z přidaného systému FRAg, ale z návrhových a implementačních nedostatků. Ty jsou částečně popsány v kapitole 8.5.

Jedná se o absenci interních funkcí, které interpret Jason podporuje. Ty jdou do značné míry nahradit vlastní externí funkcí, kterou si uživatel sestaví sám, ale vyžaduje to schopnost programovat v jazyce Prolog.

Také chybí podpora komunikace mezi agenty, což do značné míry omezuje funkčnost, kdy nebude možné interpretovat agentní systémy, které toto vyžadují.

Interpret umožňuje práci s prostředím a v debug režimu, va kterém jej dokáže vypsat. Definování struktury prostředí a její výpis je opět ovšem možný jen pokud jej uživatel sestaví v jazyce Prolog, což vyžaduje programátorské dovednosti. Interpret Jason ovšem v tomto ohledu není lepší, jen využívá pro vizualizaci prostředí jazyk Java.

Další nedostatek je způsoben chybou v návrhu. Pokud použijeme režim interpretace FRAg a pracujeme při tom aktivně s představami, a to tak, že je nejen čteme, ale i přidáváme a ubíráme, interpret není schopný při zpětném navracení vrátit původní stav představ. Toto je vážný nedostatek, který by bylo vhodné vyřešit jinou reprezentací představ agenta v interpretačním programu. Nutno podotknout, že je to pouze chyba návrhu a implementace a nevypovídá o chybě systému FRAg jako takového.

Kapitola 10

Závěr

V rámci této diplomové práce jsem se seznámil s problematikou interpretace agentně orientovaných systémů, řízených záměrem. Dále jsem se blíže seznámil s jazykem AgentSpeak(L) a systémem FRAg. Byl vytvořen interpret agentních systémů v jazyce Prolog, který dokáže interpretovat agentní systém v jazyce AgentSpeak(L). Tento interpret má podporu systémů s prostředím, dokáže fungovat v debug režimu, kdy jde realizovat daný systém krok po kroku a sledovat aktuální stav agenta a prostředí a podporuje systém FRAg. Dokáže realizovat atomické plány. Tento interpret byl zasazen do jednoduchého vývojového prostředí. V práci je uvedeno několik příkladů pro ověření funkce interpreta. Tyto příklady potvrzují funkčnost vyvinutého interpreta včetně přínosu FRAg systému. Ukázky jsou k nalezení na přiloženém médiu a jejich výstupy jsou jsou dostupny pro různé vstupní nastavení v předešlých kapitolách. Také byla sestavena dokumentace, která prezentuje, jak má uživatel s interpretem zacházet.

Interpret, implementovaný v rámci této diplomové práce je komplexní nástroj na interpretaci agentních a multiagentních systémů řízených záměrem. Splňuje veškeré požadavky dané zadáním. Navíc má velkou podporu v samotném Prologu. Uživatel může využít všech jeho knihoven a použít je v činnosti agentů. Má ovšem i své nedostatky. Na rozdíl od interpretu Jason je mnohem menšího rozsahu. Chybí podpora komunikace mezi agenty. A v režimu FRAg je problém s nekonzistencí představy agenta a jeho plány při zpětném navracení. Jsou to podněty pro další práci a vývoj na interpretu.

10.1 Další vývoj

Interpret je možné dále vyvíjet a doplňovat o nové funkcionality. Neustále se dá rozšiřovat portfólio vestavěných operací, Jako hlavní bod dalšího vývoje bych zařadil převedení představy agenta z interní databáze Prologu do proměnné. Tento nedostatek brání správné interpretaci pokud je agentní systém interpretován v režimu FRAg a pracuje aktivně s představami. Bylo by vhodné přidat podporu komunikace mezi agenty.

Interpret má nyní velmi jednoduché vývojové prostředí, pro zlepšení komfortu při vývoji agentních systémů by bylo přínosem mít nějaké atraktivnější řešení.

Literatura

[1] Apt, K. R. et al. *From logic programming to Prolog.* Prentice Hall London, 1997.

[2] Bordini, R. H. a Hübner, J. F. BDI agent programming in AgentSpeak using Jason. In: Springer. *International Workshop on Computational Logic in Multi-Agent Systems.* 2005, s. 143–164.

[3] Bordini, R. H. a Hübner, J. F. A Java-based interpreter for an extended version of AgentSpeak. *University of Durham, Universidade Regional de Blumenau.* 2007.

[4] Bordini, R. H., Hübner, J. F. a Wooldridge, M. *Programming multi-agent systems in AgentSpeak using Jason.* John Wiley & Sons, 2007.

[5] Bordini, R. H., Hübner, J. F. a Wooldridge, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology).* USA: John Wiley & Sons, Inc., 2007. ISBN 0470029005.

[6] Bordini, R. H. a Moreira, A. F. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak (L). *Annals of Mathematics and Artificial Intelligence.* Springer. 2004, sv. 42, 1-3, s. 197–226.

[7] Bratko, I. *Prolog programming for artificial intelligence.* Pearson education, 2001.

[8] Bratman, M. et al. *Intention, plans, and practical reason.* Harvard University Press Cambridge, MA, 1987.

[9] Brown, P. J. Writing interactive compilers and interpreters. *Wiley Series in Computing, Chichester: Wiley, 1979.* 1979.

[10] Clocksin, W. F. a Mellish, C. S. *Programming in Prolog: Using the ISO standard.* Springer Science & Business Media, 2012.

[11] Covington, M. A., Grosz, B. J. a Pereira, F. C. *Natural language processing for Prolog programmers.* Prentice hall Englewood Cliffs (NJ), 1994.

[12] Covington, M. A., Nute, D. a Vellino, A. e. ISO Prolog. 1993.

[13] Dastani, M. 2APL: a practical agent programming language. *Autonomous agents and multi-agent systems.* Springer. 2008, sv. 16, č. 3, s. 214–248.

[14] Horn, A. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic.* Cambridge University Press. 1951, sv. 16, č. 1, s. 14–21. DOI: 10.2307/2268661.

[15] Huhns, M. N., Singh, M. P., Burstein, M., Decker, K., Durfee, K. E. et al. Research directions for service-oriented multiagent systems. *IEEE Internet Computing*. Nov 2005, sv. 9, č. 6, s. 65–70. DOI: 10.1109/MIC.2005.132. ISSN 1089-7801.

[16] HÁJKOVÁ, V. *BDI Agenti*. 2012. Referát. Brno: Masarykova univerzita, Filosofická fakulta. Dostupné z: https://nlp.fi.muni.cz/uui/referaty2012/veronika_hajkova/referat.pdf.

[17] Jennings, N. R. Specification and implementation of a belief-desire-joint-intention architecture for collaborative problem solving. *International Journal of Intelligent and Cooperative Information Systems*. World Scientific. 1993, sv. 2, č. 03, s. 289–318.

[18] Král, J., Zbořil, F. a Zbořil, V. F. Flexible Plan Handling using Extended Environment. In: *Proceedings of the 12th International Conference on Informatics*. Faculty of Electrical Engineering and Informatics, University of Technology Košice, 2013, s. 228–233. ISBN 978-80-8143-127-2. Dostupné z: http://www.fit.vutbr.cz/research/view_pub.php?id=9991.

[19] McCarthy, J. *Programs with common sense*. RLE and MIT computation center, 1960.

[20] Moreira, Á. F., Vieira, R. a Bordini, R. H. Extending the Operational Semantics of a BDI Agent-Oriented Programming Language for Introducing Speech-Act Based Communication. In: Leite, J., Omicini, A., Sterling, L. a Torroni, P., ed. *Declarative Agent Languages and Technologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, s. 135–154. ISBN 978-3-540-25932-9.

[21] Rao, A. S. AgentSpeak (L): BDI agents speak out in a logical computable language. In: Springer. *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*. 1996, s. 42–55.

[22] Sloane, N. Number of Ways of Placing n Nonattacking Queens on n× n Board, In the On-Line Encyclopedia of Integer Sequences (OEIS). *Http://www. research. break_att. com/˜ njas/sequences/A000170*. 2008.

[23] Szabó, P. a Szeredi, P. Improving the ISO Prolog Standard by Analyzing Compliance Test Results. In: Etalle, S. a Truszczyński, M., ed. *Logic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, s. 257–269. ISBN 978-3-540-36636-2.

[24] Wielemaker, J. Native Preemptive Threads in SWI-Prolog. In: Palamidessi, C., ed. *Logic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, s. 331–345. ISBN 978-3-540-24599-5.

[25] WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M. a LAGER, T. SWI-Prolog. *Theory and Practice of Logic Programming*. Cambridge University Press. 2012, sv. 12, 1-2, s. 67–96. DOI: 10.1017/S1471068411000494.

[26] Williams, M., Chen, G., Ferbrache, D., Massey, P., Salvini, S. et al. Prolog and deductive databases. *Knowledge-Based Systems*. 1988, sv. 1, č. 3, s. 188 – 192. DOI: https://doi.org/10.1016/0950-7051(88)90077-9. ISSN 0950-7051. Dostupné z: http://www.sciencedirect.com/science/article/pii/0950705188900779.

[27] Zbořil, V. F. *Plánování a komunikace v multiagentních systémech (Disertační práce)*. Disertační práce. Dostupné z: http://www.fit.vutbr.cz/~zborilf/PhD/thesis.pdf.

[28] Zbořil, F., Kočí, R., Janoušek, V. a Mazal, Z. Reactive Planning with Weak Plan Instances. In: *Proceedings of 8th ISDA* [print]. NEUVEDEN. IEEE Computer Society: IEEE Computer Society, December 2008, kap. 32322, s. 643–648.

Příloha A

Obsah přiloženého paměťového média

```
interpreter.pl                - interpret
doc/doc.pdf                   - dokumentace
examples/card2/card2.pl       - příklad prodej karet
examples/cards/cards.pl       - příklad prodej karet 2
examples/castle1/castle1.pl   - příklad koupení dárku na hradě
examples/castle2/castle2.pl   - příklad koupení dárku na hradě 2
examples/cleaning_robots/cleaning_robots.pl  - příklad uklízení
                                roboti na Marsu
examples/ways/ways.pl         - příklad projití čtvercovým polem
```