VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

LETECKÝ SIMULÁTOR VYTVOŘENÝ V MULTI-AGENTNÍM SYSTÉMU JASON

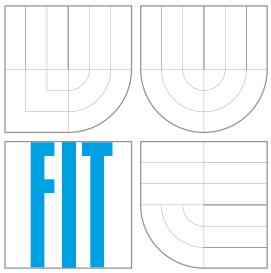
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVLÍNA PUNČOCHÁŘOVÁ

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

LETECKÝ SIMULÁTOR VYTVOŘENÝ V MULTI-AGENTNÍM SYSTÉMU JASON

FLIGHT SIMULATOR BASED ON JASON MULTI-AGENT SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PAVLÍNA PUNČOCHÁŘOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN SAMEK, Ph.D.
SUPERVISOR

BRNO 2014

Abstrakt

Cílem této práce je vytvoření 2D leteckého simulátoru s využitím multi-agentního systému Jason a jazyka AgentSpeak. V tomto simulátoru budou letadla reprezentována agenty a jejich chování bude popsáno plány v jazyce AgentSpeak. Pomocí těchto plánů budou agenti demonstrovat různé typy chování jako je detekce a vyhnutí se kolizi, zaujmutí pozice nebo pronásledování. Výsledná aplikace se skládá ze dvou částí - simulačního modulu a uživatelského rozhraní. Simulační modul je jádrem simulátoru a je vytvořen v systému Jason. Uživatelské rozhraní poskytuje možnost řízení simulace a vytváření uživatelských simulačních modelů.

Abstract

The aim of this work is to create a 2D flight simulator using Jason multi-agent system and AgentSpeak language. In this simulator, the aircrafts will be represented by agents and their behavior will be described by plans implemented in AgentSpeak language. Agents will use these plans to demonstrate different types of behavior such as detection and collision avoidance, the taking of a position or persecution. The final application consists of two parts - the simulation module and the user interface. Simulation module is the core of simulator and it is created in the Jason system. The user interface provides the possibility to control simulations and creating user simulation models.

Klíčová slova

Letecký simulátor, multi-agentní systémy, Jason, AgentSpeak, agent

Keywords

Flight simulator, multi-agent systems, Jason, AgentSpeak, agent

Citace

Pavlína Punčochářová: Letecký simulátor vytvořený
v multi-agentním systému Jason, bakalářská práce, Brno, FIT VUT v Brně, 2014

Letecký simulátor vytvořený v multi-agentním systému Jason

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing. Jana Samka, Ph.D.

. . . . . .
Pavlína Punčochářová
12. května 2014

Poděkování

Chtěla bych poděkovat vedoucímu této práce panu Ing. Janu Samkovi, Ph.D. za cenné rady a ochotu při konzultacích, své rodině a všem, kteří mě v průběhu tvorby práce podporovali.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Při výběru bakalářské práce jsem se zaměřila na inteligentní systémy, konkrétně na jejich využití v letectví, protože mě tato oblast velmi zajímá a chtěla bych se jí hlouběji věnovat. Právě proto mě oslovilo toto zadání, které spojuje obě zmíněné oblasti mého zájmu.
Simulace se v poslední době používá v mnoha souvislostech, ať už jde o přírodní, lidské či technologické systémy. Jedním z důvodu jejího použití je získání poznatků o fungování těchto systémů, případně získání informací o možných reálných dopadech určitého jednání na vybraný systém. Letecké simulace jsou důležité z mnoha důvodů. Používají se mimo jiné pro trénink letových kontrolorů a často hrají klíčovou roli při vyšetřování leteckých katastrof a pomáhají tak přispívat k bezpečnosti v letecké dopravě.

1.1 Cíle práce

Cílem této práce je vytvoření 2D leteckého simulátoru s použitím multi-agentního systému Jason a jazyka AgentSpeak. Práce je rozdělena do dvou částí. V první části je popsána teorie nutná pro pochopení základních souvislostí týkajících se agentních a multi-agentních systémů, v druhé části je pak popsána vlastní implementace simulátoru.

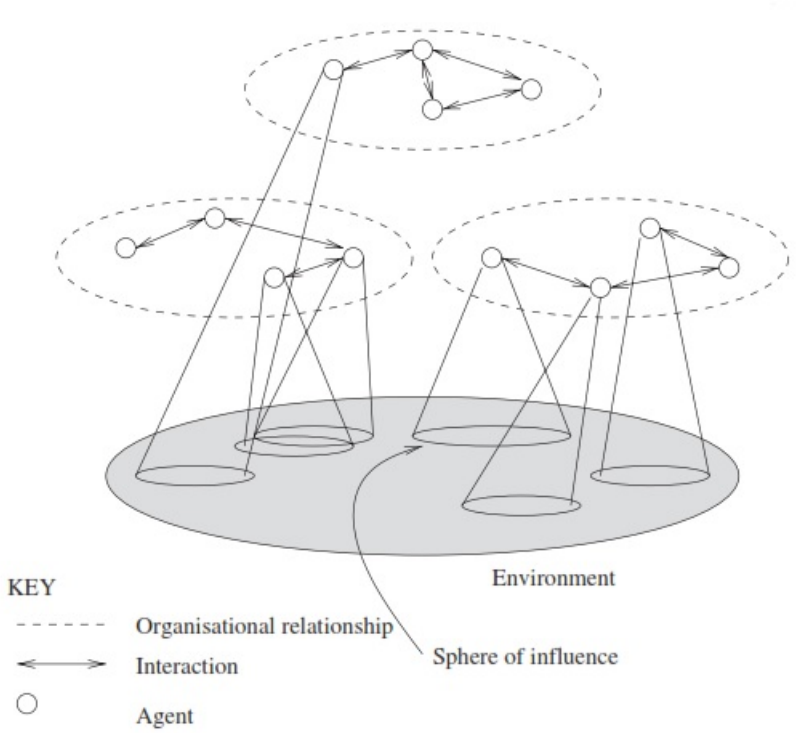
V první kapitole jsou vymezeny cíle práce. Druhá kapitola obsahuje obecný popis multi-agentních systémů a jejich náležitostí. Jsou zde popsáni agenti a jejich vlastnosti, typy a jejich včlenění do prostředí, dále pak komunikace agentů s prostředím a samotné prostředí jako prostor, v němž agenti existují. Na konci této kapitoly jsou popsány jazyky, které se využívají pro komunikaci mezi agenty.

Třetí kapitola je zaměřena na systém Jason, ve kterém je simulátor vytvořen a na jazyk AgentSpeak, který je použit pro implementaci agentů. Dále jsou v této kapitole vysvětleny základní jazykové konstrukce programovacího jazyka AgentSpeak spolu s jednoduchými příklady. Ve čtvrté kapitole je popsán návrh a samotná implementace simulátoru, včetně nástrojů použitých pro jeho realizaci. Jsou zde podrobně popsány jednotlivé části aplikace, tedy simulační modul a uživatelské rozhraní. Pátá kapitola je věnována experimentování s vytvořeným simulátorem. Na příkladech je zde popsáno chování agentů v modelových situacích. Na závěr jsou shrnuty přínosy práce a nastíněny možnosti jejího dalšího rozšíření.

Kapitola 2

Agentní systémy

- **Agentní systém** *(AS, Agent System)* je dán agentem, který je vybaven jistou mírou inteligence a prostředím, ve kterém tento agent působí. V praxi se ale spíše než tyto jednoduché systémy o jednom agentovi vyskytují systémy složitější neboli multi-agentní.
- **Multiagentní systém** *(MAS, Multi-agent system)* je takový systém, ve kterém jedno prostředí sdílí dva a nebo více agentů, kteří jsou schopni jak vzájemné komunikace, tak komunikace s prostředím, ve kterém existují [6].



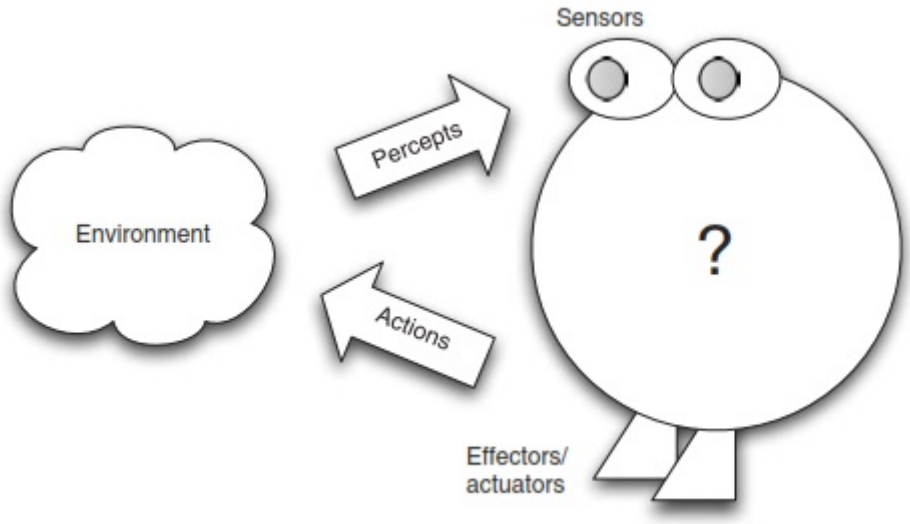
Obrázek 2.1: Typická struktura multi-agentních systémů [6].

Na Obrázku 2.1 je znázorněna typická struktura multi-agentních systémů. Agenti mohou být shlukováni do různých skupin a mohou mít ve svém prostředí tzv. sféru vlivu, což je část prostředí, kterou agent dokáže alespoň částečně kontrolovat [6].

V následujících podkapitolách budou popsány jednotlivé části multi-agentních systémů, tedy samotný agent, jeho vlastnosti, typy a vztah k prostředí, dále pak prostředí jako prostor, ve kterém agenti existují.

2.1 Agent

Agent je reaktivní systém, který má do jisté míry vlastní inteligenci a je schopný sám rozhodnout, jak dosáhnout cíle, pro který je určen. Rozhodování probíhá na základě plánů, které má agent k dispozici. Tento reaktivní systém existuje a koná v rámci nějakého prostředí. Výměna informací mezi agentem a tímto prostředím je pro chod multi-agentního systému velmi důležitá. Agent je schopen získávat podněty z prostředí pomocí senzorů (softwarové prostředky pro snímání vjemů). Získané podněty dále zpracovává a na jejich základě následně ovlivňuje okolní prostředí pomocí akcí prostřednictvím svých efektorů (softwarový mechanismus pro vykonávání akcí) [6]. Tento princip výměny informací za akce je znázorněn na Obrázku 2.2.



Obrázek 2.2: Výměna informací mezi agentem a prostředím [6].

Chování agenta můžeme zjednodušeně přirovnat k lidskému chování. Každý z nás přizpůsobujeme svá rozhodnutí podle informací přijatých z našeho okolí a podobně tak jedná i agent.

2.1.1 Důležité vlastnosti

Kromě umístění v nějakém prostředí by měl mít racionální agent další důležité vlastnosti. Mezi ně patří autonomnost, reaktivita, proaktivita a sociální schopnost. V následujícím textu budou tyto pojmy podrobně vysvětleny [6, 14].

- **Autonomnost** - Schopnost agenta provádět nezávislá rozhodnutí směřující k dosažení určitého cíle. Agent pracuje bez přímého zásahu člověka nebo jiného agenta, má kontrolu nad vlastním rozhodnutím a nad svým vnitřním stavem.

- **Proaktivita** - Vlastnost agenta, která vede k cílenému chování. Pokud má agent nějaký cíl, očekává se, že se pokusí tohoto cíle dosáhnout.
- **Reaktivita** - Schopnost agenta pružně měnit své plány k dosažení cíle na základě změn v prostředí. Některé reakce agenta mohou být na reflexní úrovni.
- **Sociální schopnost** - Schopnost agenta spolupracovat s ostatními agenty v prostředí a případně koordinovat své kroky s dalšími agenty k dosažení společného cíle. Důležitou vlastností je schopnost agenta sdílet své znalosti, plány a cíle s ostatními agenty v daném prostředí.

2.1.2 Typy agentů

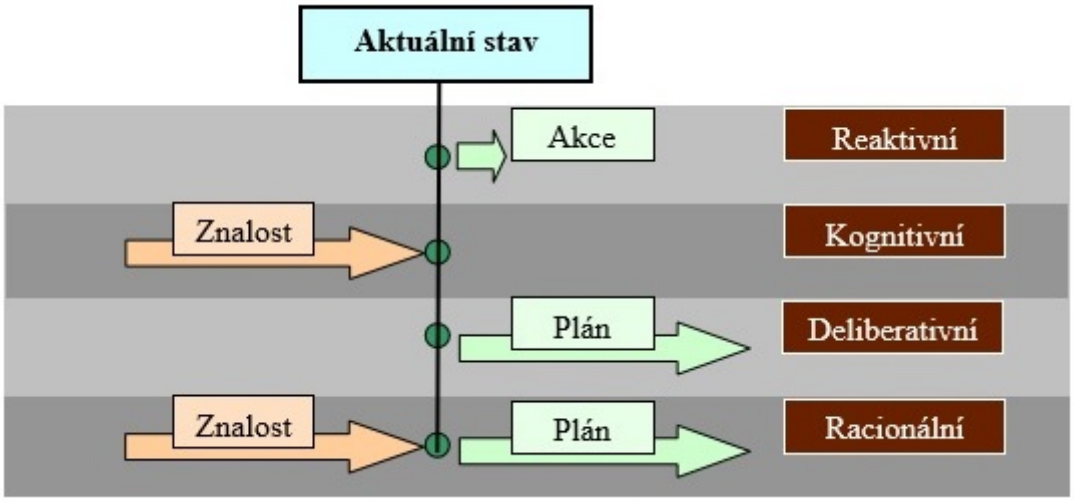
Agenty lze rozdělovat podle různých parametrů, obecně je však dělíme do tří základních skupin podle prostředí, ve kterém působí [14]:

- **Programoví agenti** - softboti (počítačové viry, agenti v počítačových hrách...).
- **Techničtí agenti** - roboti.
- **Biologičtí agenti** - lidé.

Programoví agenti se dále dělí na konkrétní typy, nejčastěji podle míry inteligence, tedy způsobu, jakým agent dosahuje racionálního chování. Některé typy si představíme podrobněji [14].

- **Reaktivní agent** - Tento typ agenta nemá žádnou vnitřní bázi znalostí, pomocí které by reagoval na změny v prostředí. Jeho reakce na podněty z okolí jsou pouze reflexní. Jeho základní vlastností je reaktivita. Tento agent je z pohledu realizace nejjednodušší.
- **Kognitivní agent** - Kognitivní agent je schopen myslet. V tomto případě to znamená, že agent má schopnost zabudovávat informace přijaté z okolí do svého vnitřního prostředí. Dokáže se tedy učit prostřednictvím získaných zkušeností.
- **Deliberativní agent** - Deliberativní neboli rozvážný agent stejně jako agent reaktivní nemá žádnou bázi znalostí. Jeho reakce se ale neomezují pouze na reakce na podněty, jako u agenta reaktivního, ale je schopen pomocí svých vnitřních výpočtů plánovat kroky k efektivnímu dosažení cíle. Jeho základní vlastností je proaktivita.
- **Hybridní agent** - Hybridní agent je speciálním případem agenta, který kombinuje některé nebo všechny z výše uvedených typů v jeden celek. Obvykle se jedná o spojení vlastností agenta reaktivního s agentem deliberativním.
- **Racionální agent** - Racionální agent je speciálním případ hybridního agenta, který kombinuje všechny výše zmíněné vlastnosti agentů. V jeho struktuře se nachází jak plánovací, tak kognitivní jednotka a také znalostní báze. Tento typ agenta je schopen učení a racionálního plánování kroků k dosažení cíle.

Výše zmíněné dělení agentů je znázorněno graficky na Obrázku 2.3.



Obrázek 2.3: Dělení agentů [14].

2.2 BDI Model

BDI model je jeden z nejvyužívanějších přístupů při tvorbě multi-agentních systémů. BDI agent je založen na systému *belief-desire-intention*. Tento systém je analogický k systému biologických agentů, tedy lidí. V následujícím textu budou jednotlivé části BDI systému popsány blíže [6, 7].

- **Beliefs** neboli představy reprezentují informace, které má agent o světě. Tyto představy ovšem nemusí být nutně pravdivé, nemůžeme je tedy nazvat znalostmi.
- **Desires** neboli přání jsou cíle nebo také stavy, kterých by chtěl agent dosáhnout. Tyto cíle mohou být krátkodobé nebo dlouhodobé a mohou být i protichůdné.
- **Intentions** neboli záměry jsou způsoby, jak může agent konat. Přeneseně můžeme říci, že jsou to postupy, které mohou vést ke splnění cíle. Záměry musí být konzistentní, tedy nesmějí být vzájemně v rozporu.

2.3 Prostředí

Prostředí je spolu s agenty základním prvkem multi-agentního systému. Je to prostor, obvykle část reality, ve kterém agenti existují a plní své cíle. Prostředí obecně dělíme na prostředí *fyzické* (pro fyzické agenty, například roboty) a prostředí *programové* (programoví agenti). Dále můžeme prostředí rozdělit na [15]:

- *Plně vs. částečně pozorovatelné* - pokud je prostředí plně pozorovatelné, pak agent může svými senzory snímat jeho kompletní stav. V opačném případě je mu umožněno snímat pouze některé stavy prostředí.
- *Deterministické vs. nedeterministické* - deterministické prostředí je takové prostředí, ve kterém je jeho následující stav předvídatelný, máme-li k dispozici stav původní a akci nutnou pro přechod mezi stavy. V nedeterministickém prostředí tato předvídatelnost neplatí.

- *Statické vs. dynamické* - statické prostředí je takové prostředí, jehož změna může být způsobena pouze akcí agenta. Dynamické prostředí se může měnit i jinak, něž pouze agentní akcí.
- *Spojité vs. diskrétní* - spojité prostředí je prostředí s nekonečným počtem stavů. Neomezuje tedy interakci mezi agentem a prostředím. Oproti tomu prostředí diskrétní má konečný nebo také spočetný počet stavů.

2.4 Komunikace

Komunikace je obecně označována jako proces, během kterého si vyměňují dva a více agentů informace pomocí jednoduchých zpráv. Výměna informací mezi agentem a prostředím a mezi agenty navzájem je velmi důležitá pro dosahování cílů, které jsou pro jednoho agenta příliš složité. Principy výměny informací mezi agentem a prostředím byly popsány v Podkapitole 2.1. Níže budou přiblíženy způsoby komunikace mezi dvěma agenty a následně jazyky využívané pro komunikaci.

2.4.1 Důvody komunikace

Agenti spolu mohou komunikovat z mnoha důvodů, komunikaci proto rozdělujeme do několika typů [9]:

- *Hledání informací* - Agent hledá odpověď na otázku u jiného agenta, u kterého věří, že odpověď zná a může mu ji poskytnout.
- *Dotazování* - Agenti neznají odpověď na otázku a spolupracují na jejím hledání.
- *Přesvědčování* - Agent se snaží přesvědčit jiného agenta, aby přijal některé z jeho tvrzení, se kterým v dané době nesouhlasí.
- *Vyjednávání* - Agenti spolu vyjednávají o rozdělení některých omezených zdrojů tak, aby byli spokojeni všichni zúčastnění.
- *Uvažování* - Agenti spolupracují na rozhodnutí, jak řešit určitý problém, který je v zájmu všech zúčastněných.
- *Rozpor* - Agenti si vyměňují informace za účelem dosažení svých zájmů (hádka).

2.4.2 Komunikační jazyky

V multi-agentních systémech se pro výměnu informací a znalostí mezi agenty používá jazyk *ACL (Agent Communication Language)* [3]. Nejznámějšími ACL jazyky, které se používají v multi-agentních systémech, jsou jazyk *KQML* a *FIPA* [4].

KQML

KQML (The Knowledge Query and Manipulation Language) je vysokoúrovňový komunikační jazyk určený pro výměnu informací pomocí zpráv, který je nezávislý na transportním mechanismu a není vázaný na žádný konkrétní jazyk. Jeho základním principem je použití tzv. *performativů*, neboli klíčových slov, která přesně specifikují komunikační akt. Komunikační zpráva je pak složena z názvu performativu a jeho parametrů určujících obsah zprávy [2].

Performativy lze dělit do několika skupin podle oblasti jejich využití:

- *Základní dotazy* - evaluate, ask-if, ask-about, ask-one, ask-all.
- *Dotazy s více odpověďmi* - stream-about, stream-all, eos.
- *Odpovědi* - reply, sorry.
- *Obecné informace* - tell, untell, cancel, achieve, unachieved.
- *Generátor* - standby, ready, next, rest, discard, generator.
- *Definice schopnosti* - advertise, subscribe, monitor, import, export.
- *Sít'* - register, unregister, forward, broadcast, route.

Pro představu o skladbě zprávy budou uvedeny dva jednoduché příklady.

Příklad dotazovací zprávy:

```
(ask-one
    :sender john
    :content (pocasi(dnes))
    :receiver joe
    :language LPROLOG
)
```

Příklad odpovědi na dotazovací zprávu:

```
(tell
    :sender joe
    :content (pocasi(dnes, oblacno))
    :receiver john
    :language LPROLOG
)
```

V prvním příkladu je uvedena zpráva s performativem typu *ask-one*. Jedná se o dotaz odesílatele na informaci uvedenou ve zprávě v poli *:content*, v tomto případě na stav počasí. Tento typ zprávy se používá v případě, že se agent potřebuje dotázat jiného agenta na otázku, na kterou potřebuje právě jednu odpověď. Mezi další důležitá pole zprávy patří například pole *:sender* udávající odesílatele zprávy, *:reciever* udávající příjemce zprávy nebo pole *:language* udávající jazyk, ve kterém je obsah zprávy napsán. V druhém příkladu je uvedena odpověď na tuto zprávu s použitím performativu *tell*.

Performativy mohou mít kromě parametrů uvedených v příkladech i další parametry. Jejich názvy a významy jsou shrnuty v Tabulce 2.1.

Parametr	Význam
:sender	Odesílatel
:reciever	Příjemce
:content	Obsah zprávy
:language	Jazyk obsahu zprávy
:force	Typ zprávy
:ontology	Specifikace obsahu
:in-reply-to	Kód zprávy, na kterou tato odpovídá

Tabulka 2.1: Typy parametrů.

FIPA

Jazyk FIPA byl navržen stejnojmennou organizaci FIPA (The Foundation for Intelligent Physical Agents), zabývající se agentními technologiemi a standardizací v multi-agentních systémech. Stejně jako jazyk KQML je i tento založen na zprávách realizujících řečové akty a není závislý na konkrétním programovacím jazyku. Jazyky KQML a FIPA jsou si velmi podobné, syntakticky se liší pouze názvy primitiv (komunikačních aktů) [2].

Mezi komunikační primitiva řadíme například *Agree* vyjadřující souhlas s vykonáním nějaké akce, *Failure* vyjadřující informování agenta o selhání pokusu o akci nebo například primitivum *Refuse*, které vyjadřuje odmítnutí provedení dané akce s udáním důvodů [1].

Kapitola 3

Systém Jason

Jason je interpretem pro rozšířenou verzi jazyka *AgentSpeak(L)* [11]. Je implementován v jazyce Java a je dostupný jako open-source distribuovaný pod *GNU LGPL*[1]. Na jeho vývoji spolupracovali Jomi F. Hübner a Rafael H. Bordini, kteří mimo jiné vytvořili za pomoci Michaela Wooldridge publikaci s názvem *Programming multi-agent systems in AgentSpeak using Jason* [6], která poskytuje podrobný popis systému Jason a popis principů programování v jazyce AgentSpeak.

Kromě interpretace jazyka AgentSpeak má systém Jason mnoho důležitých vlastností. Patří mezi ně například [8]:

- Knihovna základních interních akcí.
- Podpora vytváření nových interních akcí v jazyce Java.
- Podpora řečových aktů při komunikaci mezi agenty.
- Podpora rozšiřování základního prostředí naprogramovaného v jazyce Java.
- Přizpůsobitelnost funkcí a architektury agenta - komunikace mezi agenty, vykonávání akcí, úprava znalostí agenta.
- Dostupnost vývojového prostředí ve formě editoru jEdit nebo pluginu pro IDE Eclipse, podpora ladění.

3.1 AgentSpeak(L)

Jazyk AgentSpeak(L), dále jen AgentSpeak, je logicky založeným agentově-orientovaným programovacím jazykem pro tvorbu reaktivních plánovacích agentů, který je založen na architektuře BDI (Belief-Desire-Intention). Tato architektura byla popsána v Podkapitole 2.2.

V jazyce AgentSpeak je používáno několik datových typů. Platí zde, že každý symbol začínající malým písmenem je nazýván *atom*. Oproti tomu každý symbol začínající velkým písmenem je označován jako *logická proměnná*. Atomy společně s *čísly* a *řetězci* řadíme do jednoho celku nazývaného *konstanty*. Dalším typem je *struktura*, která reprezentuje komplexní data.

Jazykové konstrukce tohoto programovacího jazyka mohou být rozděleny do tří základních kategorií, kterými jsou *znalosti*, *plány* a *cíle*. V následujícím textu budou jednotlivé kategorie popsány podrobněji a demonstrovány na jednoduchých příkladech [13, 6].

[1] GNU LGPL - Lesser General Public License - licence svobodného softwaru

3.1.1 Znalosti

Znalosti (Beliefs) v jazyce AgentSpeak reprezentují informace, které jsou agentovi dostupné jak o prostředí, tak o ostatních agentech. Tyto znalosti se sdružují do znalostní báze, která je ve své nejjednodušší podobě popsána souborem literálů. Samotné znalosti jsou reprezentovány symbolicky pomocí predikátů.

Obecně je znalost popisována takto:

$$property(object).$$

Konkrétní příklad znalosti:

$$own\_house(john).$$

Znalosti ve znalostní bázi ovšem nemají význam absolutní pravdy. O agentovi vypovídají pouze to, že ten aktuálně věří, že je daný literál pravdivý. Ve skutečnosti ale pravdivý být nemusí.

3.1.2 Cíle

Cíle (Goals) vyjadřují vlastnosti stavů prostředí, kterých by si agent přál dosáhnout. Cílů lze dosáhnout prostřednictvím vykonávání agentních plánů, které jsou blíže popsány v následující podkapitole. Rozlišujeme dva druhy cílů:

- **Cíl k dosažení** - Tento typ cíle je před svým názvem uveden operátorem '**!**'. Je to cíl, případně jeden z cílů, kterého chce agent dosáhnout. Dosáhnutím cíle je myšlen stav, kdy po provedení určitých kroků začne agent věřit, že literál vyjadřující cíl je pravdivý.
- **Testovací cíl** - Tento typ cíle je před svým názvem uveden operátorem '**?**'. Používá se v kontextu nebo v těle plánu za účelem obnovení nebo také získání informací během vykonávání plánu. Umístění testovacího cíle přímo ovlivňuje vykonávání plánu.
  Je-li testovací cíl umístěn v kontextu plánu a selže, pak plán není vykonán. Pokud testovací plán selže v těle plánu, tak celý plán selhal.

3.1.3 Plány

Plány (Plans) rozumíme postup, který má být použit pro zpracování nějaké události. Každý agent má tyto postupy ve své knihovně plánů. Každý plán se skládá ze tří částí, kterými jsou *spoustěcí událost* neboli *triggering event*, *kontext* a *tělo* plánu, kde *spouštěcí událost* a *kontext* tvoří tzv. hlavu plánu. Jednotlivé části budou v následujícím textu popsány.

Obecně vypadá syntaxe plánu takto:

$$triggering\ event : context \leftarrow body.$$

Konkrétní příklad plánu:

$$+!start : true \leftarrow .print(\text{"}helloworld\text{"}).$$

Spouštěcí událost označuje události, kterými se plán zabývá. V tomto případě je spoustěcí událostí $+!start$.
Tělo plánu $.print(\text{"}helloworld\text{"})$ se provede pouze v případě, že je kontext pravdivý. V tomto příkladu je kontext nastaven na $true$, tělo plánu se tedy provede vždy. Poslední akce v těle plánu je vždy ukončena operátorem '**.**'.

- **Spouštěcí události** - Jak již bylo v předcházejícím textu zmíněno, spouštěcí událost označuje konkrétní událost, kterou se plán zabývá. Spouštěcích událostí existuje více druhů, jejich význam je popsán v Tabulce 3.1.

Notace	Význam
+b	Přidání znalosti
-b	Odebrání znalosti
+!g	Přidání cíle k dosažení
-!g	Odebrání cíle k dosažení
+?g	Přidání testovacího cíle
-?g	Odebrání testovacího cíle

Tabulka 3.1: Typy spouštěcích událostí [6].

- **Kontext** - Kontextem rozumíme okolnosti, za nichž může být plán použit. Aby tedy bylo provedeno tělo plánu, musí být kontext plánu pravdivý. Není-li kontext pravdivý, tak se hledá alternativní plán pro dosažení daného cíle. V kontextu plánu se mohou vyskytovat různé typy literálů. Tyto literálů jsou uvedeny v Tabulce 3.2.

Notace	Význam
/	Agent věří, že **/** je pravda
~/	Agent věří, že **/** je lež
not /	Agent nevěří, že **/** je pravda
not ~/	Agent nevěří, že **/** je lež

Tabulka 3.2: Typy literálů v kontextu [6].

- **Tělo plánu** - Tělo plánu může být složeno z jednoho nebo více kroků. Skládá-li se z jednoho kroku, pak je tento krok ukončen operátorem '**.**'. Je-li nutné provést více kroků, pak jsou tyto kroky od sebe odděleny operátorem '**;**' a poslední z nich je ukončen operátorem '**.**'.

  Obecná syntaxe plánu s více kroky v těle pak vypadá takto:

$$triggering\ event : context \leftarrow step_1; step_2; \ \ldots \ step_n.$$

3.1.4 Interní akce

Interní akce jsou akce, pomocí kterých lze rozšířit programovací jazyk (AgentSpeak) o operace, které v něm jinak nejsou dostupné. Zároveň umožňují přístup například k vnitřním datovým strukturám aplikace naprogramované, stejně jako interní akce, v jazyce Java. Hlavní charakteristikou interních akcí je, že mění prostředí. Od volání externích akcí (akcí prostředí) se liší tím, že mají symbol '**.**' před svým názvem. Systém Jason obsahuje mnoho předdefinovaných interních akcí[2]. Význam často používaných akcí je popsán níže [6].

[2]Přehled všech interních akcí předdefinovaných v systému Jason je dostupný v dokumentaci tohoto systému viz http://jason.sourceforge.net/api/jason/stdlib/package-summary.html.

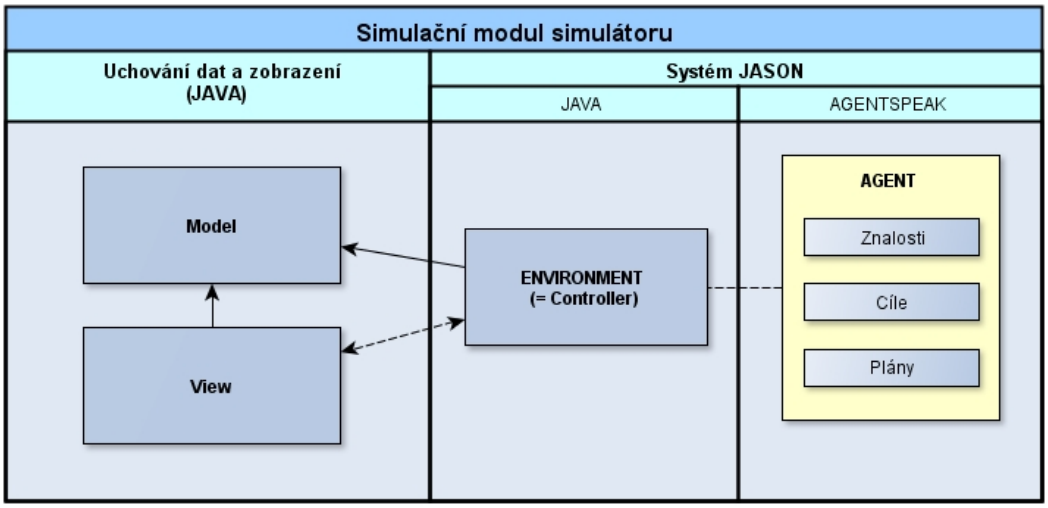
- **.print()** - výpis zpráv na konzoli, na které běží systém. Může obsahovat libovolné množství parametrů, nejenom typu *řetězec*.
- **.send()** - zaslání zprávy agentovi. Obsahuje povinné parametry v pořadí *příjemce*, *performativ*, *zpráva*.
- **.broadcast()** - zaslání zprávy všem agentům. Obsahuje povinné parametry v pořadí *performativ*, *zpráva*.
- **.my_name()** - získání specifické identifikace agenta v rámci multi-agentního systému. Obsahuje jediný parametr, který unifikuje získané jméno agenta.

V případě potřeby je možné definovat vlastní interní akce. Třídy těchto akcí jsou pak odvozeny od třídy *DefaultInternalAction* a implementují metodu *execute()*.

Kapitola 4

Návrh implementace 2D leteckého simulátoru

Pro vytvoření 2D leteckého simulátoru bude využit multi-agentní systém Jason. Jak již bylo zmíněno v Kapitole 3, jedná se o systém založený na jazyce Java. Z tohoto důvodu bude tento jazyk použit i pro samotnou implementaci simulátoru. Jednotlivá letadla budou v tomto systému reprezentována agenty. Každý agent bude mít k dispozici elementární plány napsané v jazyce AgentSpeak, pomocí kterých bude vykonávat let z počátečního do koncového bodu, řešit kolize, případně zaujímat pozice. Simulátor bude rozdělen do dvou částí, na simulační modul a uživatelské rozhraní. Simulační modul reprezentující jádro simulátoru bude založen na modelu *MVC (Model-View-Controller)*[1]. Návrh simulačního modulu je znázorněn na Obrázku 4.1.



Obrázek 4.1: Návrh implementace simulačního modulu.

Simulační modul bude vytvořen v systému Jason a bude obsahovat dvě stěžejní části a to prostředí, které bude reprezentovat *Controller* architektury *MVC* implementovaný v jazyce Java a agenty implementované v jazyce AgentSpeak. Další důležitou součástí simulátoru bude *Model*, který bude uchovávat důležitá data a provádět vnitřní výpočty týkající se pohybu agenta a komponenta

[1] Softwarová architektura rozdělující aplikaci do tří nezávislých komponent tak, aby úprava kterékoliv z jednotlivých částí měla co nejmenší vliv na ostatní části. Více na http://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/

View, která bude zajišťovat vykreslování jednotlivých agentů na zobrazovací ploše.

Cílem každého agenta bude doletět do svého cílového bodu. Pro výpočet optimální trasy letu z počátečního do koncového bodu bude využito teorie Dubinsových křivek [12]. Plánovací algoritmus založený na těchto křivkách využívá pro určení optimální cesty tří primitiv. Mezi tato primitiva patří pohyb rovně, otočka doprava a otočka doleva. Kombinace zmíněných tří základních pohybových akcí stačí k popisu optimální cesty mezi libovolnými dvěma body ve 2D prostoru. Z tohoto důvodu bude mít agent ve své knihovně plánů k dispozici tři základní plány vycházející z uvedených primitiv:

- Plán pro let rovně o vzdálenost N.
- Plán pro otáčku doprava o X stupňů.
- Plán pro otáčku doleva o X stupňů.

Jeden z těchto tří plánů může v jazyce AgentSpeak vypadat například takto:

```
+!zatoc_vpravo(U) : U > 0 <-
                otoc(doprava);
                !zatoc_vpravo(U-1).
```

Tento plán naznačuje postup pro vykonání otočky doprava o úhel U stupňů a bude opakovaně vykonáván se sníženou hodnotou úhlu o jeden stupeň tak dlouho, dokud bude úhel U větší než nula. Při každém kroku tohoto „cyklu“ bude volána akce *otoc(doprava)*. Tato akce bude v prostředí zachycena a způsobí vyvolání patřičné metody *Modelu* simulátoru, která provede výpočet nových souřadnic polohy a směru letu agenta a uložení těchto nových hodnot do vnitřních struktur programu. Po každé této akci bude aktualizováno zobrazení simulátoru. V následující podkapitole budou popsány nástroje, které byly pro vytvoření tohoto simulátoru použity.

4.1 Nástroje pro realizaci

Pro realizaci vlastní implementace bylo zapotřebí několika nástrojů. Mezi tyto nástroje patří vývojové prostředí vhodné pro tvorbu multi-agentních aplikací, dále pak multi-agentní systém a v neposlední řadě vhodný programovací jazyk. V následujícím textu budou jednotlivé nástroje stručně popsány.

- **Vývojové prostředí** - Na počátku implementace bylo využito vývojového prostředí *Eclipse*, které je multiplatformní a lze ho rozšířit pomocí pluginů. Pro vývoj multi-agentních systémů je v prostředí *Eclipse* možné instalovat Jason plugin[2], který podporuje zvýraznění syntaxe jazyka *AgentSpeak* a umožňuje vytvářet projekty založené na systému Jason a následně je spouštět v prostředí Eclipse.

  Později při vytváření uživatelského rozhraní bylo využito vývojového prostředí *NetBeans IDE 7.3.1*, které sice nepodporuje zvýrazňování syntaxe jazyka *AgentSpeak*, ale vytváření uživatelských rozhraní je v něm pohodlnější.

- **Multi-agentní systém** - V simulátoru je využit multi-agentním systému *Jason* založený na architektuře *BDI*. Pro získání přehledu o způsobu tvorby aplikací v tomto systému bylo použito příkladů, které jsou volně dostupné při stažení jakékoliv verze systému Jason ve stromové struktuře ve složce *Examples*. Podrobněji byl tento systém popsán v Kapitole 3. Pro realizaci této práce byl použit systém Jason ve verzi 1.4.0a[3].

[2] Stažení pluginu a postup instalace na http://jasonplugin.wikidot.com/guia-do-usuario.

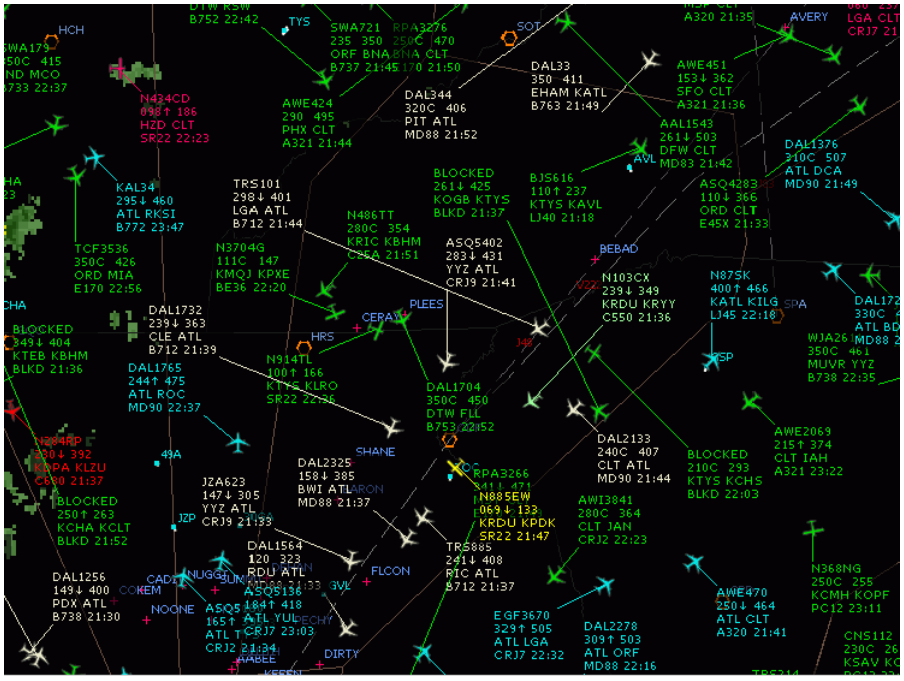
[3] Systém Jason ke stažení na oficiálních stránkách http://sourceforge.net/projects/jason/files/jason/.

- **Programovací jazyk** - Pro implementaci simulátoru je využit objektově orientovaný programovací jazyk *Java* (Version 7, Update 55), který je přenositelný mezi jednotlivými platformami. Pro vytváření agentů, konkrétně jejich znalostí, cílů a plánů, byl použit agentově orientovaný jazyk *AgentSpeak*. Podrobný popis syntaxe tohoto jazyka byl uveden v Podkapitole 3.1.

- **Grafické rozhraní** - Pro vytvoření grafické vizualizace 2D simulátoru bylo využito grafické rozhraní *AWT* a *SWING* poskytované jazykem Java. Rozhraní *AWT (Abstract Windows Toolkit)* poskytuje základní grafické prvky. V aplikaci je využito například pro definování barvy nebo grafického prvku pro vykreslování agentů. Rozhraní *SWING* je v porovnání s prostředím *AWT* novější, funkčně propracovanější a obsahuje více uživatelských prvků. V aplikaci je toto prostředí využito například při vytváření grafického okna simulátoru.

4.2 Základní pojetí simulátoru

Cílem práce bylo vytvořit letecký simulátor s vizualizací ve 2D pro více letadel. Vizualizace byla inspirována zobrazením radaru řízení letového provozu. Příklad možného zobrazení takového radaru je uveden na Obrázku 4.2.



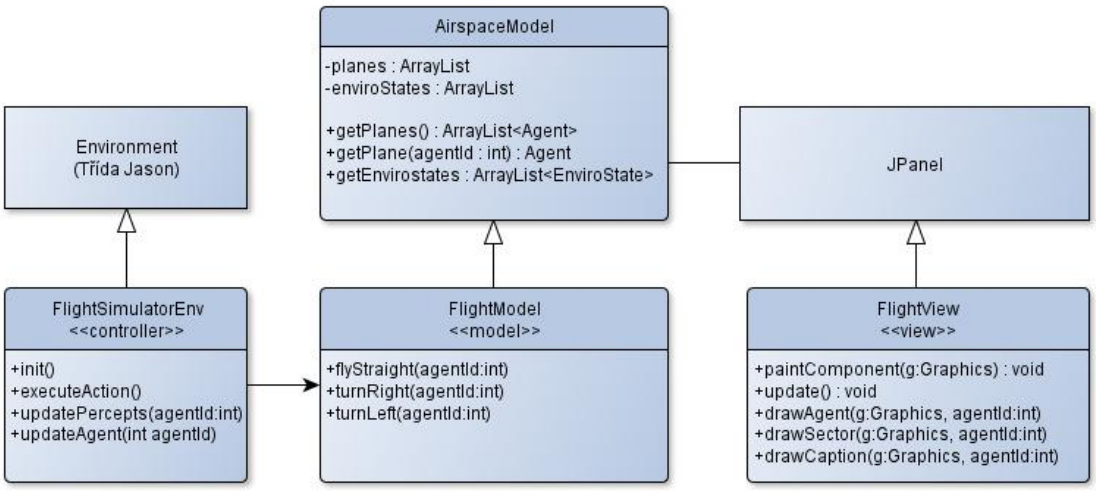
Obrázek 4.2: Příklad radaru [5].

Na takovém radaru jsou jednotlivá letadla zobrazena jako pohybující se body spolu s popiskem, který udává informace o letadlu jako je například jeho volací znak, aktuální letová hladina a podobně.

V simulátoru bude na popisku u každého letadla zobrazen jeho název, letová hladina, směr letu ve stupních v polárním souřadnicovém systému a aktuální stav, ve kterém se agent nachází. Agent se může v průběhu letu dostat do tří stavů - **NORMAL**, **COLLISION**, **ATTACK** nebo **CRASH**. Tyto stavy budou blíže popsány v Podkapitole 4.4.1.

4.3 Rozvržení simulátoru

Simulátor je rozdělen na dvě části a to na *uživatelské rozhraní*, které bude popsáno v Podkapitole 4.7 a samotný *simulační modul*. Simulační modul je založen na systému Jason a vychází z architektury **MVC**. Každá část této architektury je reprezentována samostatnou třídou. Diagram tříd simulačního modulu je zobrazen na Obrázku 4.3.



Obrázek 4.3: Diagram tříd simulačního modulu.

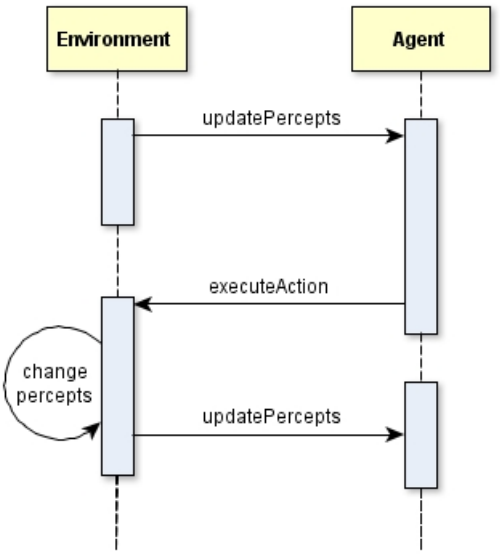
V následujících podkapitolách budou přiblíženy jednotlivé části architektury MVC simulačního modulu a třídy, kterými jsou tyto části zastoupeny v aplikaci.

4.3.1 Controller

Controller reprezentuje řídící logiku simulačního modulu a v simulátoru je zastoupen třídou *FlightSimulatorEnv*. Tato třída reprezentuje prostředí multi-agentního systému a rozšiřuje základní třídu *Environment* implementovanou v systému Jason. Pro přizpůsobení potřebám simulátoru přepisuje některé metody této základní třídy. Mezi tyto metody patří metoda *init* a metoda *executeAction*. Prostředí bude podrobněji popsáno v Podkapitole 4.5.

- **Metoda init** - Metoda zajišťující inicializaci prostředí, která je zapotřebí při spuštění simulace. V těle této metody jsou inicializovány některé vnitřní struktury a je volána metoda *initialUpdatePercepts*, která provede vložení počátečních predikátů do báze znalostí agentů.
- **Metoda updatePercepts** - Tato metoda zajišťuje aktualizování báze znalostí jednotlivých agentů na základě dat obsažených v *Modelu* aplikace.
- **Metoda executeAction** - Metoda zajišťující vykonání akcí prostředí vyvolaných agenty a následné aktualizování báze znalostí agentů pomocí metody *updatePercepts*.

Zmíněné metody jsou důležité pro interakci mezi prostředím a agentem. Princip této interakce je znázorněn na Obrázku 4.4.



Obrázek 4.4: Interakce mezi prostředím a agentem.

4.3.2 Model

Datový model simulátoru reprezentuje vnitřní datové struktury, které uchovávají veškeré informace o agentech a metody, které k těmto strukturám přistupují a manipulují s jejich daty. Základní třídou reprezentující model architektury *MVC* je třída *AirspaceModel*. V této třídě jsou uchována veškerá data, která jsou potřebná pro běh simulace. Třída *AirspaceModel* je rozšířena třídou *FlightModel* obsahující metody pro přepočet souřadnic agentů a úhlu natočení v závislosti na vykonávané akci prostředí.
Úhlem natočení agenta se rozumí úhel letu ve stupních v polárním souřadnicovém systému. Informace o tomto úhlu jsou uchovány v modelu aplikace pro každého agenta. Pro potřeby výpočtu nových souřadnic polohy letadla je úhel letu převeden na radiány pomocí vzorce 4.1. Po přepočtu souřadnic agenta je původní úhel natočení ve stupních upraven na základě vykonaného pohybu a aktualizován v modelu. Mezi metody zajištující pohyb agenta patří:

- **Metoda flyStraight(int agentId)** - Metoda zajišťující přepočet souřadnic polohy agenta, reprezentovaného parametrem *agentId*, pro pohyb vpřed pod aktuálním úhlem natočení. Nové souřadnice polohy jsou vypočítány pomocí vzorce 4.2 a 4.3.

$$\beta_r = \frac{\pi}{180} \cdot \beta \tag{4.1}$$

$$x_n = x_p + \cos \beta_r \tag{4.2}$$

$$y_n = y_p + \sin \beta_r \tag{4.3}$$

  Bod $P = (x_p, y_p, \beta)$ je trojice reprezentující aktuální polohu agenta, kde x_p a y_p jsou souřadnice polohy a úhel β reprezentuje úhel natočení agenta ve stupních. Výsledkem vykonání metody pro let rovně je bod $N = (x_n, y_n, \beta)$, kde x_n a y_n jsou nově vypočtené souřadnice polohy a úhel β je úhlem natočení agenta v polární soustavě souřadnic. Tento úhel se během letu rovně nemění.

- **Metoda turnRight(int agentId)** - Metoda zajišťující přepočet souřadnic polohy a úhlu natočení agenta, reprezentovaného parametrem *agentId*, při vykonání otočky doprava. Otočkou

doprava se rozumí vykonání pohybu po kružnici se středem v bodě $S[x_s, y_s]$ a poloměrem r (dále jen kružnice otáčení) o jeden stupeň.

Je-li agent reprezentován bodem $A = (x_a, y_a, \alpha)$, kde x_a a y_a jsou souřadnice polohy agenta a úhel α je úhel natočení agenta v polárním souřadnicovém systému, pak úhel β mezi středem kružnice otáčení a pozicí agenta je definován jako $\beta = \alpha - 90$. V případě, že úhel $\beta < 0$, pak je provedena korekce tohoto úhlu přičtením hodnoty 360.

Souřadnice středu kružnice otáčení jsou pak vypočítány pomocí vzorce 4.4 a 4.5.

$$x_s = x_a + r \cdot \cos \beta_r \tag{4.4}$$
$$y_s = y_a + r \cdot \sin \beta_r \tag{4.5}$$

Pro výpočet nových souřadnic polohy agenta je potřeba znát úhel γ polohy agenta vůči středu kružnice (dále jen středový úhel γ). Tento úhel se vypočítá z úhlu β. Pokud je $\beta < 180$, pak $\gamma = \beta + 180$, v opačném případě $\gamma = \beta - 180$. Střed kružnice otáčení a středový úhel je vypočítán pouze na začátku vykonávání otočky. Vypočtené hodnoty jsou uloženy do vnitřní datové struktury agenta do třídy *TurnConfig*. Nové souřadnice polohy agenta jsou pak získány pomocí vzorců 4.6 a 4.7. V těchto vzorcích γ_r reprezentuje úhel γ v radiánech. Po výpočtu nových souřadnic polohy je snížena hodnota středového úhlu o jednotku otáčení (1 stupeň).

$$x_n = x_s + r \cdot \cos \gamma_r \tag{4.6}$$
$$y_n = y_s + r \cdot \sin \gamma_r \tag{4.7}$$

Výsledkem vykonání metody pro otočku doprava je bod $N = (x_n, y_n, \delta)$, kde x_n a y_n jsou nově vypočtené souřadnice polohy a úhel δ je úhlem natočení agenta v polární soustavě souřadnic. Úhel δ je získán odečtením jednoho stupně od původního úhlu natočení agenta α.

- **Metoda turnLeft(int agentId)** - Metoda zajišťující přepočet souřadnic polohy a úhlu natočení agenta, reprezentovaného parametrem *agentId*, pro vykonání otočky doleva o jeden stupeň. Výpočet souřadnic středu kružnice otáčení je podobný jako v případě vykonávání otočky doprava s tím rozdílem, že úhel $\beta = \alpha + 90$. Pokud je $\beta > 360$, pak je provedena korekce tohoto úhlu odečtením hodnoty 360. Souřadnice středu kružnice jsou pak vypočítány pomocí vzorce 4.8 a 4.9.

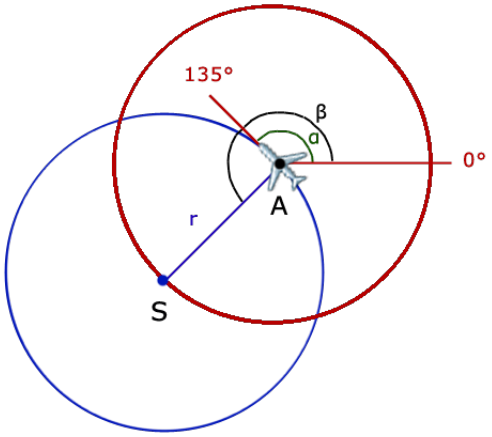
$$x_s = x_a - r \cdot \cos \beta_r \tag{4.8}$$
$$y_s = y_a - r \cdot \sin \beta_r \tag{4.9}$$

Středový úhel γ je vypočten stejně jako při provádění otočky doprava. Střed kružnice otáčení S a středový úhel je vypočítán pouze na začátku vykonávání otočky. Vypočtené hodnoty jsou uloženy do vnitřní datové struktury agenta do třídy *TurnConfig*. Nové souřadnice polohy agenta jsou pak získány pomocí vzorců 4.10 a 4.11. Po výpočtu nových souřadnic je zvýšena hodnota středového úhlu o jednotku otáčení (jeden stupeň).

$$x_n = x_s - r \cdot \cos \gamma_r \tag{4.10}$$
$$y_n = y_s - r \cdot \sin \gamma_r \tag{4.11}$$

Výsledkem vykonání metody pro otočku doprava je bod $N = (x_n, y_n, \delta)$, kde x_n a y_n jsou nově vypočtené souřadnice polohy a úhel δ je úhlem natočení agenta v polární soustavě souřadnic. Úhel δ je získán přičtením jednoho stupně k původnímu úhlu agenta α.

Na Obrázku 4.5 je zobrazena pozice kružnice otáčení při provádění otočky doleva, spolu s vyznačením některých důležitých úhlů.



Obrázek 4.5: Pozice kružnice otáčení při provádění otočky doleva.

Výše uvedené metody kromě výpočtu nových souřadnic polohy agenta aktualizují vnitřní datové struktury agenta nově vypočtenými hodnotami.

4.3.3 View

View je část programu reprezentující grafické rozhraní aplikace. V simulačním modulu je tato část zastoupena třídou *FlightView*, která reprezentuje vykreslovací plochu simulátoru (radar). Samotná třída *FlightView* rozšiřuje třídu *JPanel* a obsahuje metody pro zobrazování jednotlivých komponent radaru jako je souřadnicový systém, jednotlivá letadla a jejich startovní, průletové a cílové body. Pro vykreslování grafických komponent je použita třída *Graphics* (případně její potomek třída *Graphics2D*) z grafického rozhraní *AWT*, které poskytuje řadu metod pro vykreslování. Souřadnicový systém je vykreslován jako kartézský souřadnicový systém s počátkem v levém dolním rohu vykreslovací plochy radaru.
Transformace souřadnic do kartézského systému byla dosažena přepsáním metody *paintComponent* ze základní třídy *JPanel*. K tomu byla využita třída *AffineTransform*, pomocí jejíž metod byla invertována vertikální osa **Y** a posunut střed souřadnicového systému do levého dolního rohu. Data pro vykreslení jednotlivých agentů jsou získávána z *Modelu* aplikace. V ukázce níže je uvedena metoda *paintComponent* pro transformaci zobrazení radaru do kartézského souřadného systému.

```
@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D) g;
    paintXfrm = g2d.getTransform();
    paintXfrm.scale(1.0, -1.0);
    g2d.translate(0, getHeight() - 1);
    g2d.transform(paintXfrm);
    ...
}
```

V této části architektury je implementováno několik důležitých vykreslovacích metod.

- **Metoda draw** - Tato metoda zajišťuje překreslení radaru simulátoru vždy, když dojde ke změně polohy agenta. Bez ohledu na to, který agent změnil svou polohu, při vyvolání této metody je překreslena celá plocha radaru, tedy všichni agenti.
- **Metoda drawAgent** - Metoda zajišťující překreslení agenta. V této metodě jsou volány dílčí metody pro vykreslení jednotlivých komponent jako je detekční výseč (metoda *drawSector*), informační popisek (metoda *drawCaption*) a ikona agenta (metoda *drawPlaneImage*). Tato metoda je volána pro každého agenta zvlášť.
- **Metoda drawPoints** - Metoda zajišťující vykreslení startovního a cílového bodu, případně průletových bodů každého agenta. Z průletových bodů jsou zobrazeny pouze ty body, kterými agent ještě neprolétl.

Instance třídy *FlightView* je vytvořena v hlavním okně aplikace (třída *Simulation*).

4.3.4 Vnitřní datové struktury

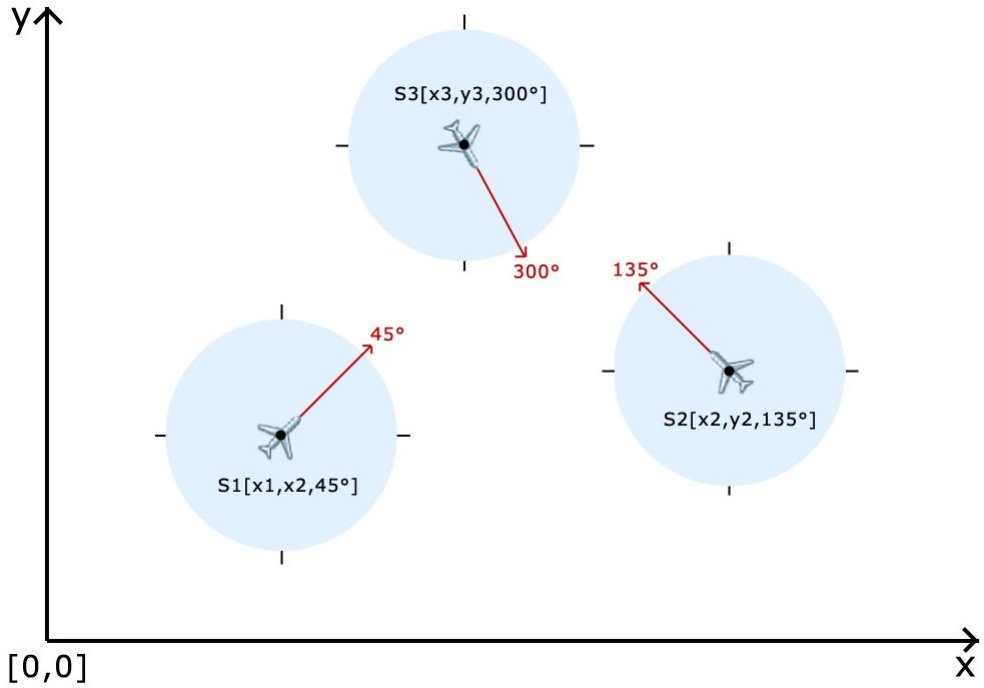
Za účelem udržování vnitřního stavu jednotlivých letadel jsou zapotřebí datové struktury. Z toho důvodu byla vytvořena hierarchie tříd, která uchovává důležité informace o každém agentovi. Data obsažená v těchto strukturách jsou pro vizualizaci stěžejní, protože třída pro vykreslování *FlightView* se při překreslování radaru těmito daty řídí. K těmto datovým strukturám lze přistupovat pomocí *Modelu*.

Mezi základní třídy vnitřní struktury patří:

- **Třída Agent** - Tato třída v sobě udržuje veškeré informace o agentovi. Uchovává jméno agenta, jeho polohu a výšku, letový plán, konfiguraci pro zatáčení a další důležité informace jako například aktuální stav agenta. Stavy, ve kterých se agent může nacházet budou popsány v Podkapitole 4.4.1. Instance následujících tříd jsou zahrnuty do třídy *Agent*.
- **Třída FlightPlan** - Tato třída obsahuje informace o letovém plánu agenta. Je zde zaznamenána startovní a cílová pozice, pozice průletových bodů, případně seznam kroků vedoucích k dosažení průletového nebo cílového bodu.
- **Třída TurnConfig** - Pro účely otáčení agenta je důležité uchovávat konfiguraci dané otočky, to znamená souřadnice středu kružnice otáčení S a úhel γ mezi středem kružnice otáčení a pozicí agenta, jako bylo popsáno v Podkapitole 4.3.2. Tato data jsou uložena ve třídě *TurnConfig*. Při začátku vykonávání otočky agenta o určitý úhel je obsah proměnných tohoto objektu naplněn údaji týkajícími se nové otočky. V průběhu otáčení agenta se střed kružnice otáčení nemění.
- **Třída Position** - Tato třída uchovává přesné souřadnice polohy agenta a úhel jeho natočení ve stupních.
- **Třída PathStep** - Seznam instancí této třídy je uložen v objektu *FlightPlan* a slouží pro uložení kroků letového plánu. Každý krok je složen z typu a parametru. Typ kroku udává, zda se jedná o let rovně, případně otočku doprava nebo doleva a parametr udává počet jednotek vzdálenosti pro případ letu rovně nebo počet stupňů pro případ otočky doprava nebo doleva.

4.4 Návrh a implementace agenta

Každý agent je ve vnitřní datové struktuře aplikace reprezentován třídou *Agent*. Informace o agentovi jsou do této třídy načítány z konfiguračního souboru simulačního modelu, který je ve formátu XML. V tomto souboru je pro každého agenta uvedeno jeho jméno, rychlost letu, příslušnost k jednomu ze dvou týmů (red nebo blue) a letový plán. Letový plán agenta se skládá z jednoho startovního a cílového bodu a libovolného množství průletových bodů. Každý tento bod je reprezentován souřadnicemi polohy ve 2D prostoru a úhlem natočení (úhel, pod kterým má agent tímto bodem prolétnout) v polárním souřadnicovém systému.



Obrázek 4.6: Pozice a úhel natočení agenta na radaru.

Na Obrázku 4.6 je znázorněn kartézský souřadnicový systém radaru s počátkem souřadnic v levém dolním rohu spolu s reprezentací polohy jednotlivých agentů.

Každý agent má definovanou rychlost letu v rozmezí $400-1100\ km/h$ a na základě této rychlosti je mu vypočítána frekvence vykreslování na radaru. Referenční hodnoty, od kterých se frekvence vykreslování jednotlivých agentů odvíjí, jsou rychlost $v_{ref} = 720\ km/h$ a vykreslovací frekvence $f_{ref} = 40\ Hz$. Referenční rychlost letadla byla stanovena na základě nejlepší manévrovací rychlosti pro letadlo Gripen, která se pohybuje v rozmezí $700-750\ km/h$. Referenční frekvence vykreslování pro tuto rychlost byla stanovena empiricky.

Vykreslovací frekvence f_a agenta je pak vypočítána z rychlosti agenta v_a pomocí vzorce 4.12.

$$f_a = \frac{v_a}{v_{ref}} \cdot f_{ref} \tag{4.12}$$

Použití vypočtené vykreslovací frekvence bude uvedeno v Podkapitole 4.5. Jednotka vzdálenosti s, kterou agent urazí při letu rovně za jeden krok vykreslení radaru, je 5 metrů a vychází ze vzorce 4.13.

$$s = \frac{v_a}{f_a} \tag{4.13}$$

Tato vzdálenost je vypočtena na základě rychlosti agenta v_a a jeho vykreslovací frekvence f_a a je pro všechny agenty stejná.

Další důležitá informace o agentovi je výška, ve které letí. Tato výška je stanovena na $11000\ m$, což je běžná letová hladina letadel. Každý agent je dále definován pomocí znalostí, cílů a plánů. Tyto náležitosti jsou implementovány v jazyce AgentSpeak a v následujícím textu budou blíže specifikovány.

4.4.1 Znalosti

Znalosti jako informace, které má agent o okolním prostředí a sám o sobě, byly popsány v teoretické části této práce v Podkapitole 3.1.1. V simulátoru jsou znalosti důležité, protože informují agenta o vjemech z prostředí a pomáhají mu rozhodovat o tom, jaký plán bude vhodné v aktuální situaci použít k dosažení cíle.

Agent svou bázi znalostí kontroluje po každém kroku provádění plánu. Tato kontrola umožňuje agentovi dynamicky měnit své cíle na základě stavu přijatého z prostředí a stavu, ve kterém se agent aktuálně nachází. Znalosti jsou aktualizovány jak z prostředí, tak z vlastní iniciativy agenta.

Znalosti získávané z prostředí:

- **start** - Predikát reprezentující inicializaci agenta. Při přijetí tohoto predikátu agent vypočítá trasu do průletového, případně cílového bodu.
- **enviro(N)** - Predikát reprezentující stav agenta v prostředí. Parametr *N* je unifikován do jednoho ze čtyř klíčových slov reprezentujících stav a to buď ***NORMAL*** (běžný let), ***COLLISION*** (riziko kolize), ***CRITICAL*** (kritická kolize) nebo ***ATTACK*** (možnost pronásledování). V jeden okamžik je v bázi znalostí agenta přítomen pouze jeden tento predikát. Výjimku tvoří znalost *enviro(CRITICAL)*, která se ve znalostní bázi vyskytuje v kombinaci s *enviro(COLLISION)*. V případě klíčového slova ***ATTACK*** obsahuje predikát *enviro* ještě jméno agenta, kterého je možné pronásledovat, například *enviro(ATTACK,r1)*.

Znalosti, které do své báze přidává sám agent na základě znalostí přijatých z prostředí a na základě provedených interních akcí jsou následující:

- **state(N)** - Predikát reprezentující aktuální stav, ve kterém se agent nachází. Agent aktualizuje tuto znalost na základě svého předchozího stavu a predikátu *enviro* přijatého z prostředí.
- **step(X,Y)** - Predikát udávající krok letového plánu. Na základě této znalosti agent vybírá plán, který bude provádět. Blíže bude tento predikát vysvětlen v Podkapitole 4.6.
- **flybyPoint** - Predikát symbolizující, že agent doletěl do aktuálně nastaveného průletového bodu a může pokračovat v letu do následujícího bodu průletu nebo přímo do cílového bodu na základě letového plánu.
- **goal** - Predikát určující, že agent doletěl do svého cílového bodu.

4.4.2 Cíle

Počátečním cílem každého agenta je doletět do stanoveného cílového bodu. Tento cíl je definován v jazyce AgentSpeak takto:

$$!fly.$$

Rozhodování o tom, jaký plán agent začne vykonávat probíhá na základě testovacího cíle *?check*, který testuje predikáty přijaté z prostředí obsažené v bázi znalostí agenta. Tento testovací cíl je vykonáván po každém kroku plánu pro let agenta. S jeho pomocí může agent dynamicky reagovat na změny v prostředí a přizpůsobovat své plány aktuální situaci.

Plány pro testovací cíl vypadají takto:

```
+?check : start <- ...
+?check : state(normal) & enviro(collision) <- ...
+?check : state(normal) & enviro(attack,E) <- ...
+?check : state(attack,_) & enviro(attack,E) <- ...
+?check : state(attack,_) & enviro(normal) <- ...
+?check : state(collision) & enviro(collision) <- ...
+?check : state(collision) & enviro(normal) <- ...
+?check : state(collision) & enviro(attack,E) <- ...
+?check.
```

V ukázce jsou uvedeny některé kombinace znalostí, které se kontrolují v testovacím cíli. Pokud má agent ve své bázi znalostí predikát *start*, znamená to jeho inicializaci. V tom případě agent pomocí interní akce *internal.getFlightPlan* vypočítá trasu z počátečního do cílového bodu a pomocí interní akce *internal.setNextStep* nastaví do své báze znalostí predikát *step* určující první krok letového plánu.

4.4.3 Plány

Agenti mají k dispozici ve své knihovně plánů tři základní plány, jejichž kombinací je možné doletět z libovolného počátečního bodu do libovolného cílového bodu. Těmito plány jsou:

- Plán pro let rovně o N jednotek vzdálenosti **+!fly_straight(N)**.
- Plán pro otočku doprava o úhel N **+!turn_right(N)**.
- Plán pro otočku doleva o úhel N **+!turn_left(N)**.

Výběr vhodného plánu pro vykonání probíhá na základě testování přítomnosti predikátu *step* v agentově bázi znalostí. Tento predikát není přijímán z prostředí, ale vkládá ho do své báze znalostí sám agent pomocí interní akce. Testování probíhá v kontextu plánu *+!fly* a na jeho základě je vykonán příslušný plán.

```
+!fly : step(straight, N) <- !fly_straight(N).
+!fly : step(left, N) <- !turn_left(N).
+!fly : step(right,N) <- !turn_right(N).
```

Pro vyvolání plánu pro vykonání úseku letu musí být ve znalostní bázi agenta přítomen příslušný predikát. V případě otočky doprava je to predikát *step(right,N)*, kde N vyjadřuje úhel ve stupních, o který se má agent v daném směru otočit.

```
+!turn_right(N) : N > 0 <-
                  turn(right);
                  -step(right, N);
                  +step(right, N-1);
                  ?check;
                  !fly.
```

Plán uvedený v ukázce slouží pro vykonání otočky doprava o úhel N. Pokud je tento úhel větší než nula, tak je volána akce prostředí *turn(right)*. V prostředí je tato akce odchycena v metodě

executeAction a zpracována tak, jak je popsáno v Podkapitole 4.5. Dále je aktualizován predikát *step* ve znalostní bázi agenta, který udržuje informaci o tom, o kolik stupňů se má agent ještě otočit. Pomocí testovacího cíle *?check* jsou zkontrolovány znalosti aktualizované z prostředí vždy po každém kroku plánu. Pokud není v bázi znalostí přítomen predikát symbolizující riziko kolize *enviro(collision)* nebo možnost pronásledování *enviro(attack,E)*, pak agent pokračuje opět s cílem *!fly*.

4.5 Návrh a implementace prostředí

Prostředí je prostor, ve kterém agenti existují a vykonávají své plány. V této práci je reprezentováno třídou *FlightSimulatorEnv*, která zastupuje funkci *Controlleru* při využití návrhového modelu *MVC*. Tato třída rozšiřuje třídu *Environment*, která je obsažena přímo v systému Jason a poskytuje metody pro předávání informací mezi prostředím a agentem. Důležité metody prostředí:

- **Metoda executeAction(ID agenta, název akce)** - Nejdůležitější metoda prostředí, která zpracovává akce volané agenty. Tyto akce jsou v prostředí definovány pomocí stejnojmenných termů. Při vykonávání této metody jsou porovnány termy definované v prostředí s akcí volanou agentem. V případě, že volaná akce se shoduje s některou akcí definovanou v prostředí, pak je volána příslušná metoda *Modelu*, která tuto akci vykoná. Metody pro vykonávání akcí byly popsány v Podkapitole 4.3.2.

  V následující ukázce kódu je uvedena část metody *executeAction*, kde je testována shoda akce volané agentem s některou z akcí definovanou v prostředí. Proměnná *delay* reprezentuje převrácenou hodnotu vykreslovací frekvenci agenta vypočtenou na základě referenčních hodnot frekvence a rychlosti tak, jak bylo uvedeno v Podkapitole 4.4.

```
public static final Term flyStraight = Literal.parseLiteral("fly(straight)");
public static final Term turnRight = Literal.parseLiteral("turn(right)");
public static final Term turnLeft = Literal.parseLiteral("turn(left)");

@Override
public boolean executeAction(String ag, Structure action) {
    ...
    if (action.equals(flyStraight)) model.flyStraight(agentId);
    else if (action.equals(turnRight)) model.turnRight(agentId);
    else if (action.equals(turnLeft)) model.turnLeft(agentId);
    else return false;
    ...

    if (!Thread.interrupted()) {
        Thread.sleep(delay);
    }
    return true;
}
```

4.6 Výpočet trasy letadla

Výpočet letové trasy si zajišťuje každý agent sám pomocí volání interní akce *internal.getFlightPlan*. Výpočet probíhá s použitím *Dubinsových křivek* a veškeré výpočty potřebné pro získání optimální trasy probíhají ve třídě *TrackPlan*.

Vstupními hodnotami *Dubinsova algoritmu* jsou tři parametry:

- Startovní konfigurace letadla K_i
- Cílová konfigurace letadla K_o
- Poloměr zatočení letadla

Počáteční konfigurace letadla je trojice $K_i = (x_i, y_i, \alpha)$, analogicky tak koncová konfigurace $K_o = (x_o, y_o, \beta)$, kde x_i, y_i a x_o, y_o jsou souřadnice pozice letadla v kartézském souřadnicovém systému a úhel α, β udává úhel natočení letadla v polární soustavě souřadnic. Poloměr otáčení letadla udává poloměr kružnice, po které agent opíše dráhu při provádění otočky doprava nebo doleva, závislou na úhlu, o který se má otočit.

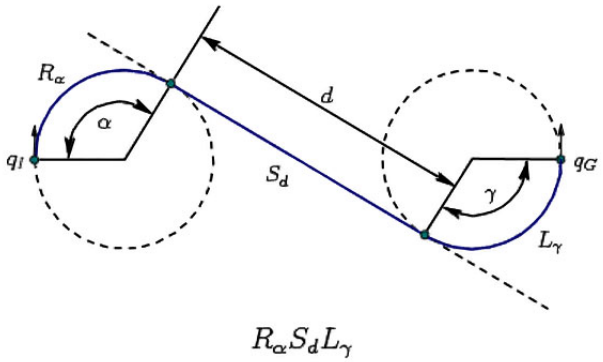
Výstupem algoritmu je kombinace tří elementárních kroků, pomocí kterých lze vytvořit optimální cestu vzhledem k poloze počátečního a cílového bodu. Každý krok se skládá z typu a parametru, jak je popsáno v Tabulce 4.1.

Typ	Význam	Parametr
LEFT	Zatoč doleva	Úhel zatočení
RIGHT	Zatoč doprava	Úhel zatočení
STRAIGHT	Pohyb rovně	Délka pohybu

Tabulka 4.1: Typy kroků a jejich parametry.

Tyto kroky lze kombinovat do sekvencí o třech krocích. Kombinace kroků mohou být následující: LSL, RSR, LSR, RLR, LRL a RSL.

Na Obrázku 4.7 je znázorněn příklad sekvence kroků RSL spolu s parametry pro každý krok této sekvence.



Obrázek 4.7: Příklad sekvence kroků typu RSL [10].

Pomocí sekvence kroků je možné dostat se z libovolného startovního do libovolného cílového bodu. Třída *TrackPlan* obsahuje metody pro výpočet parametrů jednotlivých kroků pro každou sekvenci zvlášť. Výsledná sekvence generující optimální cestu je vybrána na základě výpočtu délky cesty pro každou sekvenci. Sekvence s nejmenší délkou je zároveň optimální. Výpočty optimální cesty použité v simulátoru vycházejí z práce [12].

Výsledkem Dubinsova algoritmu je tedy jedna ze zmíněných sekvencí. Jednotlivé kroky výsledné sekvence jsou reprezentovány třídou *PathStep* a jsou vloženy do vnitřní struktury agenta do třídy *FlightPlan*. Každý krok se skládá z typu kroku *left*, *right* nebo *straight* a parametru udávajícího délku tohoto kroku. V případě letu rovně je parametrem vzdálenost, v případě otočky je to úhel, o který se má agent otočit.

Vypočtené úseky trasy jsou přidány do seznamu kroků letového plánu agenta ve třídě *FlightPlan*. Tyto kroky jsou poté postupně přidávány do báze znalostí agenta pomocí volání interní akce *internal.getNextStep* a jsou reprezentovány predikátem *step(typ_kroku, parametr)*. V jednu chvíli je v bázi znalostí agenta přítomen pouze jeden tento predikát.

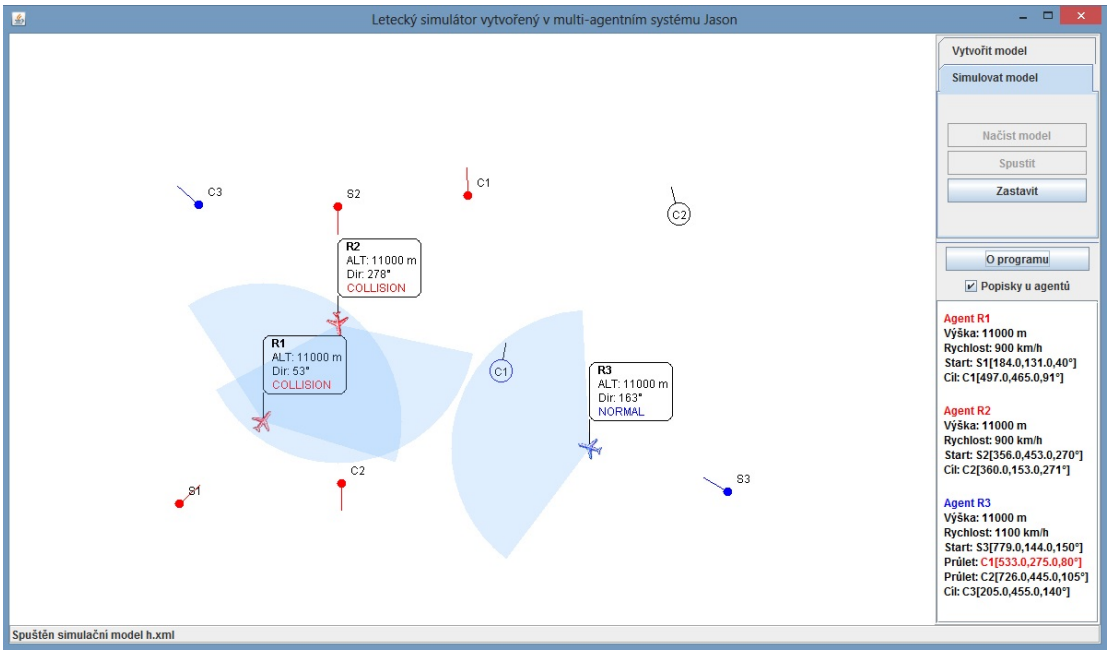
Výpočet trasy agent provádí v několika případech. Při startu letadla, kdy má agent v bázi znalostí přítomen predikát *start*. Dále je to tehdy, když agent přechází ze stavu kolize nebo ze stavu pronásledování do stavu běžného letu a potřebuje přepočítat trasu do následujícího průletového nebo cílového bodu.

4.7 Uživatelské rozhraní

Uživatelské rozhraní je reprezentováno třídou *Simulation*. Při spuštění aplikace je vytvořeno hlavní okno, které obsahuje kontrolní panel s tlačítky pro řízení simulace, informační panel zobrazující informace o jednotlivých agentech, prostor pro zobrazení samotné simulace (radar) a stavový řádek. V informačním panelu jsou pro každého agenta zobrazeny údaje o jeho výšce, startovním a cílovém bodu a bodech průletu. V případě průletových bodů jsou barevně odlišeny body, kterými agent již proletěl, aktuální bod průletu, do kterého míří a body, kterými teprve prolétat bude. Uživatelské rozhraní simulátoru umožňuje uživateli dvě akce - načtení existujícího simulačního modelu a vytvoření nového simulačního modelu.

4.7.1 Načtení simulace

Základní akcí, kterou lze v simulátoru provádět, je spouštění simulace. Vstupní data pro simulaci jsou načítána z konfiguračního souboru ve formátu XML. Pro výběr tohoto souboru ze souborového systému je využito třídy *JFileChooser*. Vybraný soubor je pomocí nástroje pro práci s *XML* soubory *JAXB* načten do třídy *FlightModel*. Příklad obsahu konfiguračního souboru je uveden v Příloze B. Simulaci je možné spustit a zastavit pomocí příslušného tlačítka na kontrolním panelu. Na Obrázku 4.8 je zobrazeno hlavní okno aplikace při spuštění simulačního modelu.



Obrázek 4.8: Okno aplikace při spuštění simulace.

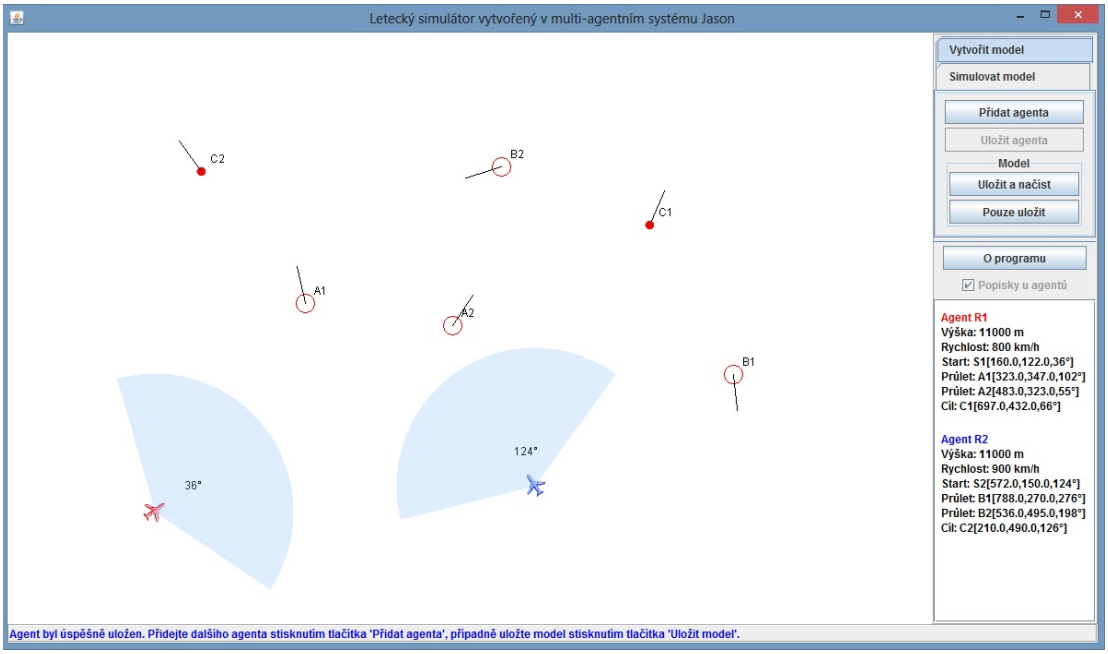
4.7.2 Vytvoření modelu simulace

Simulátor kromě načtení existujícího simulačního modelu umožňuje vytvoření vlastního modelu pomocí editoru. Editor je v aplikaci reprezentován třídou *DrawArea*, která rozšiřuje základní třídu *JPanel*. Pro zachycení kliknutí je na tento panel přidán *MouseListener* a pro zachycení pohybu myši je využit *MouseMotionListener*. Editor je spuštěn otevřením záložky *Vytvořit model* na kontrolním panelu. Na stavovém řádku v dolní části okna je uživateli zobrazován postup k vytvoření agenta.

Kliknutím myši na plochu radaru jsou přidány souřadnice bodu a následným pohybem myši a kliknutím je vybrán úhel letu agenta. Zadávání bodů agenta v editoru probíhá v pořadí:

1. *Souřadnice startovního bodu a úhel natočení letadla ve startovním bodě.*
2. *Souřadnice cílového bodu a úhel natočení letadla v cílovém bodě.*
3. *Souřadnice průletového bodu a úhel natočení letadla v průletovém bodě.*

Na Obrázku 4.9 je znázorněno okno aplikace v případě vytváření simulačního modelu.



Obrázek 4.9: Okno aplikace při vytváření modelu.

Zadání počáteční a koncové souřadnice spolu s úhly natočení agenta jsou minimální informace nutné pro vytvoření nového agenta. Průletových bodů je možné zadat libovolné množství. Souřadnice nově přidaných bodů spolu s úhly natočení agenta jsou zobrazovány na informačním panelu v hlavním okně simulátoru.

Při ukládání nově vytvořeného agenta je uživatel vyzván k zadání jména agenta, jeho rychlosti a barvy týmu, který bude agent reprezentovat.

Po ukončení vytváření modelu je možné tento model uložit do souboru ve formátu XML pomocí nástroje *JAXB*.

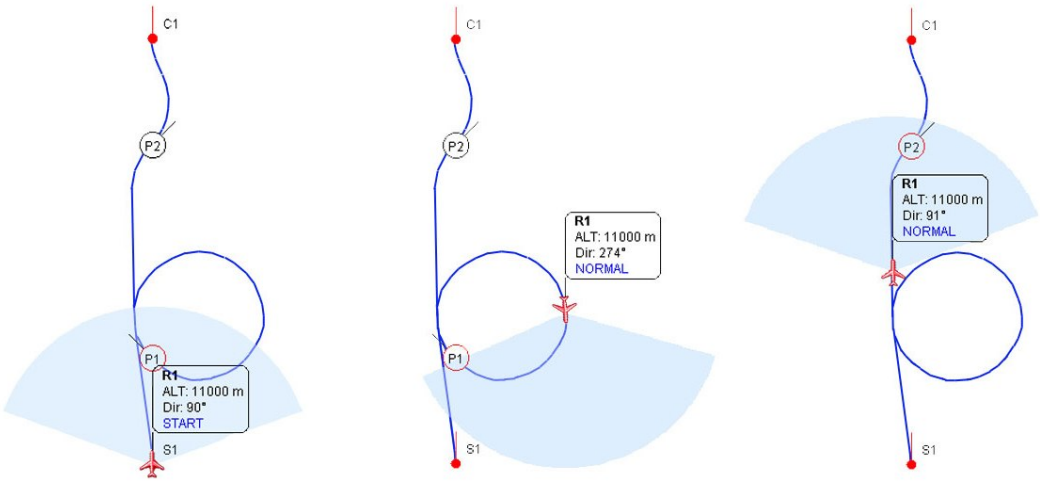
Kapitola 5

Modely chování a experimentování

Pro testování simulátoru byly vybrány tři typy modelových situací. Patří mezi ně schopnost agenta zaujmout pozici na daných souřadnicích s daným úhlem natočení, která je důležitá pro samotný let agenta a pro jeho dolet do cílového bodu. Další testovanou modelovou situací je řešení kolizí dvou a více a třetím modelem je pronásledování. Každý model bude popsán v samostatné podkapitole.

5.1 Zaujmutí pozice

Zaujmutí pozice je v simulátoru promítnuto ve schopnosti agenta doletět do cílového bodu v definovaném úhlu natočení, případně proletět průletovými body pod daným úhlem natočení. Princip výpočtu letové trasy z bodu do bodu byl popsán v Podkapitole 4.6. Na Obrázku 5.1 je znázorněna trasa letu agenta z počátečního bodu $S1$ do cílového bodu $C1$ spolu se zaujmutím pozice v průletových bodech $P1$ a $P2$. Na radaru jsou zobrazovány pouze průletové body, kterými agent zatím neprolétl.



Obrázek 5.1: Ukázka zaujmutí pozice v průletových bodech.

Při průletu agenta průletovými body, případně při jeho doletu do cílového bodu se může stát, že agent doletí do tohoto bodu s odchylkou v pozici, případně odchylkou úhlu natočení. K těmto nepřesnostem dochází z důvodu diskrétní simulace.

5.2 Kolize

Součástí práce bylo i prozkoumání oblasti kolizí agentů a implementace reakcí agentů k jejich zabránění. V reálném provozu může být na radaru střediska letové kontroly v jednom časovém okamžiku více letadel, proto je důležité zajistit, aby se letadla ve vzduchu nesrazila. V realitě se o to stará řízení letového provozu. V případě selhání pozemní kontroly existuje ještě autonomní bezpečnostní systém *TCAS*, který piloty v kokpitu před kolizí varuje a nabídne jim úhybný manévr pro vyhnutí se této kolizi.

5.2.1 Detekce

V simulátoru je kontrola rizika kolize prováděna ve třídě *CollisionDetector*. Detekce pomocí této třídy je prováděna v prostředí v metodě *executeAction* a provede se po provedení akce prostředí vždy pro agenta, který tuto akci vyvolal. Mezi důležité metody této třídy patří metoda *checkAgentRadius*, která porovnává pozici agenta, který kontrolu vyvolal (dále jen kontrolovaného agenta), s pozicemi ostatních agentů a testuje, zda se některý z agentů nachází uvnitř kruhové detekční zóny kontrolovaného agenta.

Tato metoda k základní kontrole používá vzorec 5.1 pro určení, zda se bod $E[x_2, y_2]$ reprezentující kolizní objekt nachází v kružnici se středem v bodě $A[x_1, y_1]$ a poloměrem r, kde je střed A reprezentován kontrolovaným agentem. Poloměr detekční kružnice byl stanoven empiricky a je nastaven na 150 vzdálenostních jednotek na radaru, tedy na 750 metrů a je pro všechna letadla stejný.

$$(x_2 - x_1)^2 + (y_2 - y_1)^2 < r^2 \tag{5.1}$$

Pokud je nalezen kolizní objekt v kruhové zóně kontrolovaného agenta, pak je potřeba zjistit, zda se nachází přímo v jeho detekční zóně, tedy ve výseči o úhlu 140° před letadlem. Tato kontrola probíhá v metodě *checkCriticalSector* a je založena na zjištění úhlu mezi pozicí kontrolovaného agenta (reprezentován bodem $A = (x_1, y_1, \alpha)$) a kolizního objektu (reprezentován bodem $E = (x_2, y_2, \beta)$) pomocí vzorce 5.4 a porovnání této vypočtené hodnoty s hranicemi detekční výseče.

$$d_y = y_2 - y_1 \tag{5.2}$$

$$d_x = x_2 - x_1 \tag{5.3}$$

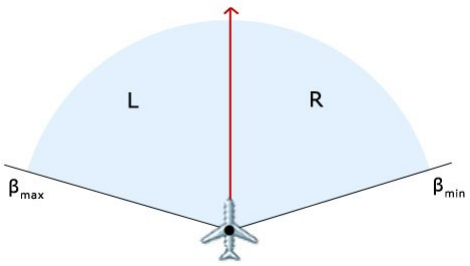
$$\theta = arctan2(dy, dx) \cdot \frac{180}{\pi} \tag{5.4}$$

Pro kontrolu, zda vypočítaný úhel θ leží v rozsahu detekční výseče kontrolovaného agenta, je potřeba vypočítat spodní a horní hranici výseče pomocí vzorce 5.5 a 5.6. Detekční výseč, spolu se zobrazením hranic této výseče, je zobrazena na Obrázku 5.2.

$$\beta_{max} = \alpha + 70 \tag{5.5}$$

$$\beta_{min} = \alpha - 70 \tag{5.6}$$

Pokud se úhel mezi agenty θ nachází v intervalu úhlů $< \beta_{min}, \beta_{max} >$, pak se jedná o riziko kolize, v opačném případě ne. V případě rizika kolize je agent prostředím informován pomocí predikátu *enviro(collision)* a predikátu *sector(left)* nebo *sector(right)* podle toho, v jaké části detekční výseče se kolizní objekt nachází. Aby byl kolizní objekt detekován v levé části výseče, musí platit, že $\beta_{max} \geq \theta > \alpha$. V tom případě je do báze znalostí agenta vložen predikát *sector(left)*. Pro detekci



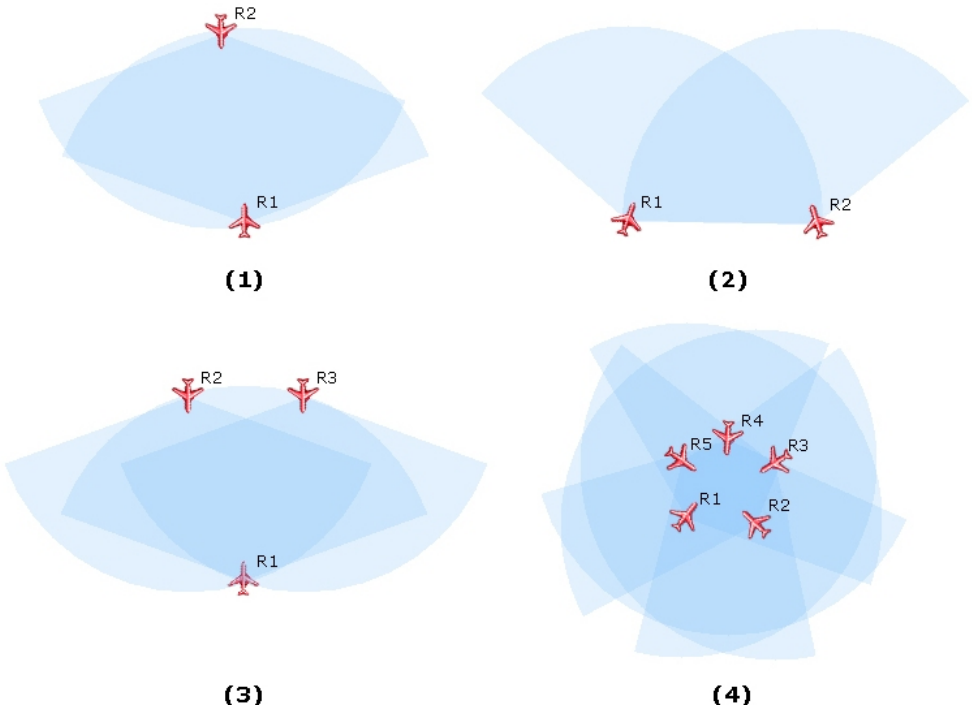
Obrázek 5.2: Detekční výseč agenta.

objektu v pravé části výseče musí platit $\alpha \geq \theta > \beta_{min}$. V takovém případě je agentovi vložen predikát *sector(right)*.

Posledním predikátem, který může být na základě detekce kolize vložen do báze znalostí agenta, je predikát *enviro(critical)*. Tento predikát symbolizuje, že byla detekována kritická kolize, kterou není možné vyřešit výše zmíněným způsobem. Riziko kolize je klasifikováno jako kritické, pokud kolizní objekt zároveň detekuje kontrolovaného agenta v opačné části detekční výseče než agent detekoval tento objekt. Zjednodušeně - pokud agent detekuje kolizní objekt v levé části detekční výseče a kolizní objekt detekuje agenta v pravé části výseče, pak je kolize vyhodnocena jako kritická. To platí i v opačném případě. Agent tento predikát přijatý z prostředí detekuje a přidá si do své báze znalostí vlastní predikát *critical*.

5.2.2 Vyhnutí se

V případě detekce rizika kolize je nutné se této kolizi vyhnout. Řešení kolize vyhnutím se je podmíněno tím, že detekovaný kolizní agent patří do stejného týmu jako agent, který ho detekoval.



Obrázek 5.3: Modely kolizí agentů.

Pro experimentování s mechanismem řešení kolizí bylo použito modelů zobrazených na Obrázku 5.3. V následujícím textu budou popsány přístupy k řešení kolizí, které byly v simulátoru testovány.

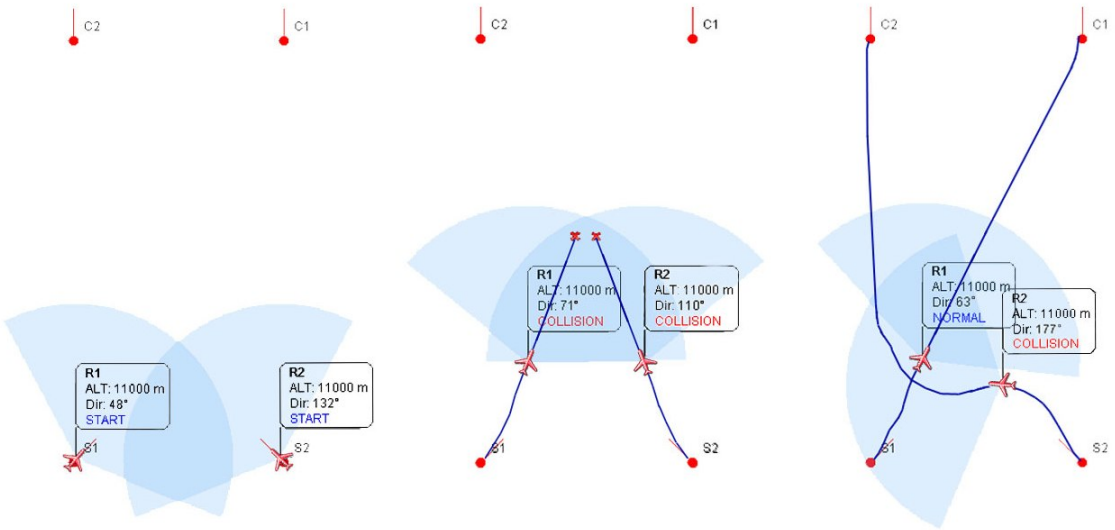
1. **Úhybný manévr doleva** - Na počátku řešení kolizí bylo chování agenta implementováno tak, aby byl vždy při detekci kolizního objektu proveden úhybný manévr otočkou doleva nehledě na to, ve které části detekční výseče se kolizní objekt nacházel. Tento způsob byl ale použitelný pouze v případě řešení jednoduchých kolizí dvou agentů a ne vždy byl efektivní. V případě testovaných modelů z Obrázku 5.3 byla kolize úspěšně vyřešena pouze v případě modelu *(1)* a *(2)*, v ostatních modelových situacích se agenti srazili.

2. **Úhybný manévr doprava/doleva/rovně** - Z důvodu neefektivnosti předchozí přístupu bylo chování agentů při řešení kolize změněno tak, aby byl agent schopen provádět úhybný manévr nejen doleva, ale také doprava nebo rovně. V tomto typu řešení používá agent k výběru vhodného úhybného manévru informace o pozici kolizního objektu v detekční výseči. Tato informace je vložena prostředím do báze znalostí agenta v podobě predikátu *sector*, který byl popsán v Podkapitole 5.2.1. V Tabulce 5.1 je shrnuto použití úhybných manévrů v tomto přístupu.

Místo detekce kolizních objektů	Predikát z prostředí	Úhybný manévr
Levá podvýseč	*sector(left)*	Otočka doprava
Pravá podvýseč	*sector(right)*	Otočka doleva
Levá i pravá podvýseč	*sector(left)*, *sector(right)*	Let rovně

Tabulka 5.1: Úhybný manévr v závislosti na pozici kolizního objektu.

Toto řešení eliminuje riziko srážky tří agentů, ale zvyšuje riziko srážky v případě řešení kritické kolize dvou agentů. Na příkladu modelových situací na Obrázku 5.3 jsou tímto přístupem úspěšně vyřešeny kolize na modelech *(1)* a *(3)*.



Obrázek 5.4: Význam detekce kritické kolize.

Na Obrázku 5.4 je znázorněno řešení kolize mezi dvěma agenty spolu s trajektorií jejich letu. Na prvním obrázku zleva je zobrazen startovní a cílový bod obou agentů. Prostřední obrázek zobrazuje způsob řešení vzniklé kolize druhým přístupem, tedy bez ošetření kritické kolize.

Agenti se v této modelové situaci vzájemně detekují na hranici svých detekčních výsečí. Agent R1 řeší tuto kolizi otočkou doleva a agent R2 otočkou doprava. Po provedení úhybného manévru o jeden stupeň se agent R1 dostane mimo detekční výseč agenta R2 a naopak. V dalším kroku se tito agenti vzájemně nedetekují a na základě polohy svých cílových bodů C1 a C2 pokračují v letu do cílového bodu. Agent R1 provádí otočku doprava a agent R2 otočku doleva. Tím se agenti opět dostanou jeden druhému do detekční výseče a situace se opakuje. Agenti se k sobě tímto způsobem přibližují tak dlouho, dokud se nesrazí. Obrázek 5.4 vpravo ukazuje řešení této kolize třetím přístupem, který kritickou kolizi zohledňuje.

3. **Rozšíření o detekci kritické kolize** - Tento mechanismus řešení rozšiřuje předchozí přístup o detekci kritické kolize. Kritická kolize je taková kolize, kdy agent detekuje kolizní objekt ve své levé podvýseči a zároveň tento objekt detekuje agenta ve své pravé podvýseči nebo naopak. Agent je o kritické kolizi informován z prostředí tak, jak bylo popsáno v Podkapitole 5.2.1. Řešení takové kolize probíhá vždy úhybným manévrem doleva bez ohledu na to, v které části detekční podvýseče se kolizní objekt nachází. Na Obrázku 5.4 vpravo je řešení kritické kolize znázorněno spolu s trajektorií pohybu agentů.

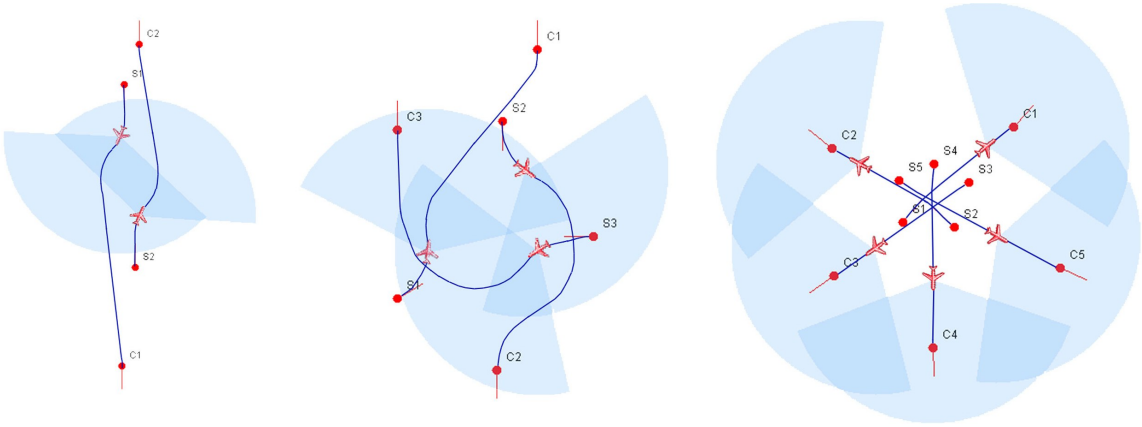
   Použití tohoto přístupu v simulátoru zvýšilo úspěšnost řešení kolize především v případě dvou agentů. Při použití tohoto mechanismu se agenti úspěšně vyhnou ve všech modelových situacích kromě situace na modelu *(4)* (Obrázek 5.3).

4. **Přidání třetí dimenze** - Poslední testovaný přístup k řešení kolizí rozšiřuje předchozí přístup o další (třetí) dimenzi v podobě letové hladiny, která je přidána každému agentovi. Výchozí výškou letu agenta je 11000 metrů. Každý agent při detekci kolize provede kontrolu, zda se některý z kolizních objektů nenachází v kritické vzdálenosti, která je empiricky stanovena na 250 metrů (50 vzdálenostních jednotek na radaru simulátoru). Tato kontrola probíhá pomocí interní akce *internal.isCriticalDistance*. V případě, že se k sobě agenti přiblíží na kritickou vzdálenost, pak agent v kolizi, který je ve vnitřní struktuře aplikace reprezentován nejnižším identifikačním číslem, je stanoven řešitelem této kolize.

   Řešitel si do své báze znalostí přidává pomocné predikáty určující počet kolizních agentů a jména těchto agentů z důvodu, aby byl schopen tyto agenty informovat o tom, jak mají změnit výšku tak, aby se vyhnuli kolizi. Informování kolizních agentů o změně výšky probíhá pomocí interní akce *.send(E,achieve,heightChange(F))* z prostředí Jason. Symbol *E* zde reprezentuje jméno kolizního agenta a *heightChange(F)* reprezentuje cíl, kterého má kolizní agent dosáhnout.

   Každý kolizní agent, který tento pokyn dostane, provede interní akci *internal.goDown(F)*, kde symbol *F* značí, o jaký násobek hodnoty 300 metrů má agent změnit svou výšku. Hodnota 300 metrů je zvolena jako bezpečný vertikální rozestup letadel ve vzdušném prostoru.

   Při použití tohoto mechanismu agenti úspěšně vyřeší kolizi i v případě modelu *(4)* na Obrázku 5.3. V hlavním okně aplikace je uživatel o změně výšky agenta informován na informačním panelu zobrazením šipky, která ukazuje, jakým směrem agent změnil svou výšku. Po vyřešení kolize agenti kontrolují, zda se v kritické vzdálenosti nachází nějaký kolizní objekt. Pokud ne, pak se agent vrací na původní výšku 11000 metrů. I přes zavedení možnosti změny výšky agentů se nemusí podařit kolizi vyřešit. Pokud se letadla za letu srazí, pak je jejich poslední pozice označena na radaru křížkem.



Obrázek 5.5: Příklady řešení kolizí pomocí simulátoru.

Na Obrázku 5.5 jsou znázorněny trajektorie letadel při řešení kolizí v simulátoru. Úhybný manévr potřebný k řešení kolize probíhá pouze po dobu rizika této kolize. Jakmile riziko pomine, agent okamžitě přepočítává svou trasu a pokračuje v letu do cílového, případně průletového bodu.

5.3 Pronásledování

Pronásledování je možné v případě, že se v detekční výseči agenta nachází pouze jeden kolizní objekt a tento objekt nepatří do stejného týmu jako agent, který ho detekoval. Agent je o možnosti pronásledování informován z prostředí pomocí predikátu *enviro(attack,E)*, kde *E* reprezentuje jméno agenta, kterého je možné pronásledovat. Pokud agent detekuje tento predikát ve své bázi znalostí, tak je volána interní akce *internal.isPersecutionPossible*, na jejímž základě agent rozhodne, zda je pronásledování možné či nikoliv.

Pronásledování je možné za předpokladu, že detekovaný kolizní objekt nemá ve své detekční výseči agenta, který ho detekoval. Tím je zajištěno, že nenastane situace, kdy by se dva agenti pronásledovali navzájem. Taková situace by skončila srážkou. V případě, že se agent i kolizní objekt detekují navzájem, pak i přesto, že každý patří do jiného týmu, je upuštěno od pronásledování a kolize je řešena vyhnutím se.

V simulátoru bylo experimentováno se dvěma přístupy k určení pronásledovacího manévru:

1. **Výpočet trasy k nepříteli** - Tento přístup je založen na určení optimální trasy do bodu, ve kterém se kolizní objekt (nepřítel) naposledy nacházel. Výpočet vychází z Dubinsových křivek, viz Podkapitola 4.6. Ze získané sekvence kroků definujících optimální cestu je vždy provedena pouze jedna iterace prvního nenulového kroku. Příklad sekvence kroků:

   LEFT(50) – **RIGHT**(25) – **LEFT**(68)

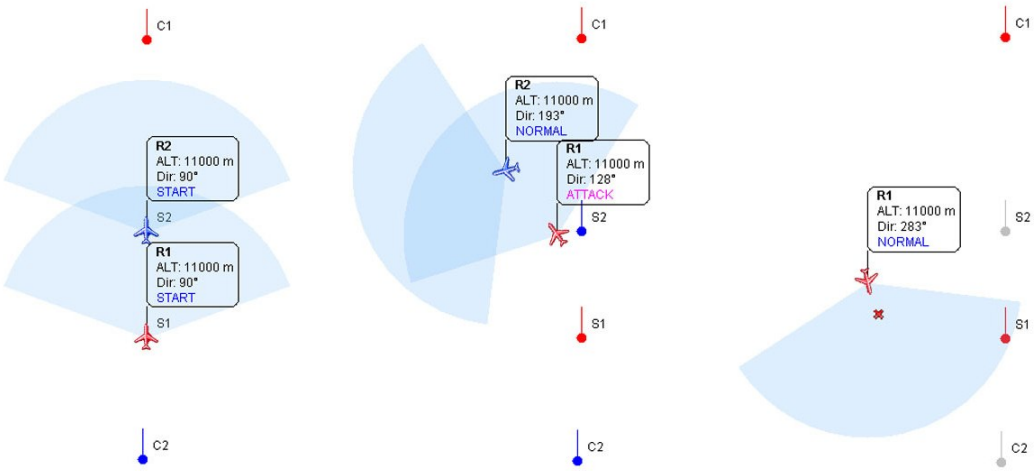
   V uvedeném příkladě, kde *LEFT* značí otočku doleva o 50°, příp. 68° a *RIGHT* otočku doprava o 25°, je agentem provedena otočka doleva o 1°. Pokud je po provedení tohoto manévru pronásledování stále možné (kolizní objekt je stále v detekční výseči agenta), pak je znovu přepočítána optimální trasa k nepříteli.

   Použití Dubinsových křivek pro určení pronásledovacího manévru nebylo ideální. Mohla zde nastat situace, kdy se pronásledovatel od svého nepřítele vzdaloval, tím ho ztratil ze své detekční výseče a pronásledování tak bylo neúspěšné.

2. **Zkracování vzdálenosti** - Pronásledovací manévr, který agent použije, závisí v tomto přístupu na pozici kolizního objektu (nepřítele) v detekční výseči agenta. V případě, že se nepřítel nachází v levé podvýseči agenta, pak je provedena otočka doleva o 1° a naopak. Agent tento pronásledovací krok získává pomocí interní akce. Princip výběru pronásledovacího manévru je shrnut v Tabulce 5.2.

Místo detekce kolizního objektu	Predikát z prostředí	Pronásledovací manévr
Levá podvýseč	*sector(left)*	Otočka doleva
Pravá podvýseč	*sector(right)*	Otočka doprava

Tabulka 5.2: Pronásledovací manévr v závislosti na pozici pronásledovaného agenta.



Obrázek 5.6: Pronásledování a sestřelení nepřátelského agenta.

Na Obrázku 5.6 je tento přístup k pronásledování znázorněn graficky. Při použití tohoto přístupu nedochází ke ztrátě nepřítele z detekční výseče agenta vlivem výběru špatného pronásledovacího manévru, jako tomu bylo v předcházejícím přístupu. V tomto případě má pronásledovatel tendenci se k nepříteli přibližovat.

Ztráta nepřítele z detekční výseče ovšem může nastat v případě, kdy pronásledovaný agent má mnohem vyšší rychlost letu než jeho pronásledovatel. Pokud pronásledovatel ztratí nepřítele z detekční výseče, pak přepočítává trasu do cílového, případně průletového bodu a pokračuje v letu. Pronásledovatel průběžně kontroluje vzdálenost mezi sebou a kolizním objektem pomocí interní akce *internal.isAttackPossible*.

Pokud se pronásledovatel k nepříteli přiblíží na vzdálenost menší nebo rovnou 200 metrům (40 vzdálenostních jednotek na radaru) a zároveň se tento nepřítel nachází v detekční výseči přímo před pronásledovatelem, pak pronásledovatel může tohoto nepřítele sestřelit. Sestřelený agent je na radaru označen křížkem a jeho běh v simulátoru je ukončen.

Kapitola 6

Závěr

Cílem práce bylo vytvořit letecký simulátor v multi-agentním systému Jason. K této problematice byly nastudovány příslušné materiály, ze kterých bylo čerpáno při tvorbě teoretické a praktické části práce. Praktická část práce je rozdělena na simulační modul založený na systému Jason a uživatelské rozhraní. Jednotlivá letadla jsou v simulátoru reprezentována agenty a jejich chování je popsáno plány v jazyce AgentSpeak. Agenti mají implementovány tři druhy chování. Mezi toto chování patří schopnost zaujímat pozice v průletových a cílovém bodě, detekce a následné řešení kolizí a pronásledování.

Uživatelské rozhraní aplikace umožňuje vykonávání dvou akcí a to spouštění existujícího simulačního modelu a vytváření nových simulačních modelů pomocí editoru. V simulátoru bylo experimentováno se čtyřmi přístupy k řešení kolizí. Za účelem co nejvyššího počtu úspěšně vyřešených kolizí byl v simulátoru implementován přístup zohledňující výšku letu agenta. Díky tomu jsou agenti schopni řešit i kolize o více kolizních objektech. I přes zavedení možnosti změny výšky agenta není řešení kolize vždy úspěšné. Důležitým faktorem ovlivňujícím úspěšnost je vzdálenost mezi agenty při detekování kolize a počet detekovaných kolizních objektů. V simulátoru bylo testováno řešení kolizí agentů na modelech s nejvýše pěti agenty. V simulátoru byly dále testovány dva přístupy k pronásledování.

Tuto práci by bylo možné rozšířit o další modely chování agentů jako například vytváření leteckých formací. Z hlediska inteligence agentů by bylo možné práci rozšířit o přidání schopnosti agenta měnit letovou rychlost za účelem řešení kolizí a pronásledování.

Literatura

[1] *FIPA Communicative Act Library Specification*. IEEE Computer Society standards organization, [cit. 10.1.2014].
URL http://www.fipa.org/specs/fipa00037/SC00037J.pdf

[2] Agent Communication Languages: The Current Landscape. *IEEE Intelligent Systems*, ročník 14, č. 2, 1999: s. 45–52, ISSN 1541-1672.

[3] *FIPA Agent Communication Language*. IEEE Computer Society standards organization, 2002, [cit. 10.1.2014].
URL http://www.fipa.org/specs/fipa00061

[4] *FIPA (Foundation for Intelligent Physical Agents)*. IEEE Computer Society standards organization, 2008, [cit. 10.1.2014].
URL http://www.fipa.org

[5] AirTrafficAtlanta.com: Radar. [online], [cit. 20.2.2014].
URL http://airtrafficatlanta.com/radar.php?radar=1

[6] Bordini, R. H.; Hübner, J. F.; Wooldridge, M.: *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley, 2007, iSBN 978-0-470-02900-8.

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

[8] Hübner, J. F.; Bordini, R. H.: Jason: a Java-based interpreter for an extended version of AgentSpeak. [online], 2004, [cit. 11.1.2014].
URL http://jason.sourceforge.net

[9] Huget, M.-P.: *Communication in Multiagent Systems*. Springer-Verlag Berlin Heidelberg, 2003, iSBN 3-540-40385-2.

[10] LaValle, S. M.: *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006, online at http://planning.cs.uiuc.edu/.

[11] Rao, A. S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world*, Springer-Verlag New York, Inc., 1996, ISBN 3-540-60852-4, s. 42–55.

[12] Shkel, A. M.; Lumelsky, V. J.: Classification of the Dubins set. *Robotics and Autonomous Systems*, 2001: s. 179–202.

[13] Wooldridge, M.: BDI Agent Programming with AgentSpeak. [online], [cit. 11.1.2014]. URL http://www.cs.ox.ac.uk/people/michael.wooldridge/teaching/robotics/agentspeak.pdf

[14] Zbořil, F.: *Plánování a komunikace v multiagentních systémech*. Disertační práce, Fakulta informačních technologií, VUT, Brno, 2004.

[15] Šťastný, P.: *Multiagentní systémy v medicíně*. Diplomová práce, Fakulta elektrotechnická, ČVUT, Praha, 2007.

Příloha A

Obsah CD

Přiložený CD disk obsahuje všechny zdrojové kódy a kódy třetích stran potřebné pro spuštění aplikace. Přesný výčet stromové struktury je uveden níže:

- **doc/** dokumentace ke zdrojovým kódům
- **examples/** příklady konfiguračních souborů se simulačními modely
- **jar/** spustitelná verze aplikace
- **jason/** knihovna systému Jason ve verzi 1.4.0a
- **pdf/** výsledný PDF dokument této práce
- **resources/asl** zdrojové soubory specifikující chování agentů
- **resources/mas2j** zdrojový soubor specifikující umístění zdrojů při vykonávání simulace
- **src/** zdrojové soubory praktické části práce
- **tex/** zdrojové soubory pro LaTeX potřebné k vygenerování výsledného PDF souboru
- **zip/** archiv projektu s příponou *.zip* pro import do NetBeans IDE (verze 7.3.1) (primárně Windows)

Postup spuštění aplikace spolu s licenčním ujednáním k použitému softwaru je uveden v souboru **README.txt** v kořenové složce.

Příloha B

Příklad konfiguračního souboru

Na ukázce je uveden příklad obsahu konfiguračního souboru. V tomto případě se jedná o simulační model se dvěma agenty, z nichž každý má definováno jméno, rychlost, barvu týmu a letový plán sestávající se z startovního a cílového bodu, případně libovolného množství průletových bodů.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<FlightModel>
    <AgentList>
        <Agent>
            <name>Agent 1</name>
            <speed>700</speed>
            <team>red</team>
            <flightPlan>
                <startConfig dir="42" x="291.0" y="181.0"/>
                <endConfig dir="349" x="659.0" y="431.0"/>
                <flybyPoints/>
            </flightPlan>
        </Agent>
        <Agent>
            <name>Agent 2</name>
            <speed>850</speed>
            <team>red</team>
            <flightPlan>
                <startConfig dir="270" x="505.0" y="463.0"/>
                <endConfig dir="355" x="885.0" y="195.0"/>
                <flybyPoints>
                    <Point dir="310" x="585.0" y="198.0"/>
                    <Point dir="31" x="820.0" y="477.0"/>
                </flybyPoints>
            </flightPlan>
        </Agent>
    </AgentList>
</FlightModel>
```