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 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

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)

Při obhajobě semestrální části projektu je požadováno:

- První bod zadání, aspoň částečně druhý bod zadání

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Zbořil František, doc. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Tato páce se zabývá realizací iterpretu agentního systému řízeného záměrem implemetovaného v jazyce PROLOG. Jako vzor byl využit interpret Jason implementovaný v jazyce Java, který interpretuje jazyk AgentSpeak(L). Pro účely tohoto projektu je program v AgentSpeak(L) nejprve převeden na interní formu jazyka. Pro demonstraci byla použita jedna z úloh přiložených k programu Jason, jako příklad a to konkrétně tzv. "cleaning robors". Interpret dokáže interpretovat systím jako FRAg a dokáže reagovat naprostředí.

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). For the purposes of this project, the program in AgentSpeak(L) is first converted to an internal language form. For the demonstration, one of the examples included in the Jason program, specifically "cleaning robors", was used. The interpreter can interpret the system as a FRAg and can react in the enviroment.

Klíčová slova

BDIAgent, Jason, PROLOG, AgentSpeak(L), interpret, FRAg, agentní systém

Keywords

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

Citace

NĚMEC, Ladislav. *Interpretace agentního systému řízeného záměrem v jazyce PROLOG.* Brno, 2019. 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 tento semestrální projekt vypracoval samostatně pod vedením pana Doc. Ing. Zbořila Františka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ladislav Němec
22. května 2019

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[10].

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. V kapitole 2 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 3. Zabývá se především sémantickou a syntaktickou stránkou tohoto jazyka. Také je zde představen model interpretace prostřednictvím interpretu Jason. Kapitola 4 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 5 se seznámíme se systémem FRAg. Dozvíme se jeho výhody a jakým způsobem rozšiřují klasickou interpretaci agentních systémů. Kapitola 6 hovoří o mém přínosu v rámci této práce. Představuje návrh modelu mého interpreta a detailně se věnuje jeho implementaci. Nakonec posoudíme chování interpretu v kapitole 7, která je věnována testování.

Kapitola 2

Agentní systémy řízené záměrem

V této kapitole se budeme věnovat agentními systémy řízenými 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 [11].

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:

- 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.

2.1 BDI architektura

BDI architektura je přístup k tvorbě agentních a multiagentních systémů. BDI agent je zvláštní druh deliberativního či racionálního agenta, jehož jednání vychází z třech konkrétních mentálních stavů, jajich první písmena anglické podoby tvoří zkratky BDI [11]:

- **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.

2.2 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 dosáhnutí svých nově aktualizovaných záměrů slouží funkce $plan()$.

Jeden cyklus končí tehdy, je-li seznam kroků plánu prázdný (funkce $empty()$), 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 3

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 [1, 5]. 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 [14]. Primárně nás bude zajímat syntaxe a sémantika tohoto jazyka. Tato kapitola čerpá z těchto zdrojů[2, 4].

Syntaxe jazyka vychází z níže uvedené BNF gramatiky (viz 3.1), 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.

3.1 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.

3.1.1 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 3.1: BNF gramatika AgentSpeak(L)[2].

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

3.1.2 Pravidla

Jednotlivé představy mohou být vyjádřeny také pravidly. Nejlépe bude si ukázat funkčnost pravidel 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.

3.2 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.

3.3 Plány

Plány v AgentSpeaku slouží k dosažení jednotlivých cílů agenta. Konkrétní plán má následující obecnou strukturu:

spouštěcí událost : kontext <- tělo.

Tyto jednotlivé části si popíšeme v následujících podkapitolách.

3.3.1 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 naleznete v tabulce 3.2.

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 3.2: Tabulka typů spouštěcích událostí

3.3.2 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 3.3. 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 3.3: Tabulka typů negací v platformě Jason

3.3.3 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 ".". Zde jsou 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].

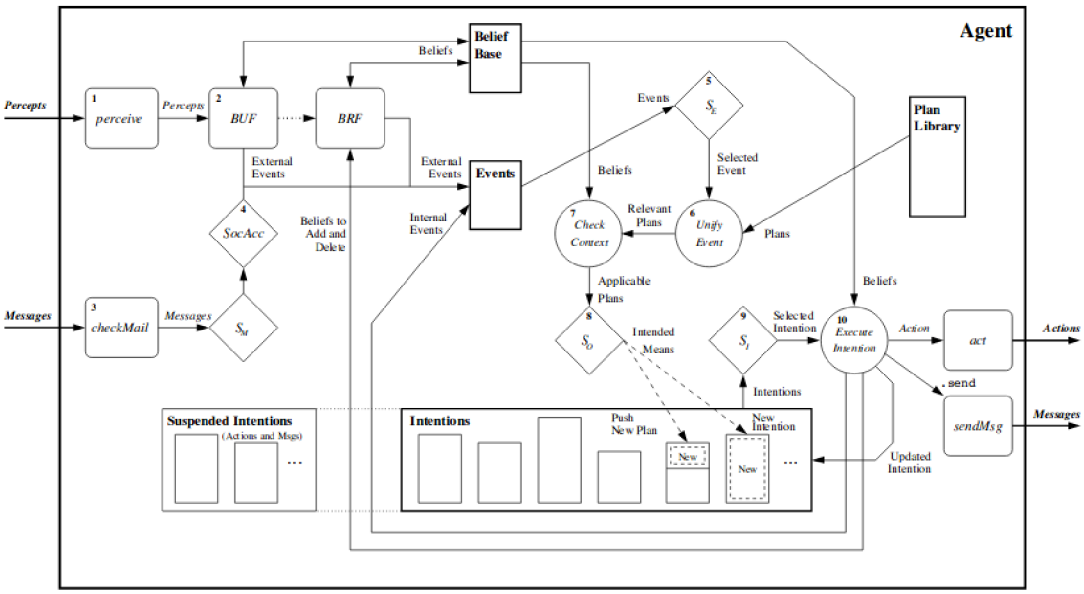
- **Dosahovaný cíl** - Pokud je nadefinovaná operace pro přidání tohoto cíle, je přidán do seznamu cílů a musí být splněn pro dokončení cíle aktuálního. Před literálem je znak "!". Například: *!arrest(burglar);*.
- **Testovací cíl** - Jak již bylo zmíněno výše, slouží k získání informace z představy 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. $X > 5;$.

3.4 Interpretační cyklus agenta

Na obrázku 3.1 je uvedený 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 3.1: Interpretační cyklus AgentSpeak(L) [1]

Kapitola 4

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* [6] a *ISO Prolog z roku 1993* [7].

4.1 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.

Programovací jazyk Prolog patří mezi tzv. neprocedurální programovací jazyky. To jednoduše řečeno znamená, že je založený na myšlence programování aplikací pomocí definic toho, co se má udělat a to jak se to má udělat, je ponecháno na interpretu jazyka. Prolog patří mezi takzvané logické programovací jazyky, tuto kategorii však v jistých ohledech přesahuje. Logické jazyky v širším významu využívají matematické logiky jako prostředku pro programování.

4.1.1 Syntaxe

Syntaxe Prologu vychází z predikátové logiky, konkrétně z Hornových klauzulí[9]. Jedná se o speciální druh klauzule, která obsahuje nejvýše jeden pozitivní literál. Lze ji obecně zapsat jako implikaci ve formě:

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

V Prologu pak klauzule vypadá následně:

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

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

4.1.2 Datové typy

V jazyce Prolog rozlišujeme několik datových typů:

- **Atomy** - Můžeme je také nazvat konstanty, jsou buď celá čísla (řada implementací pracuje již i s reálnými čísly), 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 zde není vázaná na konkrétní hodnotu, zatím co vázaná je. 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].

4.1.3 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]*

4.1.4 SDL rezoluce

Pro vyhodnocování dotazů v Prologu se užívá tzv. SDL rezoluce. 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 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.

4.1.5 Důležité operace Prologu

Jazyk Prolog nabízí řadu možností. Již podle ISO Prolog 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.

4.1.6 Interpret SWI-Prolog

SWI-Prolog nabízí komplexní bezplatné prostředí Prologu. 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, to 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.

4.2 Srovnání Prologu a AgentSpeak(L)

Pokud si prohlédneme program napsaný v AgentSpeak(L) a Prologu, zjistíme, že jsou si nápadně podobné. Oba tyto jazyky jsou založené na logickém programování. Uvedu zde výčet jejich základních rozdílů:

- Můžeme vidět podobnost mezi plány u AgentSpeak(L) a klauzulemi u Prologu. U AgentSpeak(L) máme navíc kontext.
- S rozšířeními Jason je odděleno teoretické a praktické uvažování.
- BDI architektura umožňuje:
  - Dlouhodobé cíle.
  - Změny a reagování na dynamické prostředí.
  - Manipulace s více ohnisky pozornosti.
- AgentSpeak(L) má přímou integraci s Javou.

Kapitola 5

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 [15], dále byl představen v roce 2003 na konferenci v Košicích [12]. Jako takový interpretuje systém obsahující určité víry, plány, události a záměry. Cíle mohou být ve formě testů nebo dosahování cílů, jak je běžné v téměř každém BDI systému [13]. 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 [3] nebo 2APL [8], umožňují definovat víry jako některá pravidla Prologu. Ale tuto možnost v systému FRAg uvolňujeme. Předpokládáme, že základ víry 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 víry 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 víry agenta. Pak jeden plán může představovat více než jen jediné chování agenta. Obecně může agent provádět určité výpočty, například může otestovat svůj základ víry 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.

5.1 Motivační příklady

Pro lepší pochopení zde uvedu dva příklady, na kterých bude patrnější jak systém FRAg funguje. Tyto příklady jsou převzaty od Doc. Ing. Františeka 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 víry objevují víry 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 víry, podle předpokladu a v jeho základu víry se tudíž objevují víry `cardToSell(Card)` karet, které má Adam dvakrát.

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

Algoritmus 5: Plán PB k 1. příkladu.

Pak 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, když karta, kterou si Adam vybere po revizi své sbírky, se nehodí pro Betty, ale když si vybere jinou, 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: Víry 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 víru `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.

5.2 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 databázi (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: 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 znovusplnění 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).

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

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

Tento přístup nám dovolí splnit předpoklad systému FRAg. Pokud použijeme databázi Prologu k evidenci víry agenta, můžeme tímto principem splnit plán agenta pro 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 6

Návrh a implementace interpreta systému řízeného záměrem v jazyce Prolog

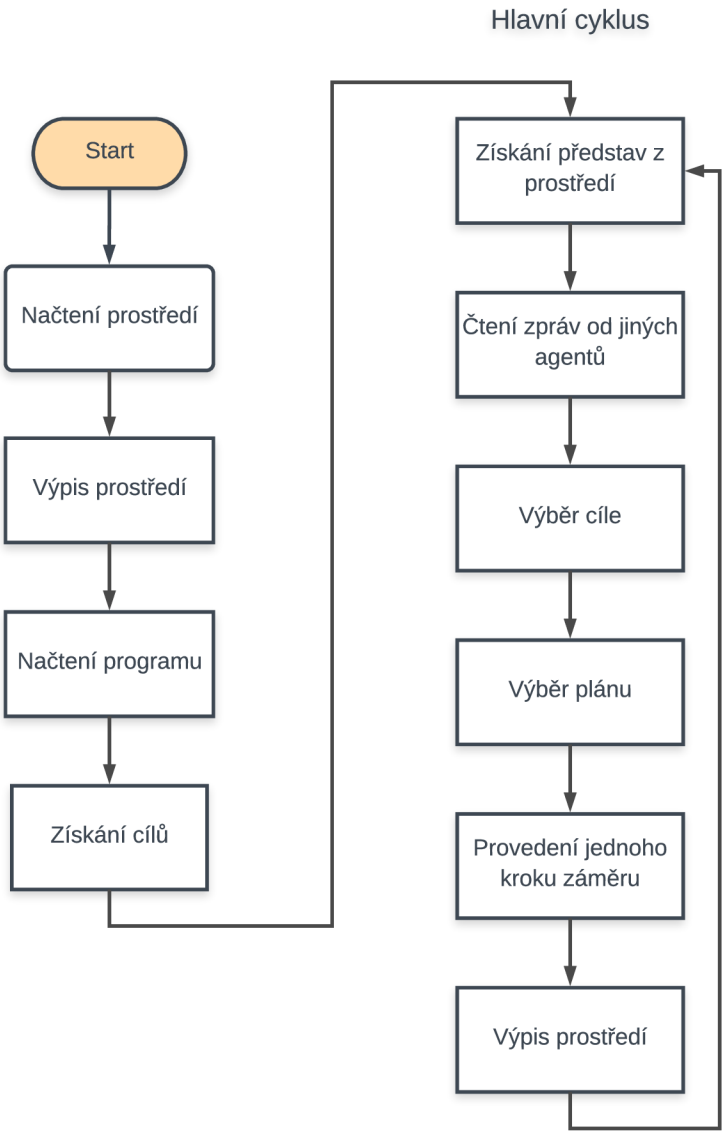
Tato kapitola popisuje mé osobní přínosy v této práci. Konkrétně jde o návrh a implementaci interpretu, který interpretuje systém řízený záměrem v jazyce Prolog s podporou chování FRAg systému. Jako vzor pro můj interpret jsem si vybral interpret Jason a jeho interpretaci AgentSpeak(L). Rozhodl jsem se převést AgentSpeak(L) na interní reprezentaci, přijímanou Prologem. Tu dále interpretovat podobně jako to dělá Jason. Navíc jsem přidal podporu chování FRAg. Zda má interpretace probíhat stejným způsobem jako je tomu u Jason a nebo má využít nadstavbu FRAg, je volitelné.

6.1 Návrh systému

Můj návrh interpretu vychází z modelu znázorněného na obrázku 3.1. Vizualizovaný návrh běhu mého interpretu, který naleznete na schématu 6.1. 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 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.



Obrázek 6.1: Model interpretu

- **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.
- **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.

6.2 Implementace

Tato kapitola popisuje implementaci modelu, který byl předveden v předchozí kapitole. Systém byl implementován v jazyce Prolog, který je interpretovatelný v SWI-prologu.

6.2.1 Převod z AgentSpeak(L) na interní reprezentaci

Před samotnou implementací je nutno sestavit program a nebo jej převést z programu napsaného v AgentSpeak(L).

V aktuální verzi programu neexistuje automatický převodník, který by tuto akci zjednodušil. Vnitřní struktura programu je seznam plánů pro jednotlivé agenty a jeho BNF gramatiku můžete vidět v následující tabulce 6.1. <ATOM> zde značí posloupnost znaků, začínající malým písmenem a <TERM> je struktura, jak je popsaná v kapitole 4.1.2. 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 víra může být buď term a nebo pravidlo. Víry musí mít v této sekci jasně definovanou podobu. Pokud je to term, bude to `bel(belief, owner_name, source)`, kde `belief` je daná víra, což může být jakýkoliv term, `owner_name` je jméno agenta, který tuto víru má a `source` je zdroj představy. V tomto případě to bude `self`. Pokud je víra udaná pravidlem, může to být jakékoliv pravidlo, ale jednotlivé víry 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 programu 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 čtyřech položkách. První je typ plánu (`plan_type`). Význam je stejný jako v AgentSpeak(L) a je uveden v tabulce 3.2. 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`). Posledním parametrem je seznam možností realizace plánu (`list_of_intentions`). Každá jednotlivá realizace má seznam podmínek pro spuštění (`list_of_contexts`) a 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 víry (`'+'` nebo `'-'`) a nebo vlastní operace (`'op'`).

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`.

```
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_intentions ")"
plan_type          ->   ("'+'"|"'-'"|"'+!'"|"'-!'"|"'+?'"|"'-?'"))
atomicity          ->   ("'atomic'"| "'noatomic'")
head_of_plan       ->   <TERM>
list_of_intentions ->   "[" [ intentions ("," intentions)* ] "]"
intentions         ->   "(" list_of_contexts "," list_of_operations ")"
list_of_contexts   ->   "[" [ context ("," context)* ] "]"
context            ->   <TERM>
list_of_operations ->   "[" [ operation ("," operation)* ] "]"
operation          ->   "(" operation_type "," <TERM> ")"|
operation_type     ->   ("'op'"|"'!'"|"'?'"|"'+'"|"'-'")
```

Tabulka 6.1: BNF gramatika interní reprezentace programu.

```
/*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)).

```
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.

6.2.2 Hlavní cyklus

Již máme program, který můžeme interpretovat. 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.

```
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),
    read_message(M),
    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 víry 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 *read_message(M,New_Mess)*** - Slouží ke generování nových plánů na základě přijatých zpráv z předchozího cyklu. Tato funkcionalita však do interpreta nebyla nyní zahrnuta a bude bude implementována v další verzi.
- **Predikát *add_new_message_to_plan(New_mess, Pl_with_bel, Pl_with_mes)*** - Opět se jedná o predikát, který zatím není implementován. Poté co bude doplněna, bude sloužit k přidání zpráv pro dané agenty 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.

6.2.3 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 paramtru `-n`. Pro vytvoření šablony pro agentní systím 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 vírách, 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 konkrétním 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 agrntního systému. Právě tento program je následně interpretován. Jde o jedinou funkci, pokud se jedná o systím 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 víry vycházející z prostředí.
get_beliefs_from_enviroment(Enviroment, ListBeliefs)
```

Touto funkcí se definuje, zda na základě změn v prostředí vznikají nové víry agenta. Nové víry 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 víry nebo '-' při odebrání, `B` je víra 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 7

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 základě něj posoudím funkčnost interpretu. Všechny zde uvedené příklady jsou dodány jako příklady k implementaci interpretu.

7.1 Příklad - Prodej karet

Tento příklad již byl popsán v kapitole 5.1. Agent Adam se snaží vydělat prodejem karet. Již zde nebudu znovu uvádět celý program. Pouze uvedu seznam představ v interní reprezentaci, od kterého se odvozuje chování tohoto příkladu:

```
[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)]
```

Pokud interpret spustíme bez FRAg režimu Adam kartu neprodá. Pro prodej vybere kartu `card1` a plán následně selže při zjišťování zda má Betty o kartu zájem.

Pokud ovše spustíme interpret s parametrem `-f` a systím běží v režimu FRAg. naleznou se dvě možné řešení:

```
% Prvni reseni
[[(!,getMoney(500)),(!,selectCard(card4)),
  (op,reviseCollection()),(?,cardToSell(card4)),
  (!,sellCards(card4,500)),(op,dealWith(betty,card4)),
  (?,bettyWants(card4,500)),[]]]

% Druhe reseni
```

```
[[(!,getMoney(100)),(!,selectCard(card6)),
  (op,reviseCollection()),(?,cardToSell(card6)),
  (!,sellCards(card6,100)),(op,dealWith(betty,card6)),
  (?,bettyWants(card6,100)),[]]]
```

Následně můžeme i posoudit, který z obchodů je pro Adama výhodnější. Díky systímu zpětného navracení a postupného prosazení se nám dostalo všech možných úspěšných plánů.

7.2 Příklad - Hledání cesty čtvercovým polem

Další ilustrační příklad je hledání cesty ve čtvercovém poli. Agent začíná v levém horním rohu a jeho cíl je pravý dolní roh. Má k dispozici dva druhy operací a to krok vpravo a krok dolů.

Samotný program v interní reprezentaci vypadá následně:

```
len(L),  Program = [(agent1,
%beliefs
[bel(direction(down),agent1,self),
bel(direction(right),agent1,self),
bel(home(L,L),agent1,self),
bel(start(1,1),agent1,self)],

%goals
[way_home()],

%plans
[('+!','noatomic',
    way_home(),
    [([],[ ('?',start(X,Y)),
           ('!',go(X,Y)) ])]),
 ('+!','noatomic',
    go(X,Y),
    [([home(X,Y)],[]),
     ([],[ ('?',direction(D)),
           ('op',step(D,X,Y,X2,Y2)),
           ('!',go(X2,Y2))])])
    ])].

step(down,X,Y,X,Y) :- len(L), Y >= L, fail.
step(down,X,Y,X,Y2) :- len(L), Y < L, Y2 is Y + 1.
step(right,X,Y,X,Y) :- len(L), X >= L, fail.
step(right,X,Y,X2,Y) :- len(L), X < L, X2 is X + 1.
```

Tento systém by bez podpory FRAg nikdy k cíli nedospěl, protože by se operace step neustále substituovala s `down` až by narazila na okraj, kde by operace selhala. Pokud spustíme interpretaci FRAg nalezne agent veškeré možné cesty polem. Což je například pro pole o dílce 6 polí 252 cestiček. Je nutno poznamenat, že časová náročnost u tohoto problému roste kvadraticky. Zde je vidět, že se dá vlastností systému FRAg využít i u samotných operací a nemusí nutně být svázána s databází víry.

7.3 Příklad - Nákup dárku na výletě

Tento příklad byl již také uveden v kapitole 5.1. Jen byl pro účely této ilustrace zjednodušen.

Opět nejprve uvedu program:

```
len(L),  Program = [(agent1,
%beliefs
[bel(presentSells(supermarket, book),agent1,self),
bel(presentSells(pernstejn, figure),agent1,self),
bel(presentSells(pernstejn, thimble),agent1,self),
bel(presentSells(bouzov,postcard),agent1,self),
bel(castle(pernstejn),agent1,self),
bel(castle(bouzov),agent1,self)],

%goals
[buyPresent,makeTrip],

%plans
[('+!','noatomic',
    goTo(X),
    [([],[ ('op',printPos(X)) ])]),
 [('+!','noatomic',
    makeTrip,
    [([],[ ('?',castle(X)),
           ('!',goTo(X))])]),
 ('+!','noatomic',
    buyPresent,
     ([],[ ('?',presentSells(X,_)),
           ('!',goTo(X)) ])])
    ])].

printPos(X,M,M):- log('DEBUG',['Pos:',X]).
```

Tento jednoduchý program chce splnit jednoduché dva cíle. Jít na výlet a koupit dárek. V naší verzi programu je odchod na dané místo reprezentován pouze funkcí, která vypíše místo na obrazovku. Pokud interpret spustíme klasicky mimo režim FRAg nalezne jedno řešení:

```
[[(!,buyPresent),(!,makeTrip),
  (?,presentSells(supermarket,book)),
  (!,goTo(supermarket)),(op,printPos(supermarket)),
  [],(?,castle(pernstejn)),(!,goTo(pernstejn)),
  (op,printPos(pernstejn)),[]]]
```

Vidíme, že agent koupil dárek v supermarketu a na výlet šel na Perštejn. V interpretaci pomocí FRAg nalezneme 96 možných řešení. Interpret kombinuje veškeré možné pořadí přepínání mezi událostmi. V tomto je přílišná variace mírně na škodu. Pokud by jsme ovšem sjednotili oba cíle do jednoho a na závěr porovnali, zda je místo nákupu a výletu stejné, došli bychom k podstatně užitečnějšímu výsledku:

```
%prvni vysledek
[[(!,check()),(!,go_castle(pernstejn)),
  (?,castle(pernstejn)),(!,present(pernstejn,figure)),
  (?,presentSells(pernstejn,figure)),
  (!,control(pernstejn,pernstejn,figure)),[]]]
%deruhy vysledek
[[(!,check()),(!,go_castle(pernstejn)),
  (?,castle(pernstejn)),(!,present(pernstejn,thimble)),
  (?,presentSells(pernstejn,thimble)),
  (!,control(pernstejn,pernstejn,thimble)),[]]]
%treti vysledek
[[(!,check()),(!,go_castle(bouzov)),
  (?,castle(bouzov)),(!,present(bouzov,postcard)),
  (?,presentSells(bouzov,postcard)),
  (!,control(bouzov,bouzov,postcard)),[]]]
```

Zde jasně vidíme, že nám vystávají tři možnosti, jak koupit na stejném místě dárek a zároveň si užít výletu.

7.4 Příklad - Čistící roboti na Marsu

Poslední příklad jsem si vypůjčil z příkladů interpretu Jason. Jedná se se o příklad s prostředím. Jelikož je jeho 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 odpadek, uchopí jej a donese do středu plochy, kde jej druhý robot spálí. První agent se poté vrátí na místo, odkud odebral odpadek a pokračuje v prohledávání. Příklad přesně kopíruje předlohu a nemá přirozené ukončení. Proto je vhodné jej spouštět v debug módu. Tento příklad uvádím jen jako ilustraci toho, že interpret zvládá bez problému i práci s prostředím a obecně náročnější systémy.

V závěru chci ještě uvést jeden problém, na který z testování interpretu vyplynul. Jelikož jsou víry reprezentovány pomocí interní databáze v Prologu, nestahuje se na ně zpětné navracení. Pokud máme interpret spuštěný v režimu FRAg je tedy nemožné udržet kontext mezi vírou agenta je jeho plány. Proto by bylo vhodnější reprezentovat víry agentů jako proměnnou.

Kapitola 8

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 Prologu. Tento interpret má podporu systémů s prostředím, dokáže fungovat v debug režimu a podporuje systém FRAg. Dokáže realizovat atomické plány. Tento interpret byl zasazen do jednoduchého vývojového prostředí. Dokáže vytvořit šablonu pro vepsání nového systému a tak vývojáři ulehčit práci. Dále byla sestavena sada příkladů, na které byla funkčnost interpretu ověřena. 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 nekonzistenci víry agenta a jeho plány při zpětném navracení. Jsou to podněty pro další práci a vývoj na interpretu.

8.1 Další vývoj

Na interpretu se dá stále pracovat. Neustále se dá rozšiřovat portfólio vestavěných operací, Jako hlavní bod dalšího vývoje bych zařadil převedení víry agenta z interní databáze Prologu do proměnné. Funkcionalita systému FRAg by se dala i přesto zajistit. Dále by bylo vhodné dodělat komunikaci mezi agenty. Dále by vylo vhodné implementovat nástroj na převádění AgentSpek(L) na interní reprezentaci interpretu. Interpret má nyní velmi jednoduché vývojové prostředí, pro zlepšení komfortu při vývoji agentních systémů by bylo lepší mít nějaké atraktivnější řešení.

Literatura

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

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

[3] Bordini, R. H.; Hübner, J. F.; Wooldridge, M.: *Programming multi-agent systems in AgentSpeak using Jason*, ročník 8. John Wiley & Sons, 2007.

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

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

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

[7] Covington, M. A.; Nute, D.; e Vellino, A.: ISO Prolog. 1993.

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

[9] Horn, A.: On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, ročník 16, č. 1, 1951: str. 14–21, doi:10.2307/2268661.

[10] Huhns, M. N.; Singh, M. P.; Burstein, M.; aj.: Research directions for service-oriented multiagent systems. *IEEE Internet Computing*, ročník 9, č. 6, Nov 2005: s. 65–70, ISSN 1089-7801, doi:10.1109/MIC.2005.132.

[11] 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*, ročník 2, č. 03, 1993: s. 289–318.

[12] Král, J.; Zbořil, F.; 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, ISBN 978-80-8143-127-2, s. 228–233.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9991

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

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

[15] Zbořil, F.; Kočí, R.; Janoušek, V.; aj.: Reactive Planning with Weak Plan Instances. In *Proceedings of 8th ISDA*, IEEE Computer Society: IEEE Computer Society, december 2008, 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
```