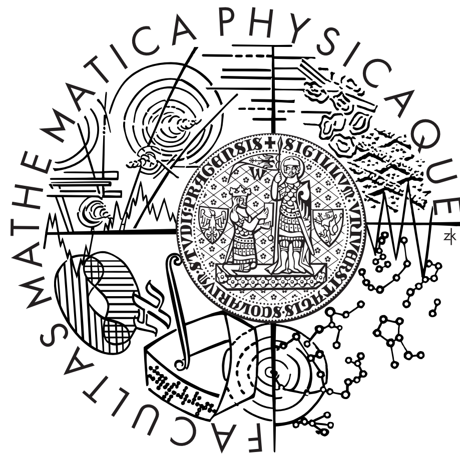
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Lukáš Zemčák

Pokročilé použitie ACT-R v Pogamute

Kabinet software a výuky informatiky

Vedoucí diplomové práce:	Mgr. Rudolf Kadlec
Studijní program:	Informatika
Studijní obor:	Teoretická Informatika

2013

Poděkování.

Vrelá vďaka patrí mojej manželke Lenke a mojim rodičom za podporu. Môjmu vedúcemu Rudovi za trpezlivosť a rady. A mojim deťom za čas o ktorý som ich touto prácou ukrátil.

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne .... Podpis autora

Název práce: Pokročilé použitie ACT-R v Pogamute

Autor: Bc. Lukáš Zemčák

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: Mgr. Rudolf Kadlec, Kabinet software a výuky informatiky

Abstrakt: Na virtuálnych agentov sú kladené čoraz náročnejšie požiadavky. Pre riadenie komplexného správania sa agenta je možné využiť kognitívne architektúry, ktoré vznikli na rozhraní neurovied a umelej inteligencie. Táto práca sa zaoberá knižnicou PoJACTR, ktorá prepája knižnicu Pogamut pre vývoj inteligentných agentov v hre UT2004. A knižnicu jACT-R, čo je implementácia jednej z popredných kognitívnych architektúr ACT-R v jazyku Java. Práca študuje vybrané problémy implementácie agentov v PoJACTR, navrhuje ako riešenie ladiace nástroje, ktoré boli následne implementované na platforme Eclipse IDE. Okrem toho rozširuje moduly knižnice PoJACTR o navigačný, komunikačný a modul pre hru Capture The Flag. Pre validáciu sa vyvinuli dvaja agenti (boti) hrajúci zmienenú hru, jeden v štandardnom Pogamute a jeden v PoJACTR. Pri súbojoch tímov mal PoJACTR bot porovnateľný výkon ako Pogamut bot. Výsledky ukázali, že ladiace nástroje uľahčili vývoj PoJACTR agentov.

Klíčová slova: virtuální agenti, kognitivní modelování, ACT-R, Pogamut

Title: Advanced use of ACT-R in Pogamut

Author: Bc. Lukáš Zemčák

Department: Department of software and computer science education

Supervisor: Mgr. Rudolf Kadlec, Department of software and computer science education

Abstract: The requirements for virtual agents are more and more demanding. In order to manage the complex behavior of the agent, it's possible to take advantage of cognitive architectures which arised on the field neuroscience and artificial intelligence. This work examines PoJACTR library which links Pogamut library for developing intelligent agents in Unreal Tournament 2004 and jACT-R library which is Java implementation of one of the leading cognitive architectures ACT-R. This work also studies certain agent implementation problems in PoJACTR and proposes a solution for them in form of debugging tools, which were subsequently implemented on an Eclipse IDE platform. In addition, it expands PoJACTR navigation and communication library modules for the game - Capture The Flag. As a validation, two agents (bots) were developed to play game, one in standard Pogamut and one in PoJACTR. When matched against each other in battle, PoJACTR bot had comparable performance to a Pogamut bot. The results showed that debugging tools facilitated development process of PoJACTR agents.

Keywords: virtual agents, cognitive modeling, ACT-R, Pogamut

Obsah

1 Úvod **2**

2 Štruktúra práce **4**

3 Oblasť práce **5**
- 3.1 ACT-R . . . 5
  - 3.1.1 Čo je to kognitívna architektúra . . . 5
  - 3.1.2 Modularita mozgu a Fodorova modulárna architektúra . . . 6
  - 3.1.3 Základný slovník ACT-R . . . 7
  - 3.1.4 Architektúra ACT-R . . . 7
  - 3.1.5 jACT-R . . . 9
- 3.2 Prečo používať kognitívne architektúry . . . 10
- 3.3 Pogamut . . . 10
- 3.4 UT2004 a Capture The Flag . . . 11
- 3.5 Eclipse . . . 11
  - 3.5.1 jACT-R plugin . . . 11

4 Ciele a metodika **14**
- 4.1 Ciele . . . 14
- 4.2 Validácia . . . 15
- 4.3 Metodika porovnania . . . 15

5 Príbuzné práce **17**
- 5.1 Syntetický nepriatelia pri tréningu MOUT . . . 17
- 5.2 SOAR . . . 17
- 5.3 CTF . . . 19
- 5.4 Zhrnutie . . . 20

6 Vývoj botov v plugine PoJACTRv1 **21**
- 6.1 Implementácia PoJACTRv1 . . . 21
- 6.2 Implementácia modelu . . . 22
  - 6.2.1 Breakpointy a Java Debugger . . . 23
  - 6.2.2 Textové logovanie . . . 23
  - 6.2.3 Ladenie Hunter bota . . . 23
- 6.3 Čo vývojár potrebuje . . . 24

7 Rozšírenia Eclipse IDE **25**
- 7.1 Platforma Eclipse . . . 25
- 7.2 Čo obsahuje rozšírenie PoJACTR pre Eclipse . . . 27
- 7.3 Editor pre jACT-R modely . . . 28
- 7.4 jACT-R Project . . . 28
- 7.5 Jadro pluginu . . . 28
- 7.6 Pogamut Manager . . . 29
- 7.7 PoJACTR JMXBot . . . 30
- 7.8 PoJACTR Log . . . 31
- 7.9 Conflict Set, Buffers state . . . 33

7.10 Logovanie PojACT-R . . . 34
7.11 PoJACTR Launch Configuration . . . 35
7.12 PoJACTR Perspective . . . 35
7.13 Zhrnutie . . . 36

8 CTF Boti v Jave **38**
8.1 Čo potrebuje jednoduchý CTF bot . . . 38
8.2 Implementácia . . . 38
8.3 Postup pre vývojára . . . 41

9 Navigačný modul pre PoJACTR **42**
9.1 Prečo navigovať . . . 42
9.2 Navigačný graf v UT2004 . . . 43
9.3 Pohyb po virtuálnom svete . . . 43
9.3.1 Plánovanie cesty . . . 43
9.3.2 Nasledovanie cesty . . . 44
9.4 Navigácia v ACTR . . . 44
9.5 Základy navigácie v Pogamute . . . 45
9.5.1 Plánovanie trasy . . . 45
9.5.2 Nasledovanie trasy . . . 46
9.6 Navigačný modul v PoJACTR . . . 46
9.6.1 Posielanie príkazov do motorických modulov . . . 46
9.6.2 Nastavenie cieľa navigácie . . . 47
9.6.3 RandomNavpointBuffer . . . 47
9.6.4 Napojenie Pogamut navigácie . . . 48
9.6.5 Využitie ladiacich nástrojov . . . 48

10 CTF Boti v PoJACTRv2 **50**
10.1 Využitie Eclipse pluginu . . . 50
10.2 CTF Modul . . . 52
10.3 Chat modul a epizodická pamäť . . . 54
10.3.1 Chat modul . . . 54
10.3.2 Epizodická pamäť . . . 54
10.4 CTF Bot . . . 55

11 Porovnanie implementácií botov **58**
11.1 Cieľ . . . 58
11.2 Metóda . . . 58
11.3 Výsledky . . . 58
11.4 Diskusia . . . 58

12 Budúca práca **61**

13 Záver **62**

Zoznam použitej literatúry **63**

1. Úvod

Riadenie virtuálnych agentov napreduje míľovými krokmi. Už dávno nestačí reaktívne riadenie ani jednoduchý plánovač. Do tohto oboru prenikajú čoraz častejšie vedomosti z kognitívnej psychológie (napríklad MOUT [15] alebo SOAR [14]). Je to oblasť psychológie, ktorá sa zaoberá poznávacími procesmi ľudskej mysle, ako sú vnímanie, myslenie alebo učenie. Tieto poznatky pomáhajú pochopiť, akým spôsobom je mozog organizovaný tak, že dovoľuje vedomie.

Pre komplexné skúmanie, nestačí opísať jednotlivé kognitívne procesy, dôležitou otázkou je, ako ich pospájať tak, aby vznikol koherentný model celej ľudskej mysle. Odpoveďou na túto otázku sa snažia byť kognitívne architektúry. Tie sa pokúšajú definovať nielen výsledné správanie agenta a jednotlivé subsystémy ale aj vnútornú štruktúru a organizáciu systému riadenia. Kognitívne architektúry sú okrem psychológie výrazne ovplyvnené aj neurológiou a nezriedka mapujú svoje súčasti na štruktúry centrálnej nervovej sústavy. Jednou z takýchto architektúr je Adaptive Control of Thought-Rational (ACT-R [12]), ktorou sa zaoberá táto práca. ACT-R je motivovaná výsledkami neurobiológie v oblasti skúmania mozgu pomocou funkčnej magnetickej rezonancie. Snaží sa preto mapovať okrem štruktúr mozgu na moduly, aj aktivitu modulov na aktivitu jednotlivých mozgových štruktúr.

Vývoj takýchto modelov vyžaduje iný ako klasický prístup k vývoju umelej inteligencie. V tomto prípade sa výskum zaoberá najmä paralelami k ľudskej kognícii a samotný výkon agenta nie je až tak dôležitý. Príkladom môže byť úloha bludiska: dôležitý nie je iba čas nájdenia cieľa, ale hlavne cesta k nemu a spôsob, akým sa agent k výsledku dopracoval.

Všetky modely riadenia agentov, vrátane kognitívnych architektúr, potrebujú pre svoju examináciu virtuálne prostredie. Výskumník ma vtedy tri možnosti. Buď si naprogramuje prostredie na mieru, a získa jednorázovú, často neprenositeľnú a neporovnateľnú implementáciu modelu. Navyše pri komplikovanejších virtuálnych prostrediach, ako sú 3D svety s reálnou fyzikou, je táto možnosť časovo veľmi náročná. Preto je výhodnejšia druhá možnosť a to použiť existujúci virtuálny svet, a implementovať iba prepojenie modelu a sveta. Treťou naideálnejšou možnosťou je použiť platformu pre vývoj virtuálnych agentov, ktorá obsahuje virtuálny svet a má prepojenie už naiplementované. Takouto platformou je Pogamut[2][3][4], ktorý používa ako virtuálny svet hru Unreal Tournament 2004 (UT2004). Táto platforma obsahuje aj plugin PoJACTR určený pre vývoj modelov ACT-R.

Pre zložitejšie kognitívne architektúry platia rovnaké zákonitosti ako pre akýkoľvek iný zložitý software. S náročnosťou implementácie stúpa potreba nástrojov, ktoré ju uľahčujú. Medzi užitočné nástroje na vývoj patria obzvlášť nástroje na sledovanie stavu agenta, ladenie chýb a validáciu agentov. Je obrovský rozdiel hľadať chybu v tisícoch riadkov ako v prehľadnej tabuľke. Takéto nástroje by nemali chýbať v žiadnej dobrej platforme pre vývoj virtuálnych agentov.

Ďalšia výhoda platformy pre vývoj kognitívnych architektúr plynie z ich modulárnosti. Ich jednotlivé moduly by mali byť jednoducho zameniteľné a znovupoužiteľné. To naznačuje, že by bolo pre vývojárov výhodné, ak by mali čo najviac všeobecných modulov už predimplementovaných. Následne by výskumník pri skúmaní epizodickej pamäte nestrácal čas implementovaním navigácie a naopak.

Využitie takýchto nástrojov môže výrazne skrátiť čas na implementáciu a examináciu modelu. Ak sú ešte nástroje pokope v jednom integrovanom vývojovom prostredí, výskumník sa môže viac venovať práci na samotnom modely. Príkladom takéhoto prostredia je Eclipse IDE.

Táto práca sa zaoberá náročnosťou vývoja ACT-R agentov na platforme Pogamut. Porovnáva vývoj agentov pomocou plugin PoJACTR[1] oproti klasickému vývoju. Zároveň sa zaoberá vylepšeniami základného pluginu PoJACTR[1] o vývojové nástroje, chýbajúce znovupoužiteľné základné moduly a integráciu s vývojovým prostredím Eclipse IDE.

[1]http://pogamut.cuni.cz/main/tiki-index.php

2. Štruktúra práce

Táto práca je rozdelená do nasledujúcich kapitol:

Kapitola 3: popisuje použité technológie (Pogamut, UT2004, Eclipse) a krátko predstavuje základ kognitívnej teórie ACT-R.

Kapitola 4: vytyčuje hlavný cieľ: implementáciu PoJACTRv2. Preberie jednotlivé úlohy pre jeho dosiahnutie. Okrem toho navrhne metodiku porovnania a testovania.

Kapitola 5: predstaví niekoľko prác s podobnou tematikou a porovná ich s touto prácou.

Kapitola 6: ukáže aké problémy už vyriešil PoJACTRv1 a ktoré otázky ponechal otvorené. Zároveň poukazuje na problémy vývoja agentov v PoJACTRv1 a analyzuje ako by sa dali vyriešiť.

Kapitola 7: ukazuje aké problémy sa riešil pri implementácii vývojových nástrojov PoJACTRv2

Kapitola 8: predstavuje problém, na ktorom táto práca porovnáva vývoj agentov a ukáže ako na tento problém vyvinúť agenta v štandardnom Pogamute.

Kapitola 9: zaoberá sa problémom navigácie, ukazuje ako ho rieši Pogamut a ako implementuje navigačný modul PoJACTRv2.

Kapitola 10: ukazuje ako CTF agentov vyvinúť v PoJACTRv2. Popisuje CTF modul pre PoJACTR, vývoj CTF bota pomocou Eclipse prostredia a ukazuje ako PoJACTR CTF bot funguje.

Kapitola 11: predstavuje testovanie výkonu a prezentuje výsledky experimentov.

Posledné dve kapitoly sa zaoberajú budúcou prácou na PoJACTR a zhrňujú dosiahnuté výsledky.

3. Oblasť práce

Táto práca sa zaoberá implementáciou novej verzie pluginu PoJACTR. Ten používa viacero technológii, ktoré nie sú triviálne a je preto potrebné predstaviť aspoň ich základy.

3.1 ACT-R

Adaptive Control of Thought-Rational je kognitívna architektúra(KA), ktorej základy položil J.R. Anderson viac ako pred 30 rokmi. Za ten čas evolvovala do komplikovanej a komplexnej teórie ľudskej mysle. PoJACTR slúži práve na vytváranie modelov tejto teórie, preto by bez znalosti jej základov zvyšok textu dával len malý zmysel. Cieľom tejto podkapitoly nie je opísať detailne celú teóriu, ale ponúknuť základ potrebný pre pochopenie pluginu PoJACTR. Väčšina informácií pochádza z [12],[13], [11],[10], kde je možné nájsť detailnejší opis biologického základu a ACT-R.

3.1.1 Čo je to kognitívna architektúra

Je nemožné dobre definovať KA jednou vetou. Vo všeobecnosti sa KA snaží prepojiť behaviorálnu a štrukturálnu stránku vedomia. Na jednej strane stojí správanie agenta so všetkými psychologickými aspektami mysle a na druhej komplexita centrálneho nervového systému. Autor teórie ACT-R J.R. Anderson KA definuje pre túto snahu nasledovne[13]:

> Kognitívna architektúra je špecifikácia štruktúry mozgu takej abstrakcie, aby vysvetľovala ako dosahuje funkcie mysle.

Inými slovami, KA je širokospektrálny výpočtový kognitívny model, ktorý pokrýva dôležité štruktúry a procesy mysle. Používa sa najmä na niekoľkoúrovňovú analýzu správania. KA vznikli ako odpoveď na otázku či ľudská kognícia pracuje symbolicky alebo subsymbolicky. Táto otázka je stará viac ako 30 rokov a v princípe táto otázka rieši, z akým typom informácie mozog pracuje na svojich základných úrovniach:

Symbolicky: Používa symboly a ich vzťahy medzi nimi ako reprezentáciu znalostí, a následne symboly spracováva. Pracuje sériovo. Zaoberá sa hlavne spracovaním už existujúcich znakov.

Subsymbolicky: Ukladá informácie distribuovane a nejakým spôsobom vie takúto informáciu spracovávať. Pracuje paralelne. Zaoberá sa hlavne rozpoznávaním senzorických vstupov

Napriek tomu, že tieto dva prístupy sú značne rozdielne, podľa [12] sú to dve tváre toho istého problému. ACT-R konkrétne tvrdí, že ľudská kognícia funguje subsymbolicky na senzorovo-motorickej úrovni, a existuje centrálne riadenie

pracujúce so symbolmi. Preto sa ACT-R zaraďuje k takzvaným hybridným kognitívnym architektúram.

Zprvoti sa KA snažili vysvetliť len konkrétne aspekty ľudskej kognície ako je učenie alebo riešenie problémov. Moderné KA sa snažia vytvoriť čo najkoherentnejší a najplauzibilnejší model celej mysle.

3.1.2 Modularita mozgu a Fodorova modulárna architektúra

Človek žije v dynamickom svete plnom podnetov, ktoré sú simultánne predkladané mozgu, a ten musí zmysluplne reagovať. Podľa [10] kognitívne funkcie mozgu rozdeliť na

Percepčné (zmyslové): Človek musí reagovať na vonkajšie podnety, filtrovať tie nedôležité (cvrlikanie svrčka) a naopak urýchlene reagovať na tie podstatné(padajúci strom).

Motorické (svalové, manuálne): Človek musí vedieť vykonávať rozličné činnosti. Niektoré paralelne (ľubovoľný šport) a niektoré reflexne(vyhnutie sa padajúcemu stromu).

Kontrolné (koordinačné): Človek dokáže riešiť komplexné úlohy, akými je napríklad operácia srdcovej chlopne. Preto mozog musí vedieť koordinovať jednotlivé vnemy a činnosti, aby nasledovali v správnom poradí.

Práve koordináciou sa odlišuje človek od primátov, pretože percepčné a motorové schopnosti sú podobné(okrem schopnosti reči). ACT-R je vo veľkej miere inšpirovaná Fodorovou modulárnou architektúrou. Vo svojej knihe Modularity of Mind [11] sa Fodor zastáva názoru, že mozog je postavený z modulov, ktoré plnia vlastné funkcie. Táto myšlienka je inšpirovaná limitáciou mozgu: Spotreba energie pri prenos informácie medzi dvoma neurónmi je úmerná ich vzdialenosti. Preto sa predpokladá, že väčšina komunikácie je lokálna a to podporuje modularitu mozgu.

Pre ACT-R sú dôležité vlastnosti, ktoré Fodor pre modulárnu architektúru definuje. Anderson ich v zhrňuje nasledovne:

Špecificita domény: Modul spracováva iba obmedzenú doménu informácií.

Záväzné operácie: Človek nemôže zmeniť, akým spôsobom vníma svet, teda modul musí reagovať a ako reaguje nemôže byť vedome ovplyvnené.

Zapuzdrenie informácie: Modul spracováva väčšinu informácií interne a málo komunikuje s ostatnými modulmi.

Rýchle lokálne operácie: V rámci modulu sú kognitívne procesy najrýchlejšie.

Plytké výstupy: Rozhranie medzi modulmi je plytké, teda modul propaguje jednoduché fakty.

Fixovaná neurónová štruktúra: Každý modul je fixovaný na svoje dedikované neurónové štruktúry.

Jazyk: Existuje samostatný modul na spracovanie rečí.

Centrálna kognícia: Fodor tvrdil že mozog nemá centrum pre operáciu Modus Ponens. To by znamenalo, že neexistuje centrálne spracovanie. V tomto bode s ním ACT-R zásadne nesúhlasí. V ACT-R toto centrum reprezentuje procedurálny modul.

3.1.3 Základný slovník ACT-R

Chunk: Základná jednotka informácie s ktorou dokáže mozog pracovať ako s celkom.

Slot: Každý chunk sa skladá z niekoľkých slotov, v ktorých je uložená jednoduchá informácia. Napríklad slot *farba* má hodnotu *čierna*.

Buffer: Zásobník chunkov

Modul: Komponenta ACT-R, ktorá sa zaoberá spracovaním jedného druhy informácií. Každý modul má buffer, v ktorom sprístupňuje informácie iným modulom.

Model: Skupina modulov a ich bufferov, ktorá má napodobniť konkrétny aspekt ľudského správania.

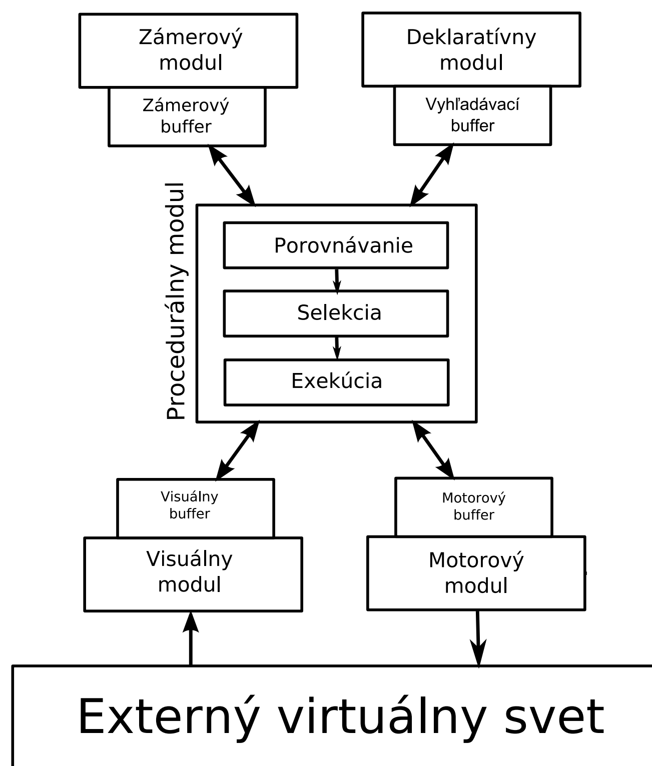
3.1.4 Architektúra ACT-R

ACT-R hybridná modulárna kognitívna architektúra definujúca niekoľko modulov a ich koordináciu. Snaží sa tieto modulu integrovať tak aby vytvorili koherentný kognitívny systém. Teória ďalej obsahuje definíciu niekoľkých modulov a ich namapovanie na oblasti ľudskej centrálnej nervovej sústavy. Súčasťou modulov sú buffere, ktoré ponúkajú informácie ostatným modulom v podobe chunkov. Centrálne riadenie zastupuje procedurálny modul, ktorý funguje ako produkčný systém. To znamená, že v každom kroku sa podľa stavu bufferov vyberie jedno produkčné pravidlo, ktoré sa vykoná (tento modul robí operáciu Modus Ponens). Pravidlo zase nastaví chunky v iných bufferoch. To sa dookola opakuje v 50ms cykloch[12][1][13]. Subsymbolizmus sa v ACT-R používa pre určenie konkrétneho pravidla pri nejednoznačnosti a interne v jednotlivých moduloch (napríklad rozpoznávanie vzorov v senzorických moduloch).

Na obrázku 3.1 sú zobrazené niektoré základné moduly ACT-R a štruktúra architektúry. Moduly pracujú jednotlivo a každý je principiálne schopný masívne paralelného spracovania. Navonok sa moduly prezentujú bufferami. Moduly okrem procedurálneho sa navzájom nevidia, a komunikujú výlučne cez procedurálny modul[2]. Ten pracuje v troch fázach:

[1] Odhadovaný čas, ktorý je potrebný pre multisynaptickú slučku v bazálnych gangliách - štruktúre mozgu ktorej procedurálny modul odpovedá.

[2] To nie je v ľudskom mozgu až tak samozrejmé a sú známe výnimky tohto pravidla. Napríklad chuťový vnem vyvolá zrakový ak ochutnáme jahody zo zavretými očami[10].



Obr. 3.1: ACT-R architektúra

1. **Podmieňovanie:** V tejto fáze procedurálny modul prehľadáva databázu produkčných pravidiel, a podľa stavu bufferov zisťuje ktoré pridlá môže použiť.

2. **Selekcia:** Druhá fáza slúži na výber jediného zo všetkých použiteľných pravidiel. ACT-R definuje spôsob akým zistiť takzvanú utilitu pravidiel, a podľa tejto utility sa snaží vybrať to najlepšie pravidlo.

3. **Exekúcia:** V poslednej fáze sa pravidlo vykoná a buffer sú aktualizované pre ďaľší cyklus.

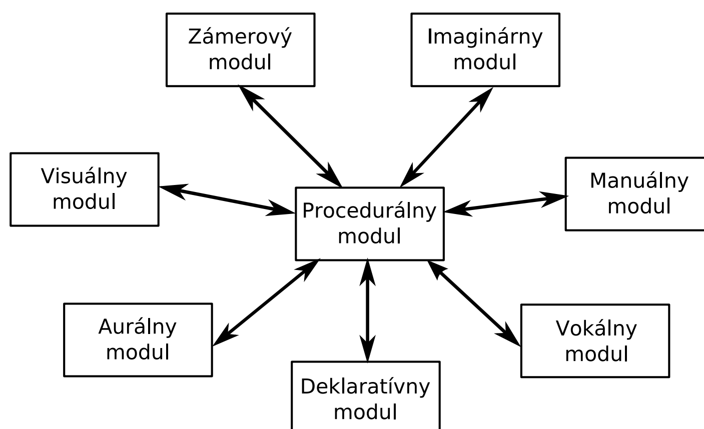
ACT-R okrem produkčného modulu definuje niekoľko interných, percepčných a motorových modulov (3.2):

Deklaratívny modul: je modul ktorý reprezentuje agentovu pamäť. Ukladá informácie typu: *Psi nemajú radi mačky.* ACT-R sa vo veľkej miere zaoberá subsymbolickými procesmi, ktoré vedú k vyvolaniu takejto informácie do buffera.

Zámerový modul: ukladá zámery agenta. Vďaka tomuto modulu agent dokáže vykonávať komplikovanejšie úlohy a nie len reaktívne reagovať

Imaginačný modul: predstavuje krátkodobú pamäť, ktorá je dôležitá pri riešení problémov (udržiava napríklad medzivýsledky pri počítaní)

Percepčné(senzorické) moduly: sú moduly spracovávajúce vonkajšie podnety, ktoré transformujú do chunkov. Dôležitou vlastnosťou týchto modulov je schopnosť filtrovať nedôležité vnemy a naopak zvýrazniť tie životne dôležité. Medzi percepčné moduly patrí hlavne vizuálny a aurálny modul.



Obr. 3.2: Moduly ACT-R

Motorové(pohybové) moduly: ovládajú reakcie agenta a to hlavne pohyb. Medzi motorické moduly patrí manuálny a vokálny modul.

3.1.5 jACT-R

ACT-R je štandardne distribuované so simulátorom[3], ktorý je napísaný v jazyku Lisp[4]. Pogamut je však implementovaný v jazyku Java, a preto plugin PoJACTR používa Java implementáciu ACT-R nazvanú jACT-R [5]. Autorom je Anthony M. Harrison a heslom *Making cognitive science portable (Urobme kognitívnu vedu prenosnú).*

Výhody a nevýhody tejto implementácie sa dajú zhrnúť do následných bodov:

- PoJACTR nepotreboval prepojenie Javy a Lispu, ktoré nie je jednoduché implementovať.
- jACT-R podporuje okrem jazyka Lisp aj podporu písania modelov v modernom jazyku XML.
- Z jazyka Java plynie implicitne typová bezpečnosť implementácie.
- jACT-R obsahuje plugin pre Eclipse IDE, čo výrazne pomohlo pri implementácii pluginu PoJACTR 2.
- jACT-R je stále relatívne nová a málo používaná implementácia, a preto obsahuje chyby a nie je ideálne zdokumentovaná.
- veľa ACT-R výskumníkov používa Lisp implementáciu, a preto je stále málo implementovaných modulov pre jACT-R.

[3]http://act-r.psy.cmu.edu/actr6/
[4] ACT-R 6.0 používa Common Lisp (http://common-lisp.net)
[5] http://www.jactr.org/

3.2 Prečo používať kognitívne architektúry

Napriek tomu, že pri vývoji agentov je jedným z dôležitých faktorov je, aby splnili úlohu na ktorú sú designovaný. Toto však nie je jediné kritérium podľa ktorého sa určuje dobrý agent. KA majú oproti iným prístupom niekoľko výhod:

KA su inšpirované ľudskou kogníciou, a to nielen behaviorálne, ale aj štrukturálne resp. neurologicky. To pomáha pochopiť ľudskú kogníciu, a posúva hranice toho čo vieme o ľudskom mozgu a mysli.

KA sú inteligentné systémy, ktoré sú kognitívne realistické (aspoň do určitej miery). Mali by byť viac *ľudské* ako iné systémy.

Pri riešení konkrétnej úlohy nemôže výskumník vyrobiť len modul, ktorý úlohu rieši, ale musí tento modul zakomponovať do existujúcej štruktúry KA. Samotný modul je kvôli tomu komplexnejší a teoreticky silnejší. To dáva modelom väčšiu robustnoť, a samotným modulom väčšiu vierohodnosť, pretože sú odskúšané v komplikovanejšom systéme.

Napriek všetkým výhodám, KA majú svoje obmedzenia. Často bývajú príliš komplikované, a vývoj jednoduchého modulu sa zmení na boj z komplexitou KA. Ďalšou nevýhodou KA je ich výkon, ktorý je práve kvôli komplexite, ale niekedy aj kvôli použitej platforme príliš nízky.

3.3 Pogamut

PoJACTR[6] je plugin do Pogamutu[2][3][4], ergo je na mieste predstaviť túto platformu. V čase písania diplomovej práce bola aktuálna verzia 3.5.1 [7]. Pogamut je platforma pre vývoj inteligentných umelých agentov, ktorá abstrahuje simulátor virtuálneho sveta. Platforma je stále vo výraznom vývoji a pozostáva z viacerých modulov, ktoré sa dajú rozdeliť do nasledujúcich kategórií:

1.Jadro: tvorí abstraktné rozhranie medzi konkrétnym virtuálnym svetom a riadením virtuálneho agenta. Jadro je nezávislé na použitom virtuálnom svete, teda principiálne (a aj prakticky) je možné pripojiť ho k rôznym simulátorom.

2.Pripojenia k virtuálnym svetom: v čase vzniku tejto práce sa boli podporované alebo vo vývoji pripojenia k Unreal Tournament 2004, Unreal Engine 2, Unreal Engine 3, DefCon. V praxi to znamená, že riadenie pre jeden virtuálny svet, sa dá upraviť pre iný podporovaný svet v relatívne krátkom čase.

3.Rozšírenia: medzi ne patria najmä rôzne typy riadení a modulov virtuálnych agentov, ako je napríklad POSH[9], genetic bots[5], StorySpeak[8] a aj PoJACTR[1].

4.IDE: existuje integrácia Pogamutu s vývojovým prostredím Netbeans, ktorá pomáha pri vývoji a ladení.

[6] PoJACTR bol implementovaný v rámci bakalárskej práce [1]
[7] http://pogamut.cuni.cz

3.4 UT2004 a Capture The Flag

Knižnica PoJACTR využíva virtuálny svet UT, čo je akčná hra z pohľadu prvej osoby. Agenti-boti využívajú štandardné objekty ako sú zbrane, brnenia a lekárničky. Ako typ hry sa používa mód ”Ukoristi vlajku”, v angličtine Capture The Flag (ďalej len CTF). V tomto móde bojujú proti sebe dva tímy. Každý tím má základňu, v ktorej má svoju vlajku. Cieľom hry je ukoristiť nepriateľskú vlajku, a doniesť ju do svojej základne. Za to získava tím body. Vyhráva tím s maximálnym počtom bodov. Formalizovanejšie pravidlá pre UT botov by sa dali zhrnúť takto:

- Boti su rozdelený do dvoch tímov a každý bot patrí práve do jedného tímu.
- Každý tím má základňu v ktorej má na začiatku hry svoju vlajku.
- Bot môže zodvihnúť nepriateľskú vlajku tak, že cez ňu prebehne.
- Ak je zabitý bot, ktorý nesie vlajku, vlajka spadne na zem.
- Ak bot prejde cez svoju spadnutú vlajku, vlajka sa automaticky vracia na svoju základňu.
- Tím dostáva bod vtedy, keď má svoju vlajku vo svojej základni a donesie k nej aj nepriateľskú vlajku.
- Vyhráva tím ktorý dosiahne konkrétny počet bodov, alebo tím ktorý dosiahne väčší počet bodov v časovom limite.

3.5 Eclipse

Eclipse IDE je vývojové prostredie s rozširovateľným systémom pluginov, vďaka čomu je možné upraviť Eclipse pre množstvo použití[8]. Súčasť knižnice PoJACTRv2, ktorou sa zaoberá táto práca je aj plugin pre Eclipse, ktorý pomáha vo vývoji PoJACTR botov.

3.5.1 jACT-R plugin

Výhody rozšíriteľnosti IDE využíva naplno jACT-R, ktorý je síce samostatne spustiteľný (tak ho púšta PoJACTRv1), ale je distribuovaný práve s pluginom pre Eclipse. Ten pozostáva z mnoha častí, ktoré ich autor A.M. Harrison využíva pri rôznych experimentoch. Tieto jACT-R pluginy využíva a rozširuje práve PoJACTRv2. Porovnaním vývoja s nimi a bez nich sa zaoberá kapitola 7. Pre túto prácu sú najpodstatnejšie nasledujúce rozšírenia:

1. **jACT-R Run:** Eclipse IDE ponúka okrem vstavaných spúšťačov, možnosť vytvoriť si vlastný, nastaviteľný spúšťač. V ňom sa nastavuje aké modely sa majú spustiť, aké logovania spusti, prípadne nastavenie ďalších prídavných modulov. Knižnica jACT-R to využíva, aby dokázala spustiť ACTR model takým spôsobom, aby dokázal komunikovať s ostatnými Eclipse pluginmi.

[8]Existuje Eclipse pre jazyky C, C++, Fortran, Haskell, PHP, Python, Ruby a.t.ď

Obr. 3.3: UT2004: Pogamut bot hraje CTF

jACT-R Log | jACT-R Probes | 10:43

count-model

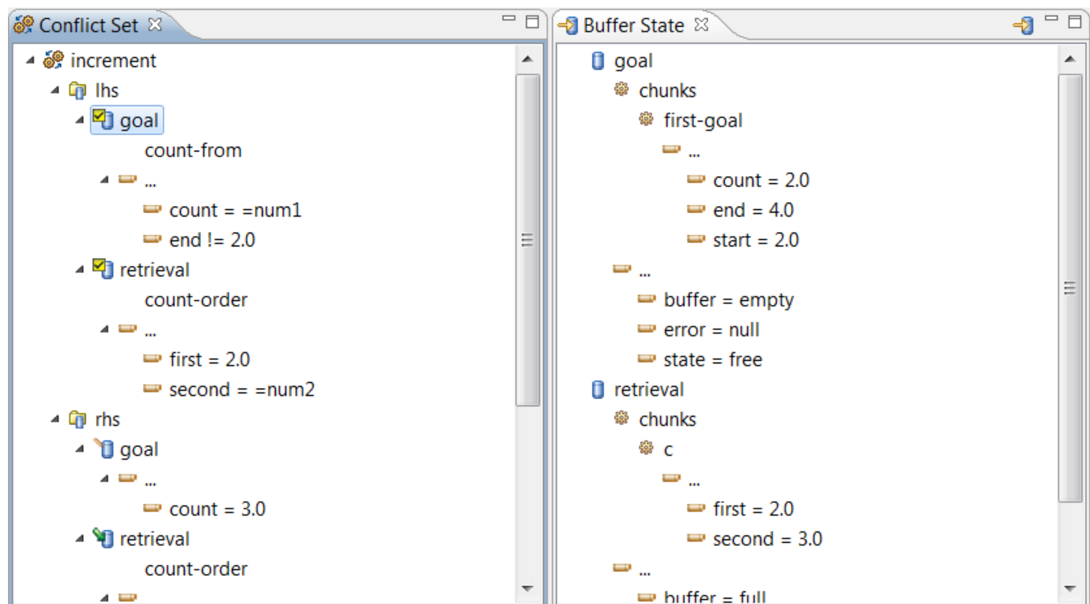
TIME	MARKERS	OUTPUT	GOAL	IMAGINAL	RETRIEVAL	PROCEDURAL	DECLARATI...	VISUAL	AURAL	MOTOR
00:00:00,00		Searching f...				Conflict Set [st...	Evaluating ...			
00:00:00,05						Conflict Set []				
00:00:00,10		2.0				Conflict Set [in...	Evaluating ...			
00:00:00,15						Conflict Set []				
00:00:00,20		3.0				Conflict Set [in...	Evaluating ...			
00:00:00,25		Answer 4.0				Conflict Set [st...				
00:00:00,30						Conflict Set []				
00:00:00,35						Conflict Set []				
00:00:03,00						Conflict Set []				

BUFFER : retrieval.buffer=empty (was full)
Removed c from retrieval
retrieval.state=busy (was free)
Added pattern [count-order:first = 3.0] to retrieval

DECLARATIVE : Evaluating exact search matches [d]
d has highest activation (0,00=0,00+0,00)

EVENT: Queued Modify(first-goal in goal @ 0,15)
Queued org.jactr.core.module.retrieval.six.DeclarativeFINSTManager$FINSTExpirationTimedEvent(@ 3,10)
Queued org.jactr.core.buffer.delegate.AsynchronousRequestDelegate$FinishRequestTimedEvent(@ 0,15)
Queued Output('2.0' @ 0,10)
Fired Output('2.0' @ 0,10)

Obr. 3.4: jACT-R Log okno. Hore vidíme logy podľa času a jednotlivých bufferov, zatiaľ čo dole sú kompletné logy pre daný časový okamih.



Obr. 3.5: Vľavo je Conflict Set, ktorý obsahuje produkčné pravidlá v konkrétnom časovom okamihu. Obdobne vpravo Buffer State zobrazuje obsah bufferov v tom istom okamihu.

2.jACT-R Log: Je plugin, ktorý zobrazuje štruktúrované logy ACT-R modelu (obrázok 3.4). Prehľadne ukazuje najdôležitejšie logy v čase.

3.Conflict Set: Zobrazuje produkčné pravidlá, ktoré sú aplikovateľné vo vybranom časovom okamihu (obrázok 3.5). Vďaka tomu vývojár presne vie, čo robil v danom cykle produkčný modul, a medzi akými pravidlami sa rozhodoval.

4.Buffer State: Zobrazuje obsah bufferov vo vybranom časovom okamihu. Táto funkčnosť je kritická z pohľadu ladenia PoJACTRu, pretože často sa stáva, že vyvíjaný modul nedáva do buffera tie chunky, ktoré by tam vývojár očakával. Napríklad ak vývojár zle namapoval senzorické objekty Pogamutu.

5.jACT-R Editor: je editor XML súborov, ktorý je optimalizovaný pre `.jactr` súbory. Okrem automatickej kontroly tagov ich parametrov obsahuje aj prehľadne stromovú štruktúru modelu.

6.jACT-R Project: je nový typ projektu, ktorý definuje štruktúru projektu potrebnú pre vývoj jACT-R modulov.

7.jACT-R Launch Configuration: je rozšírenie pomocou ktorého dokážeme konfigurovať a spustiť beh jACT-R modelu.

8.jACT-R Perspective: definuje počiatočné nastavenie okien a ich rozmiestnenia v Eclipse tak, aby čo najlepšie vyhovovali vývoju jACT-R modelov.

4. Ciele a metodika

Táto kapitola má za úlohu sumarizovať problémy ktoré táto práca rieši. ukázať úlohu na ktorej budú problémy ilustrované. Zároveň načrtneme metodiku testovania a porovnávania CTF botov.

4.1 Ciele

PoJACTRv1 bol iba prototypom prepojenia ACT-R a Pogamutu a preto zanechal množstvo otvorených veľa otázok. Hlavným cieľom bolo posunúť plugin na vyššiu úroveň. Preto bolo potrebné vybrať konkrétne otázky, ktoré by boli reálne splniteľné v rozsahu tejto práce. Od začiatku sa ponúkali dva smery: teoretický a implementačný.

Teoretický smer: V tomto prípade by sme vybrali jeden veľký problém, ktorý by práca teoreticky podložila a následne na príklade implementovala. Ponúkala sa hlavne plauzibilná navigácia. Problémom tohto prístupu bolo to, že väčšina času by musela byť venovaná vedeckému podloženiu problému. To by znamenalo, že by sme mali vedeckú validáciu, avšak samotný plugin by veľmi nepokročil.

Implementačný smer: Druhou možnosťou bolo venovať čo najviac času implementácii, urobiť PoJACTR čo najlepším nástrojom a nechať vedeckú validáciu pre ďalšie práce. Preto sa bolo treba zamyslieť nad tým, že PoJACTRu najviac chýba a aké sú jeho najzávažnejšie problémy. Zároveň bolo treba vybrať jednoduchší validačný problém.

Z praktického hľadiska bol výhodnejší implementačný smer. Hlavným dôvodom pre tento výber bol hlavný problém PoJACTRv1, a to logovanie, spracovanie logov a ladenie (tento problém je bližšie opísaný v kapitole 7). V plugine PoJACTRv1 bolo veľkým problémom odladiť fungovanie jednotlivých modulov, čo by robilo akúkoľvek väčšiu implementáciu prakticky nemožnou. Druhým dôvodom bola implementácia navigačného modulu, ktorý v PoJACTRv1 chýbal. Jeho nutnosť pre akýchkoľvek inteligentných agentov je vysvetlená v kapitole 8. Tretím dôvodom pre implementačný smer bolo množstvo drobných chýb pluginu, ktorý bol odskúšaný len na jedinom jednoduchom príklade a boli by prekážkou pri validácii akýchkoľvek vedeckých záverov.

Nová verzia by mala posunúť PoJACTR od prototypu k použiteľnej knižnici odskúšanej na reálnom príklade. A to nielen po stránke technickej (aby plugin fungoval ako má), ale najmä po stránke užívateľskej (aby uľahčil implementáciu naväzujúcim prácam).

Globálny cieľ implementovať plugin PoJACTRv2 môžeme rozdeliť do nasledujúcich čiastkových cieľov.

1. Identifikovať problémy PoJACTRv1
2. Navrhnúť riešenie týchto problémov
3. Implementovať tieto riešenia

- Ladiace nástroje
- Navigačný modul

4. Validácia navrhnutého riešenia

4.2 Validácia

Pre odskúšanie pluginu PoJACTRv2 bolo treba vybrať niekoľko scenárov, ktoré by zvalidovali kvalitu riešenia. Ako ideálne riešenie sa javilo implementovať vybraný problém a paralelne ho implementovať pomocou pluginu PoJACTRv2 a štandardne v jazyku Java. Následne sme porovnali postup pri implementácii a kvalitu jednotlivých riešení. Nakoniec sme vybrali problém *Capture the Flag*, ktorý nie je triviálny, preto sa na ňom výborne ilustruje použitie ladiacich nástrojov. Zároveň je pre jeho riešenie vitálna navigácia, čím sa odskúša navigačný modul. Jednotlivé body validácie môžeme zhrnúť nasledovne:

1. Navrhnúť a implementovať Java CTF botov.
2. Implementovať CTF agentov v PoJACTRv2.
3. Otestovať a porovnať rôzne implementácie CTF botov.

4.3 Metodika porovnania

Pre porovnanie prístupov využijeme dve diametrálne odlišné vlastnosti botov. Prvou je prístup k programovaniu botov. Opíšeme jednotlivé fázy programovania pre Java aj PoJACTR botov. Na príkladoch ukážeme aké prostriedky na ladenie sme využili. Keďže veľkou súčasťou práce sú PoJACTR ladiace nástroje, ukážeme množstvo príkladov, kedy boli pre implementáciu dôležité. Druhou vlastnosťou ktorú budeme porovnávať je výkonnosť CTF botov v reálnych zápasoch. Kým ladenie sa dá porovnať len subjektívne, pri porovnaní výkonu vieme využiť štatistiku. Preto sme púšťali rôznych agentov proti sebe na rôznych mapách. Využili sme štvoricu máp `Maul`, `Grendelkeep`, `Citadel` a `Magma` na ktorých sme spúšťali tímy po 2 a 3 agentoch.

Pre zaujímavosť sme do testovania výkonu pridali štandardných botov z UT2004 a prepracovanú implementáciu CTF priamo z Pogamutu. Aby sme dostali štatisticky relevantné výsledky, spúštali sme zápasy pre každú dvojicu botov. Každá dvojica bola spustená v 40 zápasoch, pričom v jednom zápase sa hralo o 5 bodov. Pre testovanie CTF botov bolo využité automatické spúšťanie a sledovanie zápasov, ktoré je súčasťou balíka Pogamut. Pogamut framework automaticky sledoval nielen výhry zápasu, ale aj jednotlivé body, počet zabití, samovrážd atď. My sme sledovali hlavne počet bodov jednotlivých tímov. Aby sme to mohli ďalej štatisticky skúmať, potrebovali sme previesť zápasy do náhodných veličín. Predpokladajme, že porovnávame výkon tímu A a tímu B. Základom náhodnej veličiny bude skórovanie bodu. Ak skóroval tím A hodnota bude 0, ak tím B hodnota bude 1. Ak budú tými rovnako výkonné, stredná hodnota náhodnej veličiny bude 0,5. Pod pojmom A je 3-krát lepší ako B definujeme ako A dalo 3-krát viac bodov, čo odpovedá strednej hodnote 0,75. Takto definované skórovanie sa správa podľa

binomického rozdelenia, preto môžeme na testovanie hypotéz použiť p-test s 95% intervalom spoľahlivosti[1].

Treba poznamenať, že sme pre jednoduchosť zanedbali vplyv mapy, rozdielne postavenie a rozličnú výzbroj medzi jednotlivými bodmi. Pre zaujímavosť sme do testovania výkonu pridali štandarných botov z UT2004 a prepracovanú implementáciu CTF priamo z Pogamutu, ktoré doplnili Java a PoJACTR implementáciu CTF botov.

[1]Pre výpočet sme použili voľne dostupný štatistický nástroj R (http://cran.r-project.org/bin/windows/base/)

5. Príbuzné práce

Táto kapitola načrtne podobné práce, ktoré sa zaoberajú kognitívnymi architektúrami v spojení z virtuálnymi svetmi. Rozoberie postupne použitie ACTR pre simuláciu MOUT, kognitívnu architektúru SOAR použitú s UT a iné projekty zaoberajúce sa vývojom agentov pre CTF.

5.1 Syntetický nepriatelia pri tréningu MOUT

Americký Námorný úrad pre výskum podporuje projekt modelovania vojakov v ACT-R, aby zistila validitu syntetických nepriateľov. Túto tému skúmala dvojica Bradley J. Best a Christian Lebiere na Carnegie Mellon University. Cieľom je vylepšiť efektivitu tréningu simulácií MOUT (Military operation on Urban Terrain, obrázok 5.1), čo sú vlastne simulácie vojenských operácií v mestských lokáciach. Pre simuláciu používajú virtuálny svet pôvodného Unrealu a pre modelovanie agentov používajú natívnu LISP implementáciu ACT-R.

V základnej simulácii popisovanej v [17] sa stretnú dva tými po dvoch vojakoch. Dvojica ktorá útočí reprezentuje vojakov USA, ktorý aplikujú štandarné postupy americkej armády. Naopak brániaca sa dvojica používa viacero stratégií, čo reflektuje varietu rôznych chovaní reálnych nepriateľov. V rámci výskumu sa mimoiné úspešne riešila aj komunikácia a práca v týme[18].

Oproti PoJACTR je MOUT zastaralý, používa starý virtuálny svet pôvodného Unreal Tournamentu z roku 1999.[1]. Signifikantným problémom je aj nemožnosť stiahnutia implementácie (predpokladáme, že je majetkom armády USA). To robí z MOUT zaujímavý projekt, ale nepoužiteľný pre ďalší vývoj.

5.2 SOAR

SOAR [19] je kognitívna architektúra založená John Laird, Allen Newell a Paul Rosenbloom na Carnegie Mellon University. V súčasnosti výskum vedie John Lairdova výskumná skupina na univerzite v Michigene. SOAR pozostáva z teórie a implementácie (mimoiné existuje aj verzia v jazyku Java, ktorý sa volá jSOAR[2]). Podobne ako ACT-R je založená na produkčnom systéme, ktorého architektúra ne načrtnutá na 5.2. V skratke pracuje s takzvanými operátormi, čo je alternatíva ku produkčným pravidlám ACT-R. V SOAR každý krok pozostáva z fáze spracovania v ktorej sa rôzne informácie o probléme dostávajú do pracovnej pamäte (obdoba bufferov ACT-R). Druhá fáza je rozhodovacia procedúra v ktorej sa ohodnotia výsledky predchádzajúcej fázy a podľa toho sa vyberie adekvátna akcia.

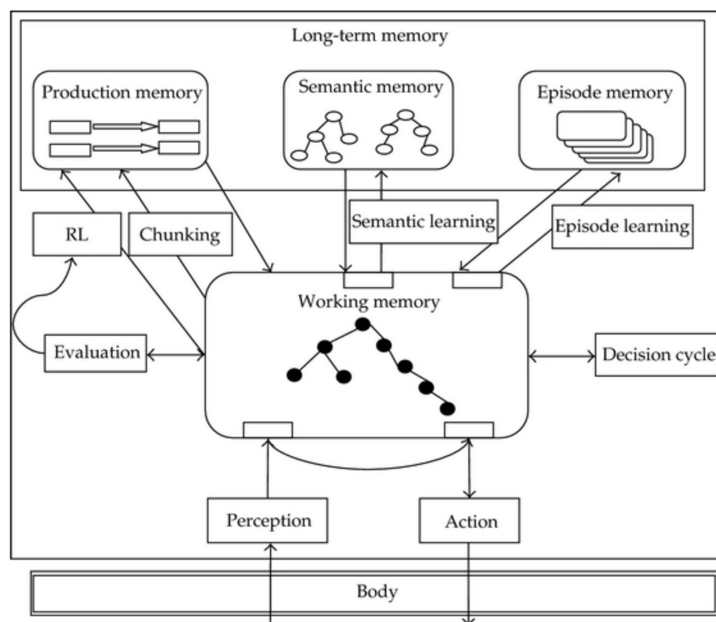
SOAR patrí k populárnym architektúram a bol prepojený mimoiné aj s UT2004. To je možné vďaka *Soar General Input/Output*[3], čo je adaptér ktorý prepája SO-

[1] `https://en.wikipedia.org/wiki/Unreal_Engine`

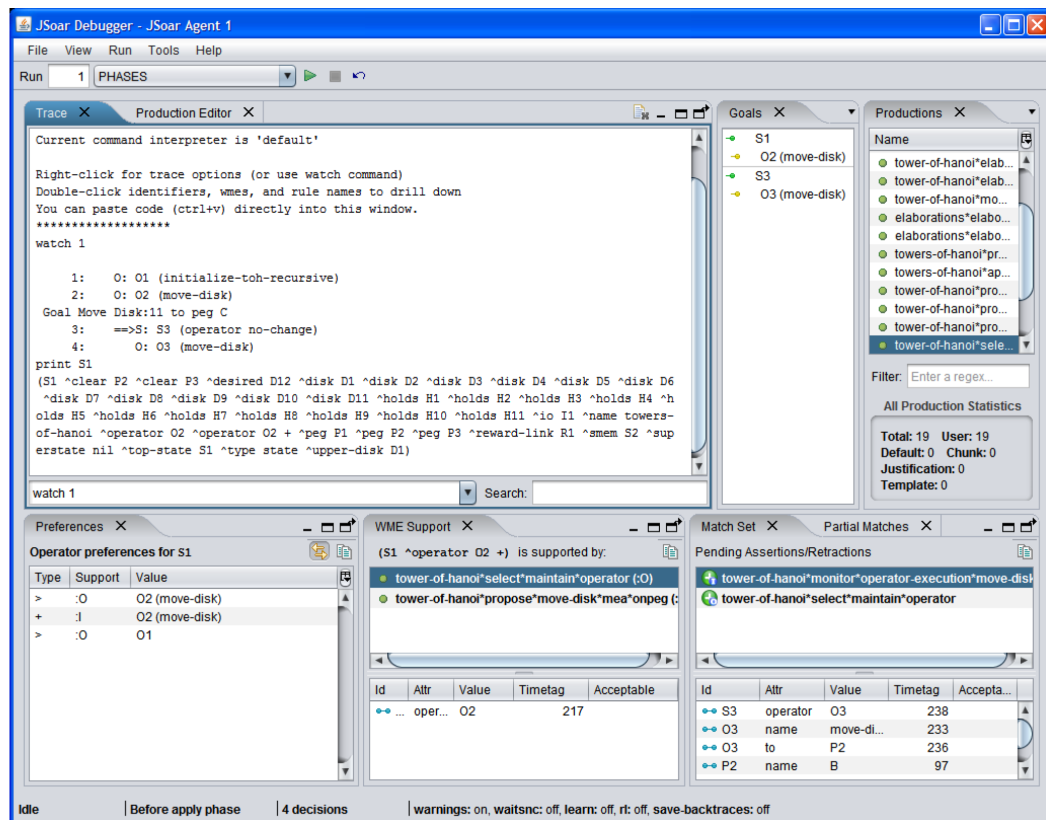
[2] `http://en.wikipedia.org/wiki/Soar_(cognitive_architecture)`

[3] `http://web.eecs.umich.edu/~soar/sitemaker/workshop/21/Brad_Jones_SGIO.pdf`

Obr. 5.1: MOUT: Agent pozoruje vojaka pripraveného vyčistiť chodbu tvaru L[15].



Obr. 5.2: Hrubý náčrt SOAR architektúry (obrázok prebratý z http://www.hindawi.com/journals/mpe/2012/530561)



Obr. 5.3: Screenshot programu SOAR Debugger (obrázok prebratý z http://code.google.com/p/soar/wiki/IntroSoarDebugger)

AR s inými platformami. Komplexným príkladom SOAR bota sa zaoberá [20]. Cieľom tejto práce ja implementácia SOAR agentov v hre Haunt, ktorá využíva virtuálny svet UT.

Ďalšou z výhod SOARu je vývojový nástroj jSOAR Debugger[4] (obrázok 5.3). Je napísaný v jazyku JAVA a jedným z ďalších cieľov vývojárov SOAR je integrovať tento nástroj do Eclipse IDE a vytvrorit tak SOAR IDE.

Principiálne je SOAR so svojim ladiacim nástrojom PoJACTRu najbližší. Porovnanie oboch platforiem je komplikované, pretože sa jedná o podobnú otázku ako ktorá kognitívna architektúra je lepšia. Pre vyriešenie tejto otázky potrebujeme experimenty, a pre experimenty potrebujeme nástroj pre obe platformy.

5.3 CTF

UT pre svoju jednoduchosť často využívaným prostriedkom pre testovanie rôznych implementácia riadenia agentov. Výnimku netvorí ani problém CTF. Z viacerých prác treba spomenúť [21], ktorá využíva knižnicu Pogamut pre vytvorenie komplexného CTF bota.

Bolo by zaujímavé aké výsledky by dosiahol PoJACTR CTF bot oproti špecializovanému CTF botovi. Pre PoJACTRv2 však CTF problém nie je cieľom, ale prostriedkom ako odskúšať validáciu navigačného modulu a ladiacich nástrojov.

[4]http://code.google.com/p/soar/wiki/IntroSoarDebugger

V rámci tejto práce sme porovnávali CTF botov oproti natívnym UT2004 botom a Pogamut implementáciou CTF.

5.4 Zhrnutie

Ako opisuje táto kapitola, existujú práce podobné PoJACTR a najmä knižnica SOAR rieši veľmi podobný problém (aj keď pomocou inej kognitívnej architektúry). Ak sa podarí popularizovať PoJACTR aj napriek relatívne malému množstvu potencionálnych záujemcov, môže sa tento plugin stať významým hráčom na poli testovania validity kognitívnych architektúr v komplexných virtuálnych svetoch.

6. Vývoj botov v plugine PoJACTRv1

Táto kapitola sa pozrie na implementáciu botov v knižnici PoJACTRv1 a PoJACTRv2. V prvej polovici popisuje implementáciu botov v knižnici PoJACTR1 a jeho najväčšie nedostatky. V druhej časti navrhneme ako tieto problémy vyriešiť integráciou knižnice PoJACTR2 s Eclipse IDE.

6.1 Implementácia PoJACTRv1

Keď sa vyvíjal prvý plugin PoJACTR najdôležitejším bodom bolo samotné prepojenie Pogamutu a jACT-R. Pohodlnosť programovania nehrala rolu. PoJACTRv1 musel vyriešiť niekoľko základných problémov:

Spustenie systémov: Pogamut a jACT-R sú obe veľké knižnice, obe majú relatívne jednoduché spustenie, avšak problém bol spustiť ich tak, aby sa všetky súčasti spustili v správnom poradí, a všetky krížové referencie boli valídne.

Reprezentácia UT sveta: Bolo treba vymyslieť ako reprezentovať informácie, ktoré Pogamut dostáva z virtuálneho sveta a rozdeliť ich tak aby kognitívne odpovedali modulom, ktoré teória ACT-R opisuje. Za týmto účelom boli implementované `AbstractChunkFactory` a `AbstractChunkEventProducer`. Producery zachytávajú senzorické správy z Pogamutu, pomocou faktory tried vyrábajú odpovedajúce ACT-R chunky. Z nich vyrábajú eventy, ktoré posúvajú registrovaným listerom. Tieto listenery registrujú odpovedajúce senzorické moduly a buffere.

Posielanie príkazov: ACT-R agent počas behu zadáva príkazy motorickým modulom. Príkazy sú chunky ACT-R teórie, preto bolo potrebné vyrobiť konvertor chunkov na príkazy pre Pogamut. Preto PoJACTR implemenuje objekty `AbstractCommandSerializer` ktoré serializujú ACT-R chunky na objekty `CommandMessage` ktoré posiela do UT rozhranie `IAct`.

Preťaženie jACT-R modulov a bufferov: Pre spoluprácu knižníc museli byť preťažené všetky senzorické a motorové moduly jACT-R agentov, aby vedeli prijímať a odosielať informácie do sveta.

Naimplementovať ACT-R Huntera: Ako jednoduchý príklad sa PoJACTRv1 implementoval jednoduchého reaktívneho Hunter agenta. Ten však neobsahoval žiaden navigačný modul, pohyboval sa len pomocou príkazu `Move`[1] a preto bol použiteľný len na jednoduché mapy.

Všetka táto práca odpovedala tisícom riadkom kódu pluginu a v rámci práce sa akurát stihol odladiť Hunter bot. Treba mať na pamäti, že bez reálneho nasadenia

[1] Move je jednoduchí príkaz pomocou ktorého sa vie bot pohybovať obyčajným krokom.

nie je možné odladiť knižnicu a čím viac sa knižnica používa tým viac chýb sa dokáže odhaliť. Pri implementovaní Hunter bota sme odhalili nielen kopec chýb, ale zistili sme, že najviac času pri implementácii trvalo ladenie jednotlivých modulov. Ak bot robil niečo neočakávané, tak to bolo v 9 z 10 prípadoch kvôli implementačnej chybe v niektorom plugine. Pre príklad môžme uviesť problém, kedy bol prestal približne po minúte robiť čokoľvek.

6.2 Implementácia modelu

Jednoduchá implementácia modelu v PoJACTRv1 predpokladá, že napíšeme jACT-R súbor modelu a XML súbor pre jACT-R prostredie. jACT-R model je taktiež XML súbor pozostávajúci z niekoľkých častí:

1. XML hlavička a meno modelu.
2. Definícia modulov: zahŕňa triedu ktorá modul implementuje a jej parametre.
3. Definícia deklaratívnej pamäte: špecifikácia typov chunkov a počiatočných chunkov.
4. Definícia procedurálnej pamäte: špecifikácia procedurálnych pravidiel. Každé pozostáva z podmienok a akcií.
5. Definícia bufferov: deklarovanie jednotlivých buffrov a nastavenie ich parametrov.

Environment súbor, ktorý potrebuje jACT-R na vstupe sa používa ako definícia prostredia jACT-R. Kompletný `environment.xml` definuje následne body:

1. Definícia kontroleru, ktorý riadi celý beh jACT-R.
2. Definícia konektoru, ktorý riadi pripojenie modelu na virtuálny svet.
3. Definícia modelov ktoré sa majú v rámci behu spustiť.
4. Definícia attachmentov, čo sú rôzne pomocné triedy, napríklad na ladenie.

Prvé tri body sú v každom environment súbore povinné. Základom napojenia jACT-R na Pogamut je práve reimplementovanie konektoru a kontroleru. Aby sme spustili bota, chýba nám už len Java trieda s `main` metódou. V nej vytvoríme PoJACTR objekt typu `Application` s parametrom cesty k environment súboru.

Keď máme toto všetko prichystané, môžeme spustiť bota. Prvým problémom, ktorý môže nastať je navalídne XML. Väčšina XML editorov síce vie skontrolovať samotnú XML validitu, to však neznamená, že triedy, ktoré sú definované v modely sú reálne. [2]

[2] V starej jACT-R knižnici bol dokonca bug, ktorý túto chybu zaobalil do svojej nešťastne zvolenej a zo samotných logov nebolo možné dohľadať, ktorú triedu sa nepodarilo nainicializovať. Bolo potrebé dať breakpoint do vnútra knižnice jACT-R a odchytiť pôvodnú výnimku. Aj keď tento problém je už dávno opravený, ilustruje potrebu vývojara rozumieť Jave, aj keď neimplementuje žiaden nový modul ani buffer.

Ak je náš model úspešne spustiteľný, pozorujeme ho kým nespraví niečo čo nemá. Ak sa tak stane, musíme problém odladiť, upraviť bota a pustiť ho znovu. Tým sa dostávame k dvom hlavným prístupom k ladeniu, ktoré sme mohli využívať pri vývoji Hunter agenta.

6.2.1 Breakpointy a Java Debugger

Základnou možnosťou, ktorú pozná dôverne každý Java vývojar je dať breakpoint priamo do kódu v ktorom očakávame chybu. Zastavíme teda proces alebo celý virtuálny stroj a môžeme sa pozrieť aké su aktuálne dáta v pamäti, teda aj v bufferoch. Taktiež si môžeme odkrokovať interné algoritmy. Šikovnejší z nás vedia vďaka OpenSource povahe jACT-R knižnice odkrokovať priamo procedurálny modul a zistiť čo sa deje v rôznych fázach ACT-R cyklu. Najväčší problém tohto prístupu je jeho jednorázovosť. Breakpoint totižto nestopne simuláciu virtuálneho sveta UT. Teda svet beží ďalej a po opätovnom spustení procesu/virtuálneho stroja už PoJACTR dostáva signály odpovedajúce inému stavu sveta. Pri väčšine chýb na Hunter botovi bolo po použití breakpointu potrebné celú simuláciu spustiť znovu pre nerelevantnosť dát, prípadne počkať kým sa bot dostane znovu do relevantnej pozície.

6.2.2 Textové logovanie

Druhou možnosťou je zapnúť si plné textové logovanie. To však má jeden zásadný problém, a to že štandardný PoJACTR bot generuje 1-2MB textových logov za sekundu, čo odpovedá 4-8 tisíc riadkom za sekundu. Dohľadávanie relevantných informácii sa stalo nočnou morou pri implementácii Hunter bota. Pre aspoň nejaké zjednodušenie sme používali sadu `bash` skriptov, ktoré filtrovy logy do dielčích logov. To riešilo situáciu len z časti, lebo ak bola zaujímavá situácia v konkrétnom čase na logu pre produkčný modul a chceli sme pozrieť čo sa vtedy dialo vo vizuálnom buffre, museli sme podľa času nájsť záznam v odpovedajúcom logu vizuálneho modulu. Nepochopiteľne sa do textových logov nedostávali informácie o aktuálnom stave bufferov, túto informáciu sme museli logovť sami v implementovaných bufferoch. Niekedy bolo nutné pre konkrétnu chybu naimplementovať konkrétny `bash` skript, ktorý vyfiltroval len konkrétne dáta potrebné odladenie danej chyby.

6.2.3 Ladenie Hunter bota

Pri ladení Hunter bota sme narazili na varietu chýb a samozrejme sme veľa používali obe metódy. Niektoré otázky sa však opakovali pri takmer každej chybe:

1. Čo sa deje v jednotlivých moduloch?
2. Čo sa nachádza v relevantných bufferoch?
3. Prečo je obsah bufferov taký ako je?

Odpoveďami na tieto otázky sa začínalo každé ladenie. Prvé dve otázky sme väčšinou zodpovedali pomocou logov, zatiaľ čo pre poslednú odpoveď bol potrebný precízne umiestnený breakpoint. Takého ladenie bolo veľmi pracné, nezriedka sa

stávalo, že sme riešili problém niekoľko hodín z čoho sme viac ako polovicu času trávili len dohľadávaním informácii. Vtedy vznikla myšlienka použiť špecializované ladiace nástroje, čo vyeskalovalo až do implementácie tejto práce.

6.3 Čo vývojár potrebuje

Pred začatím vývoj ladiacich nástrojov pre ACT-R vyvstali postupne nasledujúce otázky:

Aké informácie by uľahčili prácu vývojarovi v PoJACTR? Všetky informácie sa dajú textovo logovať, takže preformulujeme otázku: Aké zobrazenie informácii by vývojarovi pomohlo? Implementácia Hunter bota jasne ukázala, že je potrebné vidieť len informácie z času ktorý nás zaujíma. Zároveň potrebujem ľahko zobraziť len tie aspekty bota, ktoré nás zaujímaju. Príklad: čo sa dialo v produkčnom a vizuálnom module v čase keď bot zbadal nepriateľského bota?

Aké informácie má vývojár v jACT-R? Knižnica jACT-R už má pár rokov, a má za sebou niekoľko zmien na svojich nástrojoch. Je preto rozumné predpokladať, že boli implementované k spokojnosti vývojárov. Nedali by sa upraviť nástroje jACT-R tak aby vyhovovali vývoju real-time agentov vo virtuálnom prostredí UT2004? Po preskúmaní nástrojov jACT-R bolo jasné, že ak by s nimi vedel PoJACTR pracovať, veľký kus času venovaný súčasnému ladeniu by bol minulosťou.

Aké IDE využiť (Eclipse vs Netbeans)? Nedeliteľnou súčasťou vývoja hociakého software v súčasnosti je IDE. Súčasťou Pogamutu je aj plugin do Netbeans IDE, ktorý mimoiné implementuje manažér UT serverov potrebný pre PoJACTRv2. PoJACTRv2 vybral Eclipse IDE kvôli predpokladu, že implementácia manažéra UT serverov v Eclipse sa zdala jednoduchšia, ako implementácia nástrojov jACT-R v Netbeans. Nakoniec by asi bola náročnosť rovnaká, pretože všetky využité jACT-R pluginy bolo treba preťažovať a upravovať.

Po zodpovedaní týchto otázok bolo jasné, že súčasťou PoJACTRv2 bude niekoľko Eclipse rozšírení podľa vzoru Pogamut a jACT-R rozšírení.

7. Rozšírenia Eclipse IDE

V predchádzajúcej kapitole sme identifikovali kľúčové potreby pre ladenie PoJACTR. Táto kapitola ukáže aké konkrétne rozšírenia budeme implementovať. Následne prejdeme jednotlivé rozšírenia a ukážeme s akými problémami sme sa museli vysporiadať pri ich implementácii.

7.1 Platforma Eclipse

Pre začiatok bolo potrebné preskúmať platformu na ktorej je postavené Eclipse IDE (7.1). Tá pozostáva z niekoľkých vrstiev. Najspodnejšia vrstva je postavená na OSGi[1] frameworku, ktorý je navrhnutý tak, aby sa dal ľahko modulárne rozširovať. To dovoľuje upraviť Eclipse IDE pre množstvo použití od vývoja webových stránok, cez modelovacie nástroje až po Eclipse pre testerov. Nad OSGi je sada základných knižníc, ktoré majú všetky Eclipse IDE spoločné. Zabezpečujú napríklad užívateľské rozhranie alebo tímovú spoluprácu. Posledná vrstva je vrstva pluginov, ktorú rozširuje PoJACTR.

Pluginy sú vo svojej podstate Java projekty, ktoré pridávajú novú funkčnosť. Môžu byť len pridávať nové triedy, ktoré následne využijú pluginy rozširujúce samotnú funkčnosť Eclipse IDE. Rozširovanie funkčností je založené na dvoch typoch objektov:

Extension Point Ak chce plugin A umožniť iným pluginom rozšíriť alebo upraviť jeho funkčnosť, tak definuje *extension point*. V ňom deklaruje *kontrakt*-kombináciu značiek XML a Java rozhraní, ktoré musí samotný *extension* spĺňať.

Extension Iný plugin B implementuje *extension* pre konkrétny *extension point*, ktorým rozširuje/preťažuje funkčnosť pluginu A.

Príkladom pre *extension point* je zvýrazňovanie syntaxe. Plugin editor definuje rozhranie a iné pluginy dopĺňajú zvýrazňovač pre rôzne jazyky.

Vývoj rozšírenia PoJACTR pre Eclipse pozostával vo veľkej miere z implementovania objektov *extension*. Obrázok 7.1 ilustruje rozsah práce na rozšírení PoJACTR, ako aj ich *extension point*. Pretože sa práca inšpirovala jACT-R pluginmi bolo potrebné preskúmať ich funkčnosť, hlavne funkčnosť základných jACT-R pluginových projektov (v zátvorke je uvedený zhruba počet riadkov Java kódu ktorý plugin obsahuje):

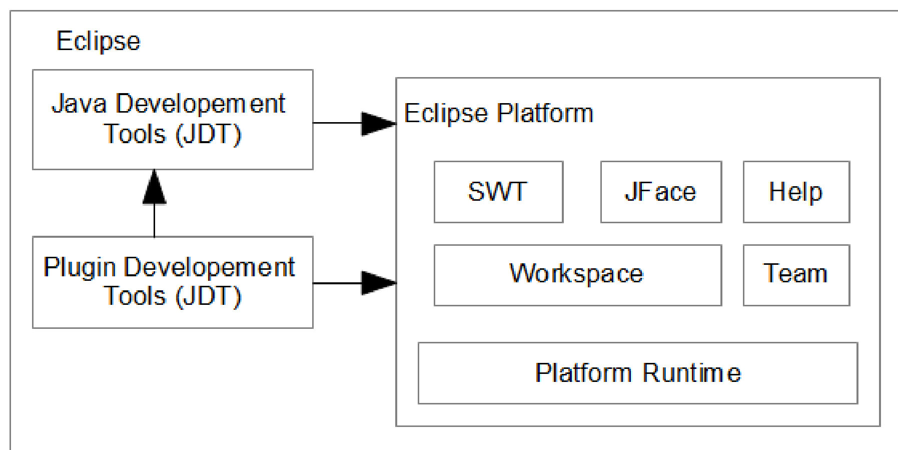
org.jactr.eclipse.core jadro všetkých jACT-R pluginov (7700)

org.jactr.eclipse.ui základné užívateľské prostredie (21000)

org.jactr.eclipse.runtime projekt pre spúšťače a komunikáciu s modelom počas behu (20000)

org.jactr.eclipse.runtime.ui užívateľské prostredie pre runtime plugin (27000)

[1] http://www.osgi.org/Specifications/HomePage



Obr. 7.1: Zjednodušený pohľad na Eclipse Platform.

Rozšírenie	Vzťah	Rozšírenie	Extension Point
jACT-R Editor			org.eclipse.ui.editor
jACT-R Project			org.eclipse.ui.newWizards
Pogamut Manager	používa	PoJACTR JMXBot	org.eclipse.ui.views
		Dátové štuktúry pre logy	org.eclipse.ui.propertyPages
PoJACTR Launcher	preťažuje	jACTR Launcher	org.*.launchConfigurationTypes
			org.*.launchConfigurationTabGroups
PoJACTR Log	sa inšpiruje	jACTR Log	org.eclipse.ui.views
Buffer state	preťažuje	AST Tree	org.eclipse.ui.views
Conflict Set	preťažuje	AST Tree	org.eclipse.ui.views
PoJACTR Perspective	sa inšpiruje	jACTR Perspective	org.eclipse.ui.perspectives

Obr. 7.2: Súčasti PoJACTR pluginu. Žltou sú vyznačené využité jACT-R súčasti a červenou súčasti implementované v rámci tejto práce.

7.2 Čo obsahuje rozšírenie PoJACTR pre Eclipse

Plugin PoJACTRv2 pre Eclipse IDE je súbor nasledujúcich rozšírení:

Editor pre jACT-R modely Syntakticky je model pre PoJACTR úplne zhodný ako model pre jACT-R a preto je možné modely písať v jACT-R editore.

Pogamut Manager Je Eclipse UI plugin, ktorý zobrazuje UT servre a ich botov v stromovej štruktúre. Slúži najmä na prehľad aktuálneho stavu UT serverov a agentov na nich pripojených. Zároveň dokáže vyvolať niektoré akcie či už na UT servri alebo jednotlivých agentoch. a vyvolanie niektorých akcií (napríklad logovania).

PoJACTR JMXBot JMX je štandardná komponenta jazyka Java, ktorá slúži na takzvaný Remote Procedure Call (RPC), čo je vzdialené volanie procedúr medzi rôznymi Java procesmi (pokojne aj na iných sieťach). Pogamut používa túto technológiu pre komunikáciu medzi pluginmi Netbeans IDE a jednotlivými inštanciami Pogamutích agentov. Podobný spôsob používa aj PoJACTR, ktorý implementuje vlastné JMX triedy, cez ktoré komunikujú pluginy Eclipse so spustenými agentmi.

PoJACTR Launcher jACT-R Launcher je plugin do Eclipse, vďaka ktorému je jednoduché spustiť model tak, aby bol automaticky napojený na ostatné Eclipse pluginy. PoJACTRv2 tento plugin reimplementoval, aby sa podobne dali spúšťať PoJACTR agenti. Vývojár len nakliká aký model chce spustiť a nastaví logovanie. Následne plugin vytvorí `environment` súbor pre PoJACTR a po spustení je agent automaticky pripravený komunikovať s Eclipse pluginmi.

Dátové štuktúry pre logy Pre rozdielnosť implementácie komunikácie medzi jACT-R a PoJACTR bolo nutné implementovať vlastné štruktúry pre logované dáta. jACT-R využíval stream, ktorý sa celý zobrazoval v Eclipse. PoJACTR na druhú stranu používa len dielčie logy z časových intervalov.

PoJACTR Log, Buffer state, Conflict Set Tri Eclipse pluginy, ktoré slúžia na štruktúrované zobrazenie dát. Výzorovo sú takmer zhodné s jACT-R pluginmi. Aby však fungovali, bolo ich treba preťažiť a reimplementovať všetky časti, ktoré pracujú s logovanými dátami. Samotné zobrazenie dát je plne prebraté z jACT-R.

PoJACTR Perspective Perspektíva v Eclipse je počiatočné nastavenie pracovnej plochy tak, aby vyhovovala konkrétnemu účelu. V tomto prípade je účelom prehľadné zobrazenie logov PoJACTR agentov.

Tieto rozšírenia dávajú dohromady komplexný nástroj na ladenie agentov.

Obr. 7.3: Ukážka jACT-R editora. Vľavo vidíme formátované XML modelu. Vpravo stromová štruktúra modelu v skratke.

7.3 Editor pre jACT-R modely

Výhodou pre programovanie v jACT-R plugine je XML editor, ktorý je špecializovaný pre formát jACT-R modelu. Obsahuje niekoľko užitočných funkčností, ktoré plne využije aj vývojár PoJACTR botov:

- Automatická kontrola XML formátu a XML schémy.
- Automatická kontrola parametrov modelu ako sú napríklad použité triedy modulov.
- Prehľadný outline špecializovaný na jACT-R modely

Rovnaký editor (7.3) je našťastie použiteľný aj pre PoJACTR modely, neboli na ňom potrebné žiadne úpravy v rámci tejto práce. Uvádzame ho tu len pre úplnosť ako súčasť výhod PoJACTRv2, aj keď samotný PoJACTR Eclipse Plugin ho len využíva.

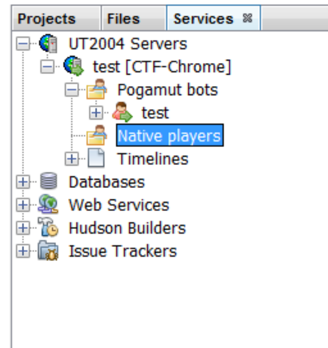
7.4 jACT-R Project

Okrem editora používa PoJACTR ešte jedno rozšírenie jACT-R bezo zmeny a to *jACT-R project*. To rozšírenie definuje štruktúru projektu, ale napríklad aj automatické kontrolovanie `*.jactr` súborov. Zároveň ho využíva spúšťač aby zistil zoznam modelov v projekte. Keďže toto rozšírenie nemalo zmysel reimplementovať využívame pôvodnú jACT-R verziu. V ďalších častiach sa už pozrieme na implementáciu PoJACTR pluginu.

7.5 Jadro pluginu

Eclipse UI plugin je špeciálny typ Java projektu. Prvým krokom k implementácii bolo pochopiť ako funguje platforma Eclipse a naimplementovať jadro rozšírenia. Absolútnym základom každého Eclipse UI pluginu sú tri súčasti:

Manifest projektu: je súbor, ktorý obsahuje základné informácie o projekte: meno, verziu, jeho závislosti na iných projektoch, balíky ktoré ponúka iným pluginom a zároveň definíciu triedy aktivátora.



Obr. 7.4: UT2004 Servers v NetbeansIDE

Manifest pluginu: `plugin.xml` špecifikuje ako plugin rozširuje Eclipse platformu, aké rozšírenia ponúka a ktoré triedy implementujú tieto rozšírenia.

Aktivátor. V prípade, že užívateľ vyvolá akciu, pre ktorú je potrebný náš plugin, Eclipsa vytvorí aktivátor. Vyvolá konštruktor a následne `startup` metódu ešte predtým, ako začne vytvárať ktorúkoľvek iný súčasť pluginu. Často, a aj v našom prípade, je plugin vytvorený ako singleton. Vďaka týmto skutočnostiam je možné používať aktivátor zo všetkých rozšírení PoJACTR pluginu.

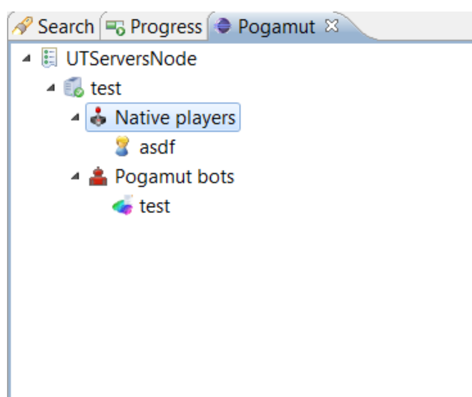
Ukázali sme, čo je potrebné pre základ pluginu. Teraz sa pozrieme na jednotlivé rozšírenia.

7.6 Pogamut Manager

Prvým rozšírením, okolo ktorého sme potom vystavali ostatné, bol Pogamut Manager. Pre prácu s PoJACTR agentmi sme potrebovali nástroj, pomocou ktorého budeme mať prehľad o servroch a ich botoch. Podobný nástroj existuje už v Netbeans IDE ako UT2004 Servers položka v Services okne7.4.

Obdobný plugin sme naimplementovali ako Pogamut Manager (7.5) okno do Eclipse IDE. Principiálne to funguje tak, že Eclipse vytvorí vlastného neviditeľného agenenta, ktorý je napojený pomocou Pogamutu na UT server. Informácie, ktoré tento agent získava zobrazuje Pogamut Manager. Na druhú stranu dokáže posielať cez tohto neviditeľného agenta na server správy. Podobný prístup bol použitý v Netbeans, avšak kvôli rozdielnosti platforiem musel byť celý Pogamut Manager naimplementovaný od začiatku. Samotná implementácia pozostávala z niekoľkých blokov:

- Stromová dátová štruktúra, ktorá bude držať dáta zobrazovaného stromu.
- Vytvorenie View pre zobrazenie tejto stromovej štruktúry.
- UT Server Agent obsahuje kód ktorý okrem iného zachytáva príchodzích a odchodzích hráčov, dokáže pripojiť natívneho bota alebo zmeniť mapu virtuálneho sveta.
- Uloženie a nahratie nastavenia po reštartovaní Eclipse.



Obr. 7.5: Pogamut Manager v Eclipse

Trochu predbehneme a prezradíme, že budeme potrebovať vyvolať logovanie na konkrétnom modeli. Prvou jednoduchšou možnosťou bolo naimplementovať položku do kontextového menu Pogamut managera. Okrem toho sa nám zdalo výhodné, keby mal vývojár možnosť vyvolať logovanie aj priamo z hry UT2004. Často sa totiž stávalo, že sme pozorovali bota (či už ako spoluhráč, nepriateľ alebo spektátor), a nastala zaujímavá situácia, ktorú sme potrebovali okamžite zalogovať (napríklad Hunter agent sa zrazu prestane pohybovať). Preto sme do UT2004 Server agenta pridali senzor, ktorý sníma klávesy stláčané v UT klientoch. Nastavili sme spúšťač logovania tak, aby reagoval na klávesovú skratku ALT + L. Takže priamo z hry dokážeme vyvolať logovanie agenta.

7.7 PoJACTR JMXBot

Pogamut Manager vyriešil problém komunikácie medzi UT2004 serverom a Eclipse platformou. Ďalšou úlohou bolo vyriešiť komunikáciu medzi Eclipse platformou a PoJACTR botom. Najjednoduchším riešením z pohľadu komunikácie bota a platformy by bolo spustiť bota na rovnakom virtuálnom stroji ako plugin, to znamená priamo v platforme Eclipse. Tento postup sme však zavrhli z dvoch dôvodov.

- Pri každej kritickej chybe by okrem bota spadla celá Eclipse.
- Na tom istom virtuálnom stroji nie je možné ten istý kód aj upravovať aj spúšťať. To by znemožnilo v plugine implementovať nové moduly.

Po tejto úvahe bolo jasné, že bot musí bežať na inom virtuálnom stroji. Aby sme mohli vidieť informácie bota, bolo potrebné vymysliet komunikáciu s ním. Tento problém už vyriešili knižnice ktoré PoJACTR spája:

jACT-R v Eclipse: využíva na prepojenie knižnicu Apache MINA [2]. Je to sieťový framework pre Java aplikácie optimalizovaný pre vysoký výkon. jACT-R však používa verziu 1.1, ktorá už nie je udržiavaná.

[2] Domáca stránka knižnice MINA: http://mina.apache.org/

Pogamut v Netbeans: využíva technológiu Java Management Extensions (JMX)[3], pomocou ktorej môžu vzdialený klienti monitorovať a spravovať aplikáciu na inom virtuálnom stroji.

Výhodou MINA bolo napojenie na jACT-R a JMX napojenie na Pogamut. Nakoniec sme sa rozhodli pre JMX ktorý je síce pomalší, ale zato je rozbustnejší. Navyše po správnom nakonfigurovaní sa jednoducho pridávajú nové funkčnosti.

Základom technológie JMX sú takzvané MBeany (Managed Beans). MBean je objekt v Jave ktorý ponúkajú prístup k funkčnostiam virtuálneho stroja. Najjednoduchší typ je statická MBean, čo je v podstate rozhranie s metódami, ktoré po nainicializovaní môžeme volať z iného virtuálneho stroja. Takúto triedu registrujeme na MBean server. Z iného virtuálneho stroja sa pripojíme na vzdialený MBean server, vyžiadame si MBean a môžeme volať vzdialene ľubovoľné metódy na tomto objekte. Na pozadí sa JMX framework postará o serializovanie a deserializovanie parametrov metódy, zavolá metódu na cieľovej MBeane a jej návratovú hodnotu zase serializuje a deserializuje na klientskom virtuálnom stroji.

Pri implementácii bolo najťažšie zistiť ako JMX v Pogamute funguje. Samotná implementácia už bola priamočiara. Základom boli dva objekty a jeden interface:

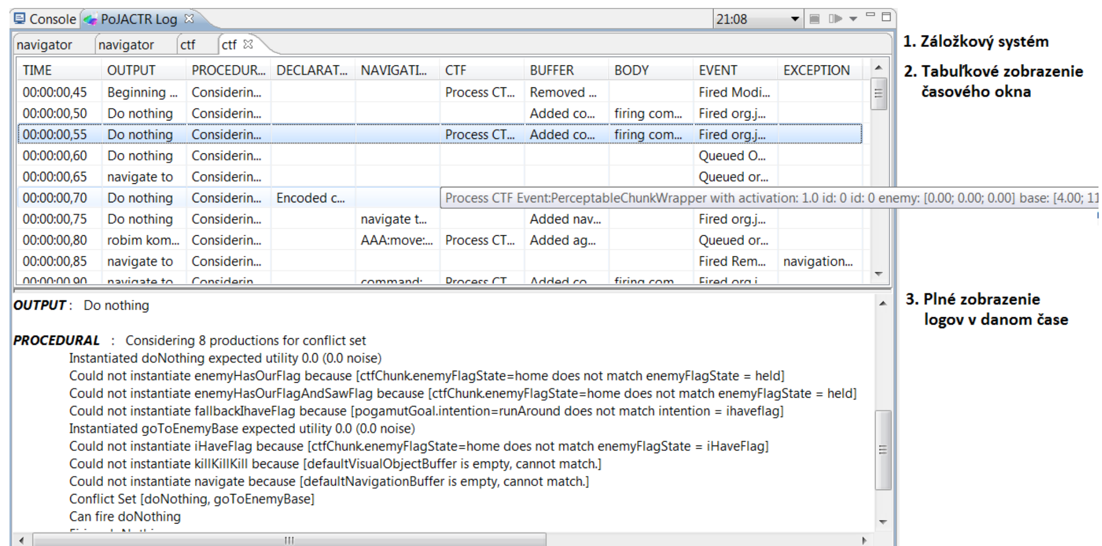
- `PoJACTRMonitorMBean` je rozhranie s metódami, ktoré PoJACTR plugin potrebuje volať z Eclipse na virtuálnom stroji PoJACTR bota.
- `PoJACTRMonitor` je trieda implementujúca `PoJACTRMonitorMBean` na virtuálnom stroji PoJACTR bota.
- `PoJACTRJMXProxy` je trieda implementujúca `PoJACTRMonitorMBean` v Eclipse, ktorá pomocou JMX vzialene metódy.

Ďalším krokom bolo zverejniť triedu na JMX server, čo sme urobili preťažením metódy `addJMXComponents` na `PoJACTRUT2004Bot`, ktorou zverejňuje svoje štandardné MBeany Pogamut. Posledným problémom bolo pripojiť sa na JMX server virtuálneho stroja PoJACTR bota. Tu nám zase pomohol Pogamut, ktorý posiela informácie o JMX servery v objektoch typu `Player` (adresu servera vracia metóda `getJmx()`. Tieto objekty zas zachytáva `UT2004Server` na zobrazovanie aktuálne pripojených hráčov, ktorý sme opísali v predchádzajúcej podkapitole. Implementáciou takéhoto prepojenia sme si zabezpečili jednoduchú a robustnú komunikáciu medzi virtuálnym strojom Eclipse a virtuálnym strojom PoJACTR bota.

7.8 PoJACTR Log

Nasledujúcim krokom je implementácia najdôležitejšej súčasti vývojových nástrojov a tou je PoJACTR Log. Toto rozšírenie odpovedá na otázku čo sa deje v konkrétnom čase v jednotlivých bufferoch. Inšpirácia pre toto rozšírenie pochádza z jACT-R a preto sme chceli využiť rovnaké zobrazenie aj pre PoJACTR. jACT-R Log zobrazenie fungovalo tak, že pre konkrétny model zobrazilo logované dáta pre celý beh modelu. To funguje dobre pre behy, ktoré majú malý počet cyklov.

[3] Domáca stránka knižnice JMX: http://www.oracle.com/technetwork/java/javase/tech/java-management-140525.html



Obr. 7.6: PoJACTR Log

Rádovo niekoľko desiatok, maximálne zopár stoviek cyklov. PoJACTR bot však beží v reálnom čase, a jeden cyklus za 50ms znamená 1200 záznamov za minútu. Pritom ladenie bota môže trvať desiatky minút. Druhým problémom, pre ktorý nie je možné použiť štandardný jACT-R prístup, je označenie dôležitých logov. Ak by aj bolo možné zobraziť desiatky tisíc logov v Eclipse, bolo by to pre ladenie podobne ťažké, pretože by sme nevedeli kde sú zaujímavé informácie. Pre tieto dva dôvody sme museli upraviť vlastnosti logovania tak, že nezaznamenávame celé logy, len konkrétne časové okná, ktoré označíme za dôležité.

Samotné zobrazenie logov pozostáva z troch častí (7.6):

- Záložkový systém pre jednotlivé časové okná. Každý agent má vlastnú záložku pre každé časové okno.
- Tabuľkové zobrazenie logov v konkrétnom časovom okne. Riadky odpovedajú jednotlivým cyklom v časovom okne a stĺpce jednotlivým modulom.
- Textové zobrazenie logov pre konkrétny čas. Nie všetko sa vojde do tabuľkového zobrazenia, preto sú kompletné dáta zobrazené aj textovo.

Naším cieľom v tomto prípade bolo zanechať vizuálnu stránku pluginu bezozmeny, avšak naplniť ju vlastnými dátami. Pôvodné okno jACT-R Log používal rozhranie `ISessionData` cez ktoré bol napojený na aktuálny beh modelu. Interne sa dáta prenášali už spomenutou knižnicou MINA z virtuálneho stroja modelu. Toto napojenie sme museli odstrániť a preto bolo potrebné reimplementovať všetky súčasti. Reimplementovať sme museli následné triedy:

`MonitorData`: reprezentujú informácie z jedného časového okna. Tento objekt dostaneme naplnený pomocou JMX z virtuálneho stroja. V princípe je to mapa podľa času, ktorá obsahoje pole objektov `AbstractTransformedEvent`. Rôzny potomkovia tejto abstraktnej triedy obsahujú rôzne aspekty behu modelu.

`ModelLogData`: reprezentuje tú časť logov, ktoré zobrazuje PoJACTR Log. Tým odpovedajú objekty typu `BulkLogEvent`.

`LogData`: reprezentuje jeden riadok tabuľky, teda obsahuje logy z jedného modelového cyklu.

`ModelLogDataContentProvider`: implementuje štandardný interface, z ktorého čerpá dáta tabuľka `TableViewer`. dáta z `ModelLogView`

`ModelLogDataLabelProvider`: implementuje štandardný interface, ktorý sa používa pre zobrazovanie dát na tabulke typu `TableViewer`.

`ModelLogView`: je samotné rozšírenie triedy `ViewPart` ktorú sme definovali v manifeste `plugin.xml`. Jedine táto trieda dedí od jACT-R triedy, z ktorej využíva niektoré metódy týkajúce sa užívateľského rozhrania. Všetok kód, ktorý pracuje s dátami, sme museli naimplementovať nanovo.

7.9 Conflict Set, Buffers state

PoJACTR Log naobsahuje všetky informácie z daného okamihu, k tomu sú potrebné ešte doplnkové okná. Okná pracujú veľmi podobným spôsobom aj keď zobrazujú odlišný typ dát. Preto ich rozoberieme v jednej podkapitole. Najprv pripomenieme čo tieto okná zobrazujú:

Conflict Set: toto okno zobrazuje informácie z procedurálneho modulu. Konkrétne dva stromy dát. Prvým sú všetky pravidlá aplikovateľné na buffere v danom cykle modelu. Druhým stromom je pravidlo, ktoré bolo vybrané a vykonané.

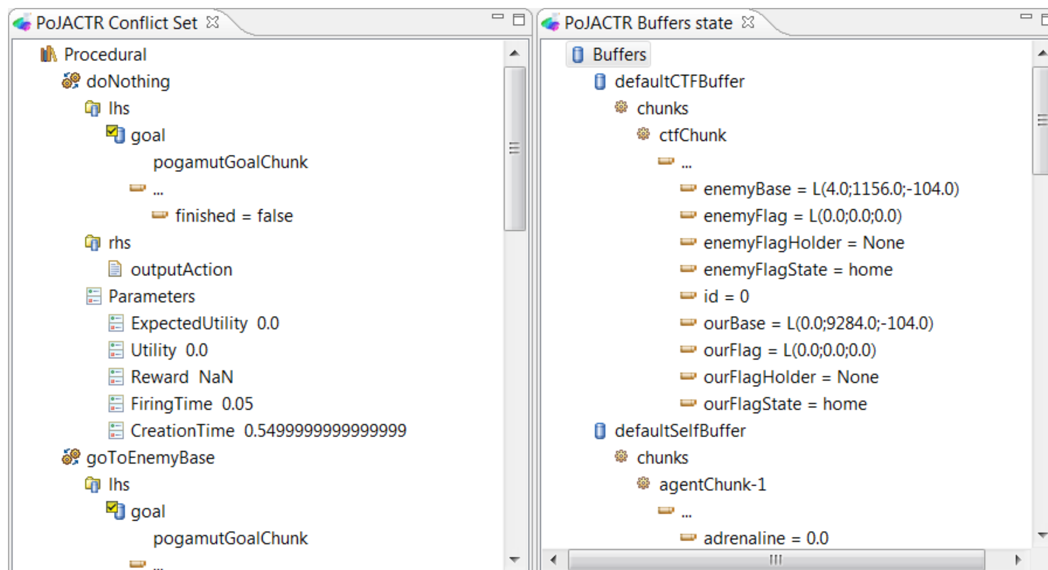
Buffers State: zobrazuje taktiež stromovú štruktúru. V nej je zobrazený obsah bufferov, ktoré sú v danom časovom okamžiku z nejakého dôvodu zaujímavé (definíciou *zaujímavosti*sa zaoberá podkapitola o generovaní logov).

Obe okna používajú pre zobrazenie stromovú štruktúru ako vidíme na 7.7. Verzie týchto okien v jACT-R používali na jej uloženie knižnicu ANTLR [4], konkrétne jej triedu `org.antlr.runtime.tree.CommonTree`. a nemali sme dôvod túto reprezentáciu meniť. Obe okná sú napojené na PoJACTR Log a ten pri zmene výberu v tabuľke logov vyvolá zmenu obsahu na týchto doplnkových oknách. Pre samotnú implementáciu okien sme museli vytvoriť tri triedy:

`AbstractTreeView` ktorý implementuje Eclipse rozhranie pre okno `ViewPart` a je schopný zobrazovať stromovú štruktúru `CommonTree`. Táto časť bola pôvodne prebratá z jACT-R, avšak v najnovšej verzii prestala byť kompatibilná s PoJACTR Logom, a tak sme ju museli reimplementovať.

`ConflictView` a `BufferView` obe dedia od abstraktnej triedy a preťažujú metódu na načítanie dát. V nej si vyťahujú z odpovedajúceho objektu `MonitorData` logy ktoré im patria v danom časovom okamihu. `Transformed-ProceduralEvent` obsahujú informácie z procedurálneho modulu, zatiaľ čo `BulkBufferEvent` obsahuje aktuálny stav bufferov. Obe triedy okien sú zvlášť registrované v manifeste pluginu.

[4] http://www.antlr.org/



Obr. 7.7: PoJACTR Conflict Set vľavo, PoJACTR Buffers State vpravo

7.10 Logovanie PojACT-R

V tejto fáze vývoja pluginu už vieme ako budeme zobrazovať dáta, vieme ako budeme prenášať dáta z virtuálneho stroja modelu a vieme ako vyvolať logovaciu akciu. Teraz ešte musíme vymyslieť ako dostať štruktúrované logovacie dáta z modelu.

Keďže pre logy využívame pôvodne štruktúry jACT-R, generovanie logov nebol problém, väčšinu logov sme nechali vygenerovať pôvodnú knižnicu jACT-R. K nim patrí logovanie bufferov a procedurálneho modelu. Kľúčom bolo zistiť, ako sa tieto logy generujú, aby sme ich vedeli presmerovať do triedy `MonitorData`. Túto potom posielame cez JMX do Eclipse IDE. Knižnica jACT-R využíva rôzne *attachmenty*a jedným z nich je aj `RuntimeTracer` , ktorý slúži ako prostredník medzi modelom a logovacím systémom. Tento *attachment*mimo iné obsahuje dve dôležité nastavenia:

ListenerClasses: toto nastavenie je zoznam tried implementujúcich `ITraceListener`, ktoré každý model instanciuje. Tieto triedy sa napájajú na rôzne súčasti modelu a generujú už spomínané inštancie `AbstractTransformedEvent`.

ITraceSinkClass: definuje triedu implementujúce rozhranie `ITraceSink` , ktoré posúva triedy `AbstractTransformedEvent` na miesto určenia. V prípade jACT-R Eclipse pluginu to bola trieda, ktorá pomocou knižnice MINA posielala dáta do Eclipse.

V našom prípade teda stačilo vyrobiť novú triedu `NetworkedSink` implementujúcu `ITraceSink` a predefinovať ju v `environment.xml`. V tejto triede sme implementovali FIFO frontu, ktorá udržuje posledných $n$logov. Pri vyvolaní logov si JMX komponenta vyžiada logy od našej `NetworkedSink` posledné logy, vyrobí z nich triedu `MonitorData` a tú vráti do JMX komponenty a cez ňu do Eclipse IDE. Tam sa už plugin postará o správne zobrazenie.

Takmer okamžite sme odhalili problém tohto prístupu a to, že logovacie dáta vyberá jACT-R. Napríklad dáta z bufferov sa logujú len vtedy, ak sa na bufferi niečo zmení. To znanemná, že pri chybe nehybnosti Hunter bota, by sme neodhalili, že vizuálny buffer je prázdny. Interne to funguje tak, že každá akcia v bufferi môže poznačiť, že buffer sa zmenil. Preto sme pridali na štandardnú implementáciu PoJACTR buffera parameter `BufferTracer`. Ak je nastavený na `true`, vyzerá to tak, ako by sa buffer menil v každom cykle a teda sa jeho obsah zaznamenáva do štandardných bufferových logov (a vidíme ho v Buffer State roširení v Eclipse). Pomocou tohto parametru sme napríklad zistili, že keď sa Hunter bot zastavil, jeho vizuálne buffere boli prázdne a zostali prázdne navždy.

Pri vývoji navigačného modulu sme narazili ešte na jeden problém. Bolo by výhodné, ak by sme mohli pri vývoji logovať akékoľvek dáta bez ohľadu na štandardné logovanie. Bolo by skvelé, ak by sme ich následne videli štruktúrovane v PoJACTR Log okne. Preto sme implementovali na štandardnú abstraktnú triedu PoJACTR modulu metódy, pomocou ktorých môžme uchovávať textové logy vrámci jedného cyklu. ktoré zbierajú logy vrámci jedného cyklu. Následne sme implementovali triedu `DefaultModuleRuntimeTracer` implementujúcu `ITraceListener` , ktorú môžme pridať do parametrov *attachmentu*`RuntimeTracer`. Táto trieda na konci celého cyklu prejde všetký moduly a zabalí všetky texty z modulov do objektu `BulkLogEvent`. Tento objekt následne pošle štandardnou cestou až do rozšírenia PoJACTR Log. Výhody tohto prístupu sme využili pri množstve drobných chýb navigačného a CTF modulu.

7.11 PoJACTR Launch Configuration

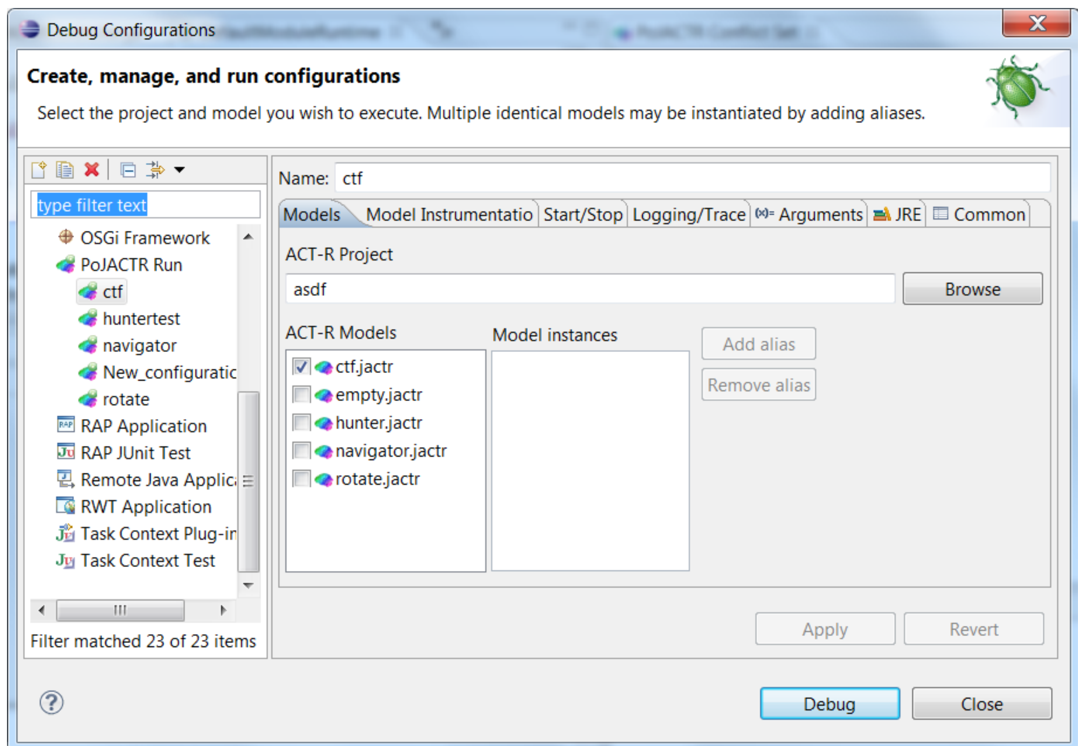
Všetky rozšírenia potrebné na ladenie PoJACTR modulu sme už implementovali. Inšpirácia jACT-R pluginom nám vnukla ešte dve pomocné rozšírenia, perspektívu a spúšťač. Perspektívu v krátkosti opíše nasledujúca podkapitola zatiaľ čo v tejto ukážeme spúšťač.

PoJACTR model je Java aplikácia, preto musí byť spustiteľný pomocou triedy s `main` metódou. V PoJACTRv1 sme takto spúšťali modely aj pre testovanie. Pre každý druh aplikácie sme museli konfigurovať zvlášť `environment.xml`. Eclipse plugin jACT-R však využíva sofistikovanejšie riešenie, a to Eclipse rozšírenie pre spúšťače aplikácií. Vďaka tomu môžme v grafickom prostredí (viz obrázok 7.9) naklikať aký chceme pustiť model, aké attachmenty chceme používať, nastaviť parametre pre Javu a spúšťač sám vytvorí konfiguračný súbor a spustí aplikáciu. Pretože konfiguračný súbor pre PoJACTR je rozdielny, museli sme reimplementovať niektoré súčasti a nadefinovať nové rozšírenie spúšťača do manifestu.

Štandardné spustenie cez `main` metódu sa používa napríklad pri dávkovom spúšťaní experimentov, alebo pri PoJACTR ako Maven projektoch.

7.12 PoJACTR Perspective

Ako posledné rozšírenie sme implementovali Eclipse perspektívu. Perspektíva definuje počiatočné nastavenie akcií (menu a panelu nástrojov) a počiatočnú množinu okien a ich rozmiestnenia v rámci pracovnej plochy v Eclipse. PoJACTR plugin pozostáva z niekoľkých oknových rozšírení a pri vývoji modelu používame



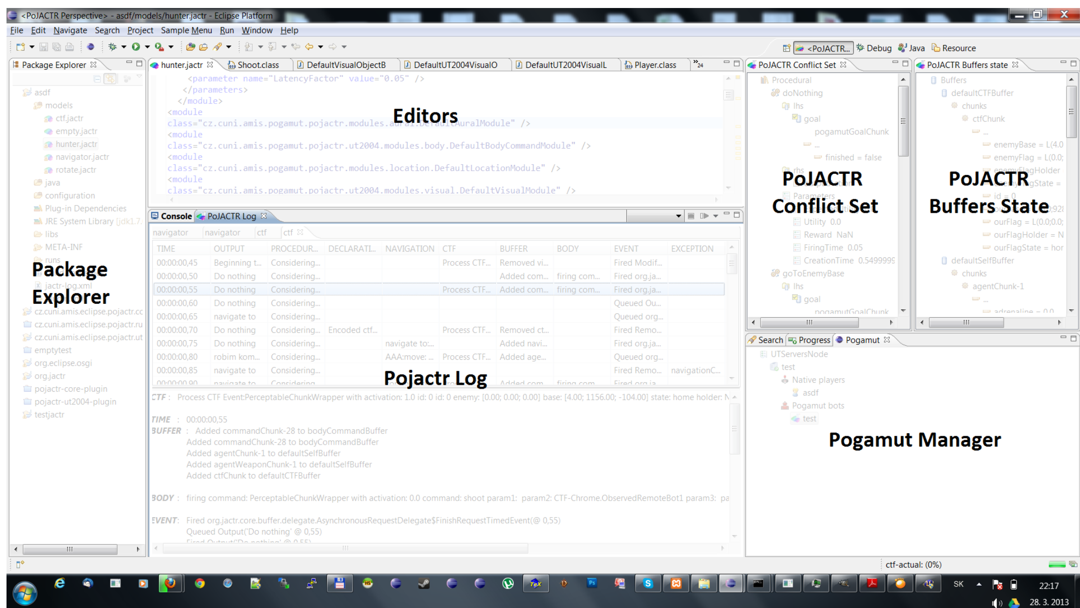
Obr. 7.8: Výber modelu v rozšírení PoJACTR Launch Configuration

všetky. Preto sme nastavením plochy, ktoré nám prišlo najprehľadnejšie uložili perspektívy. Toto rozloženie ilustruje obrázok 7.9.

Pre novú perspektívu je v Eclipse IDE potrebné implementovať interface `IPerspectiveFactory`. U nás to splnila trieda `PoJACTRPerspective` ktorá v metóde `createLayout` definuje naše úsporné rozloženie. Nakoniec sme nadefinovali rozšírenie perspektívy v manifeste `plugin.xml`.

7.13 Zhrnutie

V tejto kapitole sme implementovali rôzne Eclipse rozšírenia, ktoré sme uznali za potrebné pre ľahší vývoj PoJACTR botov. Bolo implementovaných 16 objektov typu *extension*, čo vydalo na 7500 riadkov Java kódu v pluginovom projekte. Pre úplnosť projekt PoJACTRCore má 7900 a PoJACTRUT2004 má 8500 riadkov a pôvodný PoJACTRv1 mal do 10000 riadkov.



Obr. 7.9: PoJACTR Perspective

8. CTF Boti v Jave

Táto kapitola preskúma možnosti implementácie CTF bota v klasickej Jave. Je potrebné pripomenúť, že hlavný prínos ACT-R bota a to kognitívna plausibilita sa v tejto kapitole ignoruje. Cieľ je vytvoriť bota, ktorý bude vedieť hrať hru CTF a dokáže hrať vyrovnanú partiu s predimplementovanými botmi z UT2004. Zároveň sa preskúma, ako jednoducho sa dá takýto bot vyrobiť pomocou štandarných tried Pogamutu. Treba podotknúť, že samotné správanie agenta nemusí byť nič komplikované. Dôležité pre neskoršie porovnanie je, aby komplikovanosť správania Java agenta odpovedala PoJACTR agentovi. Pre jednoduchosť sa prijal predpoklad, že boti spolu nekomunikujú, teda nevyužíva sa tímová taktika.

8.1 Čo potrebuje jednoduchý CTF bot

Problémy, ktoré musí riešiť dobrý CTF bot sa dajú rozdeliť do niekoľkých skupín podla úrovne.

Pripojenie a komunikácia so serverom: tento problém celkom pokrývajú knižnice Pogamutu takže pre implementátora je práca nulová.

Nižšia úroveň riadenia agenta: Pod tieto problémy spadá pohyb, útočenie, obrana a navigácia po prostredí. Pretože to je niečo, čo potrebuje takmer každý agent, Pogamut implementuje triedy, ktoré pomáhaju riešiť tieto problémy. Príkladom je napríklad navigačný modul, ktorý sám porieši takmer všetky problémy pohybu po prostredí. Samotné použitie navigačného modulu je prehľadne rozpísané v príklade navigačného bota, ktorý je súčasťou Pogamutu[1].

Vyššia úroveň riadenia agenta: Na tejto úrovni, sa agent rozhoduje aké ma ciele a aké nízkoúrovňové mechanizmy na ich docielenie využije. Agent medzi týmito cieľmi preskakuje podľa informácií z prostredia. Túto časť bolo potrebné napísať samostane.

Treba poznamenať, že Pogamut už obsahuje jednoduchého CTF bota, avšak pre potreby tejto práce bol naimplementovaný iný CTF bot, ktorý inak pristupuje k riadeniu a vyššej logike. Tento bot bude ale užitočný pre porovnanie s ostatnými implementáciami.

8.2 Implementácia

Táto podkapitola ukazuje, ako sa dá naimplementovať konkrétny CTF. Základom Pogamut agenta je Controller. Ten okrem iného obsahuje metódu logic(). Túto metódu volá Pogamut v nekonečnej slučke vždy, keď agent dostane dávku správ z UT servra. V tejto metóde agent reaguje na najnovšie správy zo servera. Obvykle sa logic púšta každých 250ms, ale túto premennú je možné upravovať (Pogamut je výkonnostne schopný až 100ms). Logic CTF agenta je rozdelený na niekoľko častí:

[1]`http://pogamut.cuni.cz/pogamut_files/latest/doc/tutorials/02-NavigationBot.html`

updateFlags: je metóda, ktorá sa stará o aktuálnosť mapy takzvaných flagov. Každý flag reprezentuje relatívne stav prostredia vzhľadom na bota. Každý flag buď je, alebo nie je prítomný v mape flagov a môže obsahovať dáta voči ktorým je flag viazaný. Napríklad flag `SEE_ENEMY` je prítomný vtedy, ako bot vidí nepriateľa a ako dáta obsahuje referenciu daného bota. Iným príkladom je `OWN_FLAG_HOME`, ktorý hovorí, že botova vlajka je v bezpečí na základni. A nakoniec sú tam flagy typu LOWHP, ktoré premieňajú rôzne premenné agenta na boolovské hodnoty, v tomto prípade.

highLogic: Táto metóda slúži na vyššiu úroveň riadenia. Podľa premennej `HIGH-_STATE` rozhoduje, aké príkazy pošle do nižšej úrovne agenta. Vyššiu úroveň doplňuje metóda `updateHighState` , ktorá mení `HIGH_STATE`.

lowLogic: V tejto fáze dochádza ku samotnému posielaniu príkazov na UT2004 server. Nízkoúrovňová logika pozostáva z dvoch paralelne vykonávaných úloh: pohybu a streľby. Pohyb využíva premennú *goTo* ktorá reprezentuje bod na mape, na ktorý sa má bot dostať. Túto premennú nastavuje vyššia logika v metóde highLogic. Obdobne pre streľbu sa používa premenná *enemy*, ktorá označuje nepriateľa, na ktorého má agent útočiť.

Metóda updateFlags, lowLogic a ostatné technické metódy, ktoré potrebuje každý CTF bot tvoria akýsi micro framework. Vďaka tomu je bolo možné vytvoriť niekoľkých botov podľa komplikovanosti správania. Stačilo len reimplementovať/rozšíriť metódy `highLogic` a `updateHighState`. Nášho Java CTF Bota najlepšie zhrnieme diagramom jeho stavov (obrázok 8.1).

V prvej verzii bolo treba vytvoriť základ bota a otestovať ho. Preto ma prvá verzia iba úplne jednoduchú logiku. `HIGH_STATE` prepína iba medzi dvoma stavmi, a to podľa umiestnenia nepriateľskej vlajky:

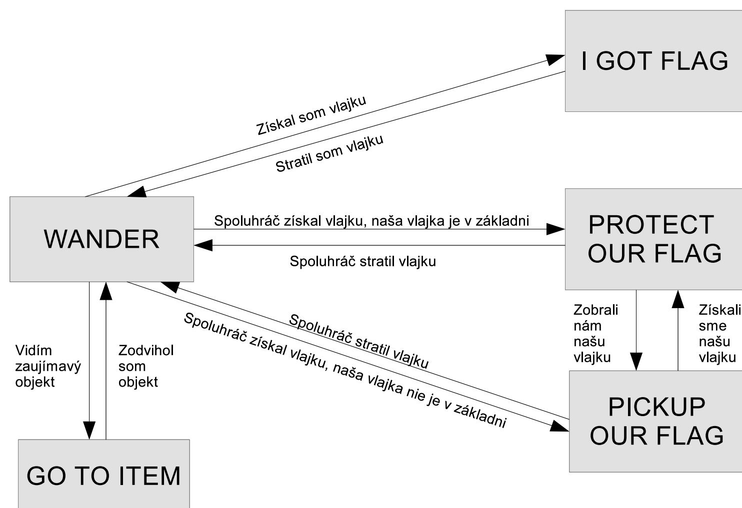
IGOTFLAG: Ak má bot vlajku vo svojich rukách uteká s ňou do vlastnej základne.

WANDER: Inak sa bot snaží dostať k nepriateľskej vlajke, či už je v nepriateľskej základni, zabili jej nositeľa, alebo ju nesie teamate do vlastnej základne.

Tieto dva stavy poslúžili akurát na otestovanie metódy `lowLogic` na 1v1 CTF mape. Pri hre tímov s dvoma a viac hráčmi bol tento bot absolútne nepoužiteľný. Stávalo sa totiž, že jeden bot je v stave IGOTFLAG a ostatný ho nasledujú, bez ohľadu na to kde je vlastná vlajka. Nakoniec sa všetci zišli v svojej základni a čakali čo urobí nepriateľ.
Aby sme vyrobili aspoň elementárne inteligentného bota, museli sme rozdeliť stav WANDER podľa toho kde je vlastná vlajka. Pridali sme ďalšie stavy:

PICKUPOURFLAG: Ak má botov tím nepriateľskú vlajku a vlastnú vlajku nemá v základni, snaží sa k nej dostať, prípadne sa dostať do nepriateľskej základne, aby navrátil vlajku domov.

PROTECTOURFLAG: Ak má botov tím nepriateľskú vlajku a aj vlastnú vlajku v základni, tak ide brániť vlastnú základňu.



Obr. 8.1: Stavový diagram Java CTF bota

Táto implementácia sa správala už relatívne inteligentne, nielen že dokázala skórovať, ale dokázala aj niektoré zápasy vyhrať.

Pri pozorovaní botov pri hre nám bolo po chvíľke jasné, že ich najväčším problémom je ignorácia užitočných objektov v prostredí (zbraní, nábojov a lekárničiek). Preto sme pre druhú verziu implementovali nový flag `NEAREST_ITEM`, ktorý je prítomný ak je nablízku užitočný objekt. Zároveň sme upravili implementáciu logiky WANDER tak, aby upravila cieľ navigácie ak je nablízku potrebný predmet.

8.3 Postup pre vývojára

Pre implementáciu treba povedať, že autor práce má skúsenosti s programovaním a pozná dobre ako vývojárske nástroje tak aj platformu Pogamut. Preto bolo jednoduché vytvoriť Maven projekt so závislosťami na Pogamut a prázdnu triedu Pogamut kontrolera. Pre samotný kontroler sme využili prepočítavanie flagov, úpravu stavu v každom cykle logiky a implementáciu funkčností jednotlivých stavov. Vďaka výborným príkladom a plnej dokumentácii Pogamutu (čo nebýva pri Open Source projektoch zvykom) bola implementácia priamočiara a zaobišla sa bez veľkých problémov. Dokonca sme nemuseli využiť ani logovacie nástroje Netbeans IDE a vystačili sme si s niekoľkými štandardnými textovými logmi do konzoly a debuggerom Eclipse IDE. Vymyslenie, implementovanie, otestovanie botov a pripravenie spustiteľného `jar` archívu trvalo zhruba 25 hodín. Jeden deň by bolo potrebné pridať pre priemerného Java vývojára, ktorý nemá skúsenosti s UT2004 a knižnicou Pogamut. Samozrejme časová náročnosť je nepriamo úmerná zdatnosti programátora.

9. Navigačný modul pre PoJACTR

Táto kapitola je venovaná navigačnému modulu pre PoJACTR. V prvej časti poukáže na dôvody vedúce k nutnosti implementácie navigácie pre CTF bota. Následne preskúmame ako sa pohybuje vo svete UT a ako problém pohybu rieši Pogamut. Posledná časť kapitoly opisuje samotnú implementáciu a výhody integrovaných nástrojov pri ladení modulu.

9.1 Prečo navigovať

Inteligentný pohyb po virtuálnom prostredí je nutnou podmienkou pre riešenie úloh, ktoré virtuálny agenti musia zvládať. Väčšinu času vo virtuálnom prostredí strávi hráč pohybom z miesta na miesto, po lekárničku, po zbraň alebo po vlajku. Prvý PoJACTR Hunter bot (ďalej Hunter) bol navrhnutý pre otestovanie senzorických a motorických modulov. Po prostredí sa pohyboval reaktívne: videl niečo *zaujímavé*, ako zbraň alebo nepriateľa a inicioval pohyb k danému objektu. Technicky to zabezpečovali produkčné pravidlá, ktoré sa spustili pri *zaujímavých* chunkoch v senzorických bufroch. Vizuálne a aurálne chunky so sebou nesú informáciu o lokácii a dosiahnuteľnosti. Pri dosiahnuteľnosti exekutívna časť pravidla poslala príkazový chunk `move` s adekvátnou lokáciou do motorického buffera zodpovedného za pohyb. Ak bota niečo zaujalo, ako napríklad korisť, vedel ju po jednoduchých mapách nasledovať kamkoľvek. Už pri Hunter botovi ale bolo jasné, že reaktívny pohyb nestačí. Triviálnym pozorovaním sa ukázalo niekoľko hlavných nedostatkov:

- Zaujímavý objekt sa nemusí dostať do senzorických buffrov. Buď je na okraji zorného poľa, alebo v ňom nie je vôbec. Stačí, aby bola lekárnička za rohom a bot ju vôbec nenájde.
- Prostredia v UT2004 a ani v iných virtuálnych simulátoroch vo väčšine prípadov nie sú dvojrozmerné. Často sa skladajú z niekoľkých vertikálnych úrovní a okrem nich obsahujú rôzne prekážky pre reaktívny pohyb. Priepasti, schody alebo výťahy sa nedajú prekonať len pomocou príkazu `move`, ale je potrebná sekvencia viacerých príkazov.
- Dobrý hráči UT vedia využiť všetky výhody prostredia, nie len tie viditeľné. Vedia kde nájsť lepšie zbrane alebo kde sa objavuje brnenie. Aj keby Hunter vedel kde tieto miesta sú, dostať by sa na ne vedel len náhodou.

Problém CTF je ešte komplikovanejší a obsahuje minimálne tri esenciálne problémy, na ktoré je potrebná navigácia. Z pohľadu bota:

- Potrebujem ukradnúť nepriateľskú vlajku. Tá je v ich základni. Ako sa tam dostanem?
- Ukoristil som vlajku. Ako ju donesiem do vlastnej základne?

- Ukradli nám vlajku. Kadiaľ mám ísť aby som zabránil nepriateľom skórovať.

Kvalita odpovedí na tieto otázky je dôležitým faktorom ovplyvňujúcim kvalitu výsledneho agenta. V každom prípade, aj na najjednoduchšie riešenie je potrebná navigácia.

9.2 Navigačný graf v UT2004

Základom navigácie pre virtuálny svet UT2004 je navigačný graf (9.1). Vrcholy grafu tvoria navigačné body (navigation point skrátene *navpoint*), umiestnené po celej mape. Navpointy sú pospájané hranami rôznych typov, ktoré reprezentujú spôsob akým sa vzdialenosť medzi navpointmi dá prekonať. Základné typy sú:

- Walk: najzákladnejší typ, prechod jednoduchou chôdzou, respektíve príkazom `move`.
- Jump: prechod skokom alebo dvojskokom. Odpovedajúci príkaz je `jump`.
- Door, Ladder, Special: dvere, rebríky a iné špeciálne prechody.

Taktiež samotné navpointy majú rôzne atribúty potrebné pre navigáciu:

- PathNode: základný navpoint bez dodatočnej informácie.
- PlayerStart, InventorySpot, SniperSpot: navpointy bez informácie pre navigáciu, avšak dôležité pre vyššiu logiku bota.
- LiftExit, LiftCenter: Navpointy signalizujúce výťah. LiftExit sú navpointy pre vstup a výstup výťahu a LiftCenter je pohyblivý navpoint spojený so samotným výťahom.
- JumpSpot: Označuje navpointy ktoré sú určené na skákanie ako štartovací bod pre hranu Jump. Špeciálnym typom je Jump Pad ktorý vystrelí bota do veľkej výšky.

9.3 Pohyb po virtuálnom svete

Inteligentný pohyb botov po virtuálnom prostredí sa dá rozdeliť na dve neľahké úlohy. Prvou, zdanlivo komplikovanejšou, je výpočet cesty z navpointu A do navpointu B. Druhou, na prvý pohľad jednoduchšou, je samotná prechod po vypočítanej ceste.

9.3.1 Plánovanie cesty

UT2004 bot má k dispozícii navigačný graf. Najjednoduchšie je teda nepoužiť jednoduchý grafový algoritmus na nájdenie najkratšej cesty, napríklad A*. Prvým problémom je, že niektoré typy riadenia botov potrebujú preplánovávať často, čo môže spomaľovať beh agenta. Toto sa dá vyriešiť predpočítaním všetkých ciest medzi všetkými navpointami, súpne pamäťová náročnosť a štart agenta, ale počas behu je počítanie trasy rýchle. Pogamut na to využíva FloydWarshall algoritmus.

Obr. 9.1: Vizualizácia navigačného grafu na mape Phobos2. Vizualizácia grafu za dá zapnúť pomocou kláves CTRL+G

Druhým problémom, ktorý robí plánovanie cesty ťažkým problémom, je definovanie *najlepšej*cesty. Najkratšia nemusí byť vždy najlepšia. Kvalita cesty závisí na mnohých faktoroch. Bot sa môže chcieť vyhnúť nepriateľom, zobrať po ceste brnenie alebo lekárničku navyše. Takého dynamické podmienky sťažujú nájdenie najlepšej cesty tak, že štandardné grafové algoritmy nemusia byť dostačujúce alebo dostatočne výkonne.

9.3.2 Nasledovanie cesty

Na prvý pohľad jednoduchá úloha, plánovač dodal cestu navigačným grafom ako zoznam navpointov a hrán po ktorých má bot prejsť. Problémom nie sú rôzne typy hrán ktoré vyžadujú na prejdenie rôzne príkazy nielen príkaz `move`. Hlavným problémom je načasovanie týchto príkazov rovnako ako je dôležité pri zadávaní príkazov klávesnice a myši pri hraní reálneho hráča. Problém pre bota je o to komplikovanejší, že človek dostáva informáciu o svete v reálnom čase, zatiaľ čo Pogamut bot len raz za časový úsek (štandardne 250ms). Konkrétnym príkladom je vyskočenie na idúci výťah, ktoré vyžaduje presné načasovanie príkazov `move` a `jump`.

9.4 Navigácia v ACTR

ACT-R ako teória nevynecháva ani problémy navigácie po priestore. Publikácií, ktoré sa zaoberajú navigáciou v ACT-R je niekoľko desiatok. Väčšina sa zaoberá konkrétnym navigačným problémom v malom virtuálnom prostredí, prípadne sa

zaoberá navigáciou len okrajovo. Príkladom konkrétnych úloh je [22], ktorý sa venuje učeniu plánovania jednoduchej navigačnej úlohy aby bola čo najefektívnejšia. Iným príkladom je [23], ktorý používa plánovanie trasy ako problém pre skúmanie schematického uvažovania.

Ako naznačila predchádzajúca kapitola, navigačný problém v komplexnom virtuálnom svete je problém komplikovaný. ACT-R teória si zakladá veľmi na svojej plauzibilnosti, čo ešte viac problém komplikuje. Návrh, implementácia, examinácia a validácia kognitívne plauzibilného riešenia je rozsahovo minimálne na jednu diplomovú prácu a teda zďaleka presahuje rozsah tejto práce. Napriek tomu je pre CTF navigácia životne dôležitá. Zároveň môže byť dôležitá pre skúmanie iných kognitívnych problémov, ktoré ju síce neskúmajú ale pre experimenty ju potrebujú. Potrebovali sme nájsť spôsob ako využiť už naimplementovanú navigáciu z Pogamutu tak, aby zapadla do modulového prostredia ACT-R. Fungovali by tak experimenty ktoré nepotrebujú plauzibilitu a navigačný modul by bol vymeniteľný.

9.5 Základy navigácie v Pogamute

Ako správna knižnica na vývoj botov má Pogamut navigačný modul už naimplementovaný. Pri implementácii Java CTF botov v kapitole 8 riadenie agenta používa vstavanú navigáciu. Aby bolo možné vytvoriť PoJACTR modul, ktorý bude využívať moduly Pogamutu, je nutné najprv ozrejmiť, ako fungujú jednotlivé súčasti navigácie v Pogamute.

Objekt ktorý je zodpovedný za celú navigáciu je `UT2004Navigation` , ktorý ponúka metódu `navigate`. Jej parametrom je objekt spĺňajúci `ILocated` a súčasťou väčšiny senzorických chunkov PoJACTRu. Riešenie navigácie v Pogamute je rovnako ako problém rozdelené do dvoch bodov: plánovanie trasy a nasledovanie trasy.

9.5.1 Plánovanie trasy

Plánovanie trasy v navigačnom grafe UT2004 zabezpečuje `PathPlanner`. Pogamut má predimplementované dva plánovače:

- `UT2004AStarPathPlanner` , ktorý slúži len ako proxy pre interný plánovač UT2004, ktorý používa algoritmus A*. Funguje tak, že zavolá príkaz s parametrami cesty a UT2004 mu vráti cestu v podobe zoznamu prepojených navpointov.

- `FloydWarshallPathPlanner`. Tento plánovač využíva algoritmus Floyd-Warshall, ktorý nájde najkratšie cesty medzi všetkými navpointami. Algoritmus beží v čase O(n^3), kde n je počet navpointov. To môže, hlavne na väčších mapách, spomaliť spustenie bota.

Samozrejme sa dá `UT2004Navigation` použiť aj s vlastným objektom `PathPlanner` a tak použiť vlastný algoritmus plánovania trasy, ktorý napríklad berie na vedomie pozíciu nepriateľov alebo optimalizuje trasu tak aby pozbieral potrebné objekty.

9.5.2 Nasledovanie trasy

V tomto prípade je základom exekútor trasy `IUT2004PathExecutor`, ktorý obaľuje `IUT2004PathNavigator` niekoľkými pomocnými kódmi (napríklad čaká kým je vypočítaná nová cesta). Navigátor je trieda, ktorá sa stará o konkrétnu exekúciu príkazov, tak aby bot prešiel po vypočítanej ceste. Vyskočí na výťah, počká, uhne pred raketou alebo využije skoky. Štandardná implementácia navigátora je `LogueNavigator`. Pre vývoj PoJACTR je dôležité, že pre posielanie všetkých príkazov do UT využíva rozhranie `IAct`.

9.6 Navigačný modul v PoJACTR

Táto podkapitola sa zaobera samotnou implementáciou navigačného modulu pre PoJACTR, a preto je dobré v bodoch si zopakovať aké sme mali požiadavky, možnosti a obmedzenia.

- Budeme implementovať nový navigačný modul.
- Budeme implementovať minimálne jeden buffer, pre komunikáciu s procedurálnym modulom.
- Chceli by sme využiť Pogamut navigáciu tak pre plánovanie trasy ako aj jej exekúciu.
- Pogamut navigácia potrebuje instanciovať v správnom poradí niekoľko kooperujúcich tried ako sú `UT2004Navigation`, `UT2004PathExecutor`, `FloydWarshallMap`, `LoqueNavigator` a niekoľko ďalších.
- Tieto triedy sa musia inicializovať pred pripojením k Pogamutu.
- `IUT2004PathNavigator` využíva pre exekúciu príkazov metódu `act` na rozhraní `IAct`.
- Triedy navigácie iniciujú s parametrom `UT2004Bot`, ktorý drží mimoiné aj rozhranie `IAct`.
- Pre posielanie príkazov na pohyb chceme používať štandardný motorický buffer `MovementCommandBuffer`.
- Každý príkaz by mal z návrhu ACT-R teórie prejsť cez procedurálny modul.
- Cieľ navigácie musel nastavovať procedurálny modul do buffera na navigačnom module.

9.6.1 Posielanie príkazov do motorických modulov

Základom bolo implementovať prázdny navigačný modul a prázdny navigačný buffer. Najprv sme vyriešili ako dostať príkazy z navigačného modulu, cez produkčný do `MovementCommandBuffer`. Za týmto účelom sme vytvorili nový typ jACT-R chunku pre príkazy a k nemu odpovedajúcu factory triedu. Keď potrebuje navigácia poslať príkaz Pogamutu, jednoducho vytvorí chunk s odpovedajúcim príkazom a publikuje ho v buffery. Tým pádom model môže preposlať tento

príkaz aj s jeho parametrami do motorického modulu. Nebude na škodu ak si ukážeme skrátenú verziu odpovedajúceho produkčného pravidla:

```
<production name="navigate">
   <condition>
      <match buffer="defaultNavigationBuffer"
             type="navigationCommandChunk">
         <slot name="command" equals="=cmd" />
         <slot name="param1" equals="=param1" />
      </match>
   </condition>
   <action>
      <add buffer="movementCommandBuffer"
           type="commandChunk">
         <slot name="command" equals="=cmd" />
         <slot name="param1" equals="=param1" />
      </add>
      <output>
"executing navigation command =cmd (=param1)"
  </output>
   </action>
</production>
```

Na exekúciu pravidla môže navigačný modul reagovať v metóde `matchedInternal(IChunk chunk)`, kde chunk je práve použitý `navigationCommandChunk`. Chunk odpovedá príkazu a teda vieme, ktorý príkaz sa vykonal a môžme chunk z bufferu odstrániť.

9.6.2 Nastavenie cieľa navigácie

Následnou otázkou bolo ako môže procedurálny modul povedať navigačnému, kam chce navigovať. Toto sa dalo vyriešiť preťažením inej metódy na bufferi, `addSourceChunkInternal(IChunk chunkToInsert)` V tejto metóde navigačný buffer spracuje chunk, extrahuje z neho lokáciu a nastaví nastaví cieľ pomocou metódy `UT2004Navigation.navigate`. Odpovedajúci kód v exekučnej časti pravidla je:

```
<add buffer="defaultNavigationBuffer"
     type="navigationCommandChunk">
   <slot name="command" equals="'navigateTo'" />
   <slot name="param1" equals="=loc" />
</add>
```

9.6.3 RandomNavpointBuffer

Okrem navigačného bufferu sme pre navigačný modul implementovali ešte `RandomNavpointBuffer`. Ako jeho meno napovedá, obsahuje náhodný dosiahnuteľný navpoint. Tento buffer sme využili pri testovaní, avšak dá sa využiť aj pri rôznych experimentoch. Napríklad pri takých, ktoré nepotrebujú aby agent navigoval na konkrétne miesto, ale len aby sa pohyboval. Pre jeho implementáciu sme vytvorili

chunk typ a jeho factory triedu pre chunk obsahujúci náhodnú lokáciu. Interne sa obsah chunku v bufferi menil vždy po niekoľkých cykloch behu.

9.6.4 Napojenie Pogamut navigácie

Posledným problémom bolo, ako dostať príkazy z Pogamut `UT2004Navigation` do nášho buffera `DefaultNavigationBuffer`. Prvým riešením ktoré nás napadlo bolo, že reimplementujeme `IUT2004PathNavigator` tak, aby miesto rozhrania `IAct` používal nejaké naše rozhranie. To by potom príkazy posúvalo do navigačného buffera. Po preskúmaní tejto možnosti zme zistili, že štandardný `LoqueNavigator` používa interne niekoľko tried, a každá z nich prispieva príkazmi do `IAct`. To by znamenalo veľa reimplementovania. Navyše pri vylepšení tried alebo výmene navigátora, by trebalo celý proces opakovať. Preto sme rozmýšľali ako to urobiť jednoduchšie.

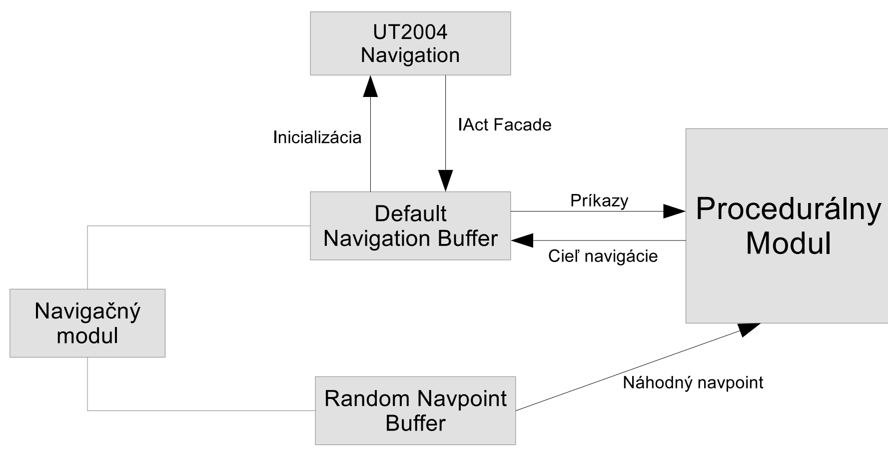
Ak by sa nám podarilo podhodiť navigačným triedam náš vlastný `IAct`, vedeli by sme ho implementovať ako *facade* triedu, ktorá by všetky príkazy posielala do navigačného buffru. To však nebolo také jednoduché, pretože toto rozhranie dostávajú v rámci inštancie `UT2004Bot`, ktorú používajú aj motorické moduly. Táto trieda je navyše inštanciovaná knižnicou Guice a nemá prázdny konštruktor, takže sa nedala len tak vytvoriť. Napriek tomu sa nám táto možnosť videla najlepšia. Preto sme len za týmto účelom upravili knižnicu Pogamut a pridali prázdny konštruktor triedy `UT2004Bot`. Vďaka tomu sme mohli pred inštanciovaním navigačných tried vytvoriť wrapper na bota, ktorý navigačným objektom podhodil našu *facade* implementáciu rozhrania `IAct`. Celú implementáciu navigačného modulu sme prehľadne zhrnuli do obrázku 9.2.

9.6.5 Využitie ladiacich nástrojov

Implementácia navigačného bufferu bola prvou veľkou skúškou ladiacich nástrojov. V tejto podkapitole ukážeme, ako sme ich využili. Samozrejme drobných využití bolo plno. Často sme napríklad potrebovali zodpovedať prečo sa dané pravidlo nevykonalo, alebo si logovať nejaký malý text. Pre príklad vyberáme dve väčšie použitia.

Prvým príkladom ich využitia bol test `RandomNavpointBuffer`. Po naimplementovaní sme potrebovali otestovať, či funguje tak ako má. Preto sme využili textové logovanie z abstraktného PoJACTR modulu a zalogovali sme text vždy, keď sme menili náhodnú lokáciu v chunku. Zároveň sme zapli nastavenie `BufferTracer`, takže v okne s bufferami sme videli v každom cykle celý obsah buffera. Vďaka tomu sme okamžite videli, či buffer funguje správne.

Druhým využiťím bolo ladenie *facade* triedy. Obdobne sme si logovali keď náš `IAct` preposlal príkaz. Zároveň sme logovali obsah navigačného buffra. Vďaka tomu sme takmer ihneď odhalili chybu v metóde `matchedInternal` navigačnom bufferi, kedy sa po využití chunku nevymazal z bufferu.



Obr. 9.2: Návrh PoJACTR navigácie

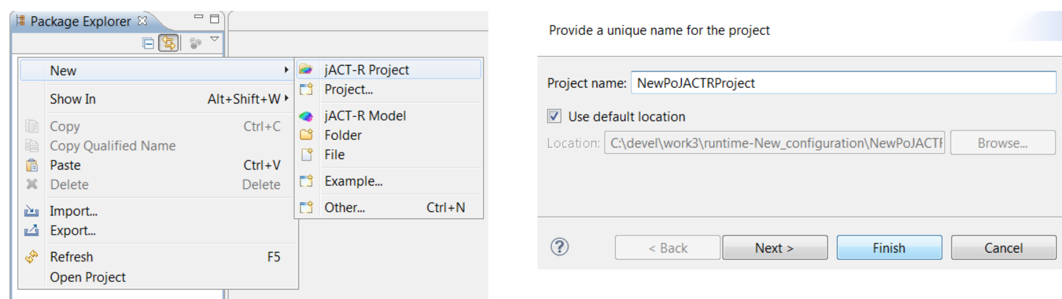
10. CTF Boti v PoJACTRv2

Táto kapitola sa zaoberá verifikáciou pluginu PoJACTRv2. Už máme naimplementované ladiace nástroje, ktoré nám pomôžu vo vývoji. Máme Java CTF bota, ktorým nám bude slúžiť ako inšpirácia a zároveň súper. A nakoniec máme navigačný modul, ktorý je vitálny pre každého CTF bota. Na začiatku kapitoly ukážeme ako využiť kompletnú sadu vývojových nástrojov PoJACTR. Pre kompletnú funkčnosť potrebujeme ešte informácie o vlajkách a základniach jednotlivých tímov. Tým sa zaoberá druhá časť kapitoly. Nakoniec sa pozrieme na samotnú implementáciu PoJACTR CTF bota.

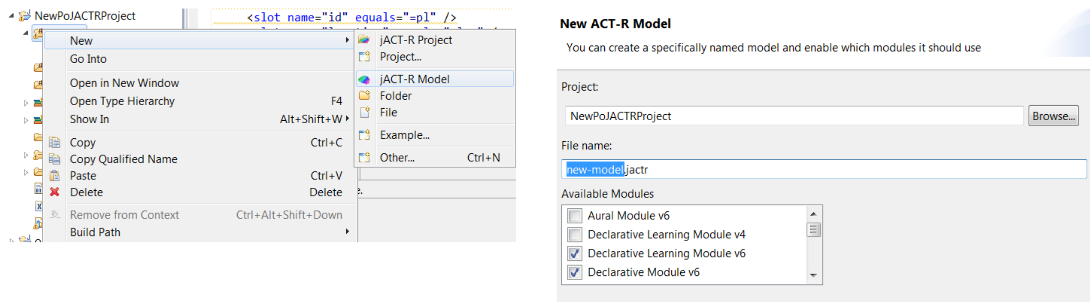
10.1 Využitie Eclipse pluginu

Už sme spomenuli niekoľko využití ladiacich nástrojov, avšak ešte sme neukázali ako sa dajú použiť ako komplexný nástroj. V tejto podkapitolke v bodoch ukážeme, ako sa jednoducho pracuje s PoJACTRv2.

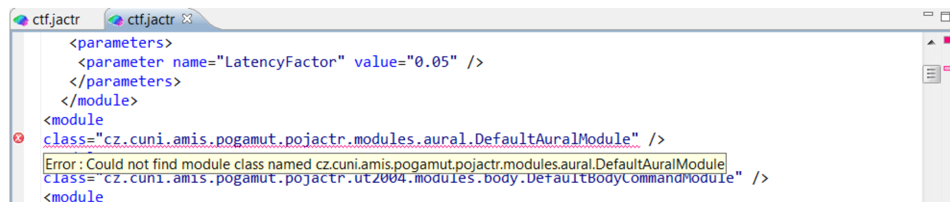
1. Spustíme Eclipse IDE s podporou pluginov jACT-R a PoJACTRv2.
2. Vytvoríme nový projekt a vyberieme jACT-R ako typ projektu. Eclipse nám sama vytvorí potrebnú adresárovú štruktúru a nastavenie projektu (10.1).
3. V zložke `models` vytvoríme nový súbor `.jactr` pomocou kontextového menu (10.2).
4. Implementujeme model celý, alebo si pre základ vyberieme niektorý z už implementovaných PoJACTR príkladov. Eclipse nám otvorí súbor automaticky v jACT-R Editore (10.3).
5. Vidíme, že editor nevedel nájsť triedy PoJACTR modulov a bufferov. Otvoríme manifest projektu a pridáme závislosti na PoJACTR projektoch (10.4).
6. Prepneme si perspektívu na PoJACTR Perspective a zobrazí sa nám Eclipse okno optimalizované pre vývoj PoJACTR agentov (7.9).



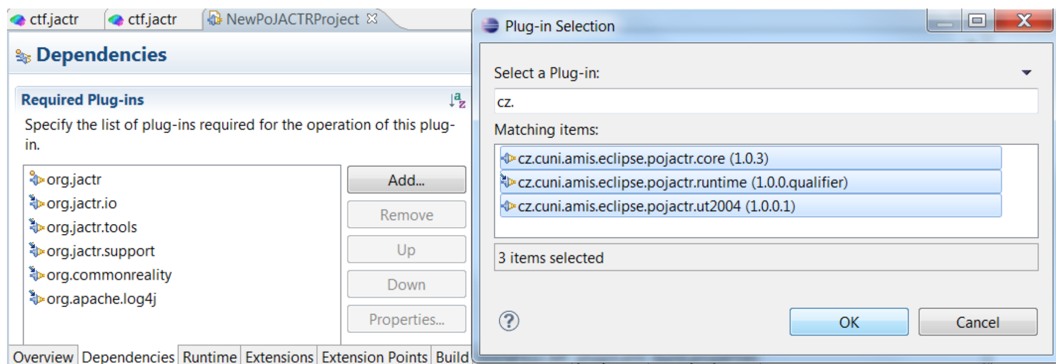
Obr. 10.1: Vľavo kontextové menu pre pridanie projektu. Vpravo formulár nového projektu.



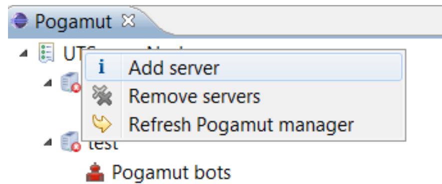
Obr. 10.2: Vľavo kontextové menu pre pridanie modelu. Vpravo formulár nového modelu.



Obr. 10.3: jACT-R Editor hlási chybu.



Obr. 10.4: Pridávanie závislosti do manifestu.



Obr. 10.5: Pridávanie servra do Pogamut managera.

7. V rámci perspektívy vidíme aj Pogamut Manager. Pomocou kontextového menu vytvoríme nový server a nastavíme IP a port. Ak na danej adrese už beží server, signalizuje nám to ikonka v strome. Zároveň uvidíme už pripojených botov, či už sú to Pogamut, PoJACTR alebo natívny boti (10.5).

8. Teraz sme pripravený spustiť model. Otvoríme si Eclipse Debug/Run konfiguráciu. V kontextovom menu vyberieme novú PoJACTR konfiguráciu. Napíšeme meno, vyberieme projekt, model. Ostatných taboch nastavíme ostatné atachmenty. Dôležité je zapnúť `DefaultModelLogger` a nastaviť mu parametre. Na tabe pre logovanie zapneme aj normálne logovanie, ak by nám spadol model ešte pri štarte (10.6).

9. Spustíme konfiguráciu. Okrem štandarnej konzoly, po chvíli vidíme bota aj v strome Pogamut Managera.

10. Cez kontextové menu v Pogamut Managerovi alebo klávesovou skratkou ALT + L priamo v hre vyvoláme logovanie.

11. Preskúmame logy v PoJACTR Log, Buffers State a Conflict Set. Upravíme model a opakujem od kroku 8.

Bez preháňania môžme povedať, že týmto spôsobom sa zrýchlil vývoj oproti PoJACTRv1 rádovo desaťnásobne.

10.2 CTF Modul

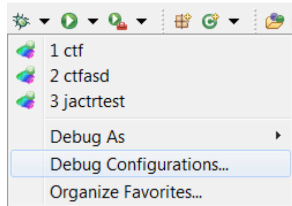
Pre CTF bota sú vitálne informácie o stave hry. Medzi ne patrí stav a pozície vlajok, ako aj umiestnenie oboch základní. Pogamut má pre tieto informácie senzorickú triedu `CTF`. Pre PoJACTR sme preto vyvinuli podobný CTF modul. Je to štandarný senzorický modul. Pre jeho realizáciu sme potrebovali implementovať niekoľko tried:

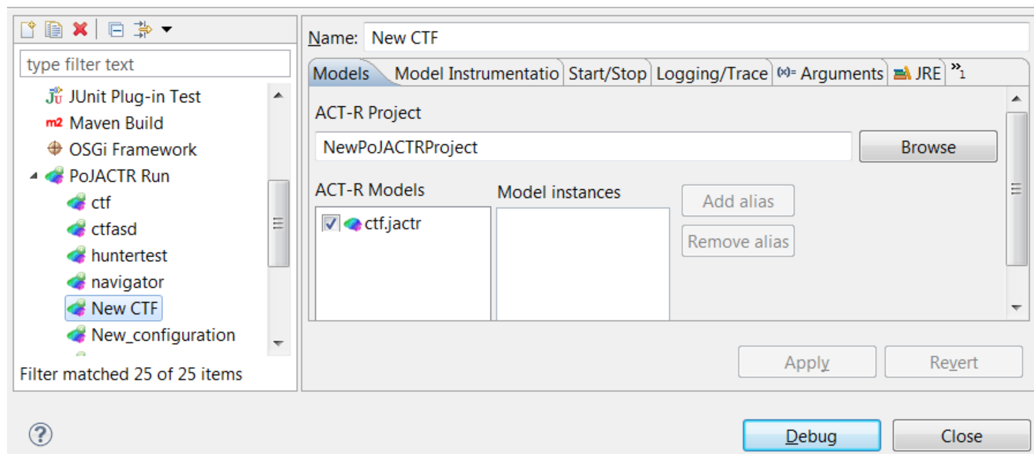
`CTFChunkWrapper`: obaľuje senzorický chunk typu CTF a pridáva funkčnosti potrebné pre štandardný senzorický buffer.

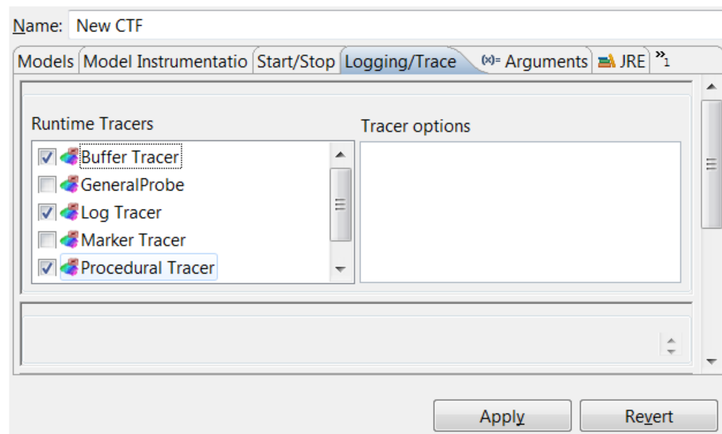
`CTFChunkFactory`: pri štarte registruje typ CTF do deklaratívneho modulu a dokáže vytvárať chunky typu CTF.

`DefaultCTFModule`: štandardný PoJACTR modul, definuje CTF buffer.

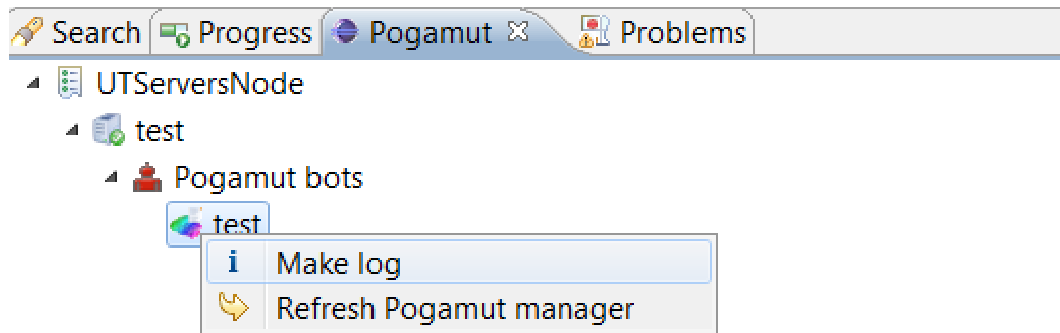
`DefaultCTFBuffer`: štandardný senzorický buffer pre CTF chunky.







Obr. 10.6: Pridanie novej Debug konfigurácie.



Obr. 10.7: Vyvolanie logovania.

`DefaultCTFProducer`: trieda, ktorá počúva na potrebné senzorické objekty z Pogamutu a následne generuje nové CTF dáta.

Podobne ako pri Java CTF botoch, aj teraz sme najprv vyrobili triviálneho bota, ktorý len chodí po vlajku a vracia sa do základne. Implementovali sme pravidlá, ktoré fungujú podobne ako dva stavy z Java bota WANDER a IGOTFLAG. Jedno pravidlo nastavuje navigačný chunk tak aby sa bot sa dostal k vlajke, druhé potom aby sa bot dostal naspäť do vlastnej základne. Vďaka rozšíreniam PoJACTR Log a Buffers State sme odhalili chyby okamžite. Napríklad ak bola vlajka doma, jej stav bol reprezentovaný dvoma rôznymi hodnotami odpovedajúceho slotu v CTF chunku, vďaka logu sme zistili prečo sa pravidlo nemohlo aplikovať, a vďaka zobrazeniu buffra sme hneď videli aké hodnoty v tej dobe chunk obsahoval.

10.3 Chat modul a epizodická pamäť

Pre úplnosť spomenieme dva moduly, ktoré sme mali v pláne využiť, ale nakoniec ich CTF bot nepotreboval.

10.3.1 Chat modul

Prvý je modul na komunikáciu. Pôvodne sme mysleli, že sa budú boti dohovárať pomocou chatu. Preto sme implementovali dva chatovacie moduly. Jeden senzorický na prijímanie správ a druhý motorický na odosielanie. Napriek tomu, že sme moduly nevyužili sú plne funkčné a stali sa súčasťou knižnice PoJACTR.

10.3.2 Epizodická pamäť

Epizodická pamäť slúži na zapamätanie si udalostí, ich času, miest a ich kontextu. Je to pohľad na minulosť očami agenta. Vďaka tejto pamäti vie agent povedať, čo robil včera alebo pred 10 minútami. Pre botov v UT2004 môže byť epizodická pamäť veľkou výhodou. Dokázali by sa rozhodovať nielen podľa aktuálnych

informácií, ale aj podľa starších pozorovaní. Príkladom je vedomosť, kde som naposledy videl vlajku alebo kedy som naposledy zobral brnenie.

Jedným z podcieľov implementácie novej verzie PoJACTR bolo aj implementovanie modulu epizodickej pamäte. Základnou myšlienkou bolo pripojiť existujúcu implementáciu ako ACT-R modul. Napriek tomu, že úloha epizodickej pamäte je pre virtuálnych agentov dôležitá, existuje len málo reálnych implementácii. Konkrétne práce sa venujú otestovaniu prototypu na konkrétnej úlohe a nepočítajú s budúcim použitím. Po niekoľkých pokusoch bolo zrejmé, že vytvorenie dobrého modulu pre epizodickú pamäť je rozsahovo nad rámec tejto práce. Preto boli niektoré epizodické informácie pridané do CTF chunku (napríklad posledná poloha vlajky). Pre úplnosť uvedieme možnosti, ktoré má vývojár ak chce implementovať epizodickú pamäť do Pogamutu/PoJACTR.

- Práca [12] Ondreja Burketa implementuje epizodickú pamäť pre bota v Pogamute[1]. Prepojenie tejto práce by vyžadovalo vyriešiť základný problém a tou je stará verzia Pogamutu. Experimenty z práce bežali na Pogamute verzie 2, zatiaľ čo PoJACTR je nadstavba pre verziu 3. Verzia 2 už nie je podporovaná niekoľko rokov.

- Zaujímavou možnosťou bol nový Bayesovský framework pre modelovanie epizodickej pamäte - DyBaNeM[7]. Ako jedna z mála prác nechce testovať konkrétny model v konkrétnej situácii, ale snaží sa vytvoriť framework, ktorý by mohli použiť iné práce/knižnice. DyBaNeM podporuje kódovanie, ukladanie a vyhľadávanie epizodických informácií. Napriek snahe sa v rozumnom čase nepodarilo prepojiť ho s knižnicou PoJACTR.

- Poslednou zaujímavou možnosťou by bola epizodická pamäť akú obsahujú knižnice SOAR[2]. Podporuje automatické ukladanie *spomienok* podľa informácií aktuálne dôležitých pre agenta. Vytvorenie podobného modulu pre ACT-R by obsahovalo extrahovanie časti knižnice SOAR a pripojenie na buffere ACT-R. Podobná práca s kognitívnym pozadím a s experimentmi by mohla rozsahovo konkurovať tejto práci.

10.4 CTF Bot

V tejto fáze už máme hotového najjednoduchšieho CTF bota, ktorý zvládol hrať zápas na mape pre dvoch hráčov. Nevie však ešte reagovať na všetky vzniknuté situácie pri hre viacerých hráčov. Preto sme pridali niekoľko ďalších pravidiel, ktoré fungujú ako stavy PROTECTOURFLAG a PICKUPOURFLAG z Java bota. Zhrnieme aké všetky pravidlá bot potreboval, aby fungoval ako CTF bot.

- Prázdne pravidlo. Bot potrebuje jedno pravidlo, ktoré môže aplikovať, ak nie je aplikovateľné ani jedno z ostatných pravidiel.

- Navigačné pravidlá, ktoré presúvajú príkazy z navigačného bufferu do motorických bufferoch. V našom prípade všetky príkazy idú do jedného bufferu `MovementCommandBuffer`. Preto nám na tento účel stačí jedno pravidlo.

[1] https://artemis.ms.mff.cuni.cz/pogamut/tiki-index.php?page=Episodic+memory+for+virtual+agent
[2] http://web.eecs.umich.edu/ nlderbin/ignore/files/tutorials/epmem_tutorial_2012.pdf

- Pravidla pre zmeny navigácie. Tieto pravidlá nahrádzajú HIGH_STATE z Java bota. Fungujú tak, že podľa stavu CTF buffera nastavujú cieľ navigácie.

- Pravidlá pre boj. To zahŕňa pravidlá pre strieľanie a zmenu zbraní.

Kompletná implementácia bota sa nachádza na priloženom DVD. Pre ilustráciu ukážeme dve zaujímavé pravidlá. Prvým je jedno z navigačných pravidiel:

```
<production name="iHaveFlag">
   <condition>
      <match buffer="defaultCTFBuffer"
             type="ctfChunk">
         <slot name="enemyFlagState" equals="iHaveFlag" />
         <slot name="ourBase" equals="=loc" />
      </match>
   </condition>
   <action>
      <add buffer="defaultNavigationBuffer"
           type="navigationCommandChunk">
         <slot name="command" equals="'navigateTo'" />
         <slot name="param1" equals="=loc" />
      </add>
      <modify buffer="goal">
         <slot name="intention" equals="'ihaveflag'"/>
      </modify>
      <output>"I got flag, go home: =loc"</output>
   </action>
</production>
```

Ako vidíme, pravidlo má podmienkovú časť a exekutívnu časť. Ako vidíme v podmienkovej časti, toto pravidlo sa vykoná, ak CTF buffer obsahuje CTF chunk so slotom `enemyFlagState` s hodnotou `iHaveFlag`. To znamená, že bot má nepriateľskú vlajku. Zároveň si v podmienkovej časti poznačíme premennú `loc`. Exekutívna časť potom obsahuje pridanie navigačného chunku, ktorý vyvolá zmenu cieľa navigácie. Zároveň vidíme, že bot si poznačil do `goal` buffera, že má vlajku a smeruje domov. Tag `output` zaloguje jeho text do štandardných logov.

```
<production name="kill">
   <condition>
      <match buffer="defaultVisualObjectBuffer"
             type="visualObjectChunk">
         <slot name="class" equals="'player'" />
         <slot name="id" equals="=pl" />
         <slot name="team" equals="0" />
         <slot name="location" equals="=loc" />
      </match>
   </condition>
   <action>
```

```
      <add buffer="bodyCommandBuffer" type="commandChunk">
         <slot name="command" equals="'shoot'" />
         <slot name="param2" equals="=pl" />
      </add>
      <modify buffer="goal">
     <slot name="following" equals="=pl"/>
    </modify>
    <output>"Beginning to follow"</output>
   </action>
  </production>
```

Druhým pravidlom je pravidlo `kill`. V princípe toto pravidlo znamená *Ak vidím nepriateľa, začnem po ňom strieľať.* . Pogamut príkaz `shoot` začne strieľať na nepriateľa, avšak sám strieľať neprestane. Pôvodne malo toto pravidlo dve pomocné pravidlá. Jedno posielalo vizuálnemu bufferu či stále vidí agenta. Druhé reagovalo na negatívnu odpoveď vizuálneho bufferu a pomocou pravidla `stopshooting` zastavovalo streľbu. V rámci implementácie CTF bota sme upravili `bodyCommandBuffer` aby streľbu po chvíli zastavil. Preto je na streľbu toto samotné pravidlo postačujúce.

Nakoniec si bot vystačil s 9 pravidlami. Ak počítame vytvorenie projektu, napísanie agenta, jeho otestovanie a úpravu triedy `bodyCommandBuffer`, tak sa výsledný čas implementácie pohyboval okolo 30 hodín. Nepočítali sme čas strávený implementáciou CTF a navigačného modulu. Taktiež sme nepočítali čas strávený ladením komponent, ktoré nesúviseli so samotným botom ale s knižnicou PoJACTR, a ten bol niekoľnásobne vyšší. Čas samotnej implementácie bol teda podobný ako čas implementácie Java bota. V nasledujúcej kapitole si ukážeme, ako na tom agenti boli výkonnostne.

11. Porovnanie implementácií botov

V tejto kapitole si ukážeme, aké výsledky dosahujú naši boti. Testovali sme našu Java implementáciu, našu PoJACTR implementáciu, CTF botov z Pogamutu a natívnych botov UT2004.

11.1 Cieľ

Aby sme porovnali výkon jednotlivých implementácií, púšťali sme ich do súbojov. Pre porovnanie sme sledovali, akú šancu má na skórovanie tím A a tím B. Aby sme porovnali botov, potrebujeme náhodnú veličinu. Ak skóroval tím A zarátame 0, ak tým B tak 1. Tým sme dostali náhodnu veličinu s binomickým rozdelením.

11.2 Metóda

Pre každú dvojicu sme opakovaňe spúštali experiment pokiaľ sme nemali 200 meraní tejto náhodnej veličiny, čo odpovedá 40 zápasom po 5 bodov[1]. Z týchto dát sme vedeli spočítať priemerné skóre zápasu. Aby sme však získali štatisticky signifikantné dáta, testovali sme valídnosť dvoch hypotéz:

- Tým botov A je lepší ako tým botov B. To odpovedá hypotéze: stredná hodnota nášho binomického rozdelenia je väčší rovný 0,5. Alternatívna hypotéza je: bot je horší.
- Tým botov B je 3 král lepší ako tým botov B. To znamená, že tým A skóruje 3 krát viac ako tým B. Stredná hodnota je väčšia rovná 0,75.

Tieto hypotézy sme testovali pomocou štatistického programu *R*a jeho funkcie `binom.test`.

11.3 Výsledky

Pre každú dvojicu tímov A a B sme spočítali ako skončí priemerný zápas a testovali sme dané hypotézy. Tabuľka 11.1 ukazuje priemerne skóre pre jednotlivé dvojice. Tabuľka 11.2 ukazuje ako dopadlo testovanie prvej hypotézy. Tabuľka 11.3 ukazuje či sú boti signifikantne lepší ako iné.

11.4 Diskusia

Ako vidíme, všetci boti signifikantne prehrali s natívnymi botmi. Na počudovanie najlepšie skóre vybojoval Java bot implementovaný v rámci tejto diplomovej

[1] Treba poznamenať, že sme pre jednoduchosť zanedbali vplyv mapy, rozdielne postavenie a výzbroj medzi jednotlivými botmi.

	UT2004	Pogamut	Java	PoJACTR
UT2004	-	4.67-0.33	4.00-1.00	4.82-0.18
Pogamut	0.33-4.67	-	4.17-0.83	4.27-0.73
Java	1.00-4.00	0.83-4.17	-	2.75-2.25
1 PoJACTR	0.73-4.27	0.73-4.27	2.25-2.75	-

Tabuľka 11.1: Výsledky experimentov. Každá bunka hovorí ako by priemerne skončil zápas tímu, ktorému odpovedá riadok, oproti tímu ktorému odpovedá stĺpec pri hre na 5 bodov.

	UT2004	Pogamut	Java	PoJACTR
UT2004	-	lepší(1)	lepší(1)	lepší(1)
Pogamut	horší(0)	-	lepší(1)	lepší(1)
Java	horší(0)	horší(0)	-	lepší(0.09)
PoJACTR	horší(0)	horší(0)	horší(0.98)	-

Tabuľka 11.2: Výsledky experimentov. Každá bunka hovorí či je tím v riadku lepší ako tým v stĺpci. V zátvorke je uvedená p-hodnota hypotézy, že tím v riadku skóruje častejšie ako tým v stĺpci.

práce. Najhoršie skóre vybojoval Pogamut CTF bot, a to aj napriek tomu, že Pogamut CTF bot jednoznačne vyhral nad oboma implementáciami tejto práce. Najzaujímavejším výsledkom je súboj medzi Java CTF a PoJACTR CTF.

Aj keď PoJACTR nakoniec prehral, dokázal udržať z Java botom krok. Tento bol zároveň najtesnejším výsledkom a dokazuje, že pri podobnej náročnosti implementácie dokáže PoJACTR kompetovať so štandartnými Java botmi Pogamutu.

	UT2004	Pogamut	Java	PoJACTR
UT2004	-	3x lepší(0.96)	3x lepší(1)	3x lepší(1)
Pogamut	3x horší(0)	-	3x lepší(0.99)	3x lepší(1)
Java	3x horší(0)	3x horší(0)	-	nie 3x lepší(0)
PoJACTR	3x horší(0)	3x horší(0)	nie 3x lepší(0)	-

Tabuľka 11.3: Výsledky experimentov. Každá bunka hovorí či je tím v riadku 3 krát lepší ako tým v stĺpci. V zátvorke je uvedená p-hodnota hypotézy, že tím v riadku skóruje 3 krát častejšie ako tým v stĺpci.

12. Budúca práca

PoJACTRv2 prešiel kus cesty od prototypu akým bol PoJACTRv1. Stále však nie je knižnicou, akou by sme chceli aby bol, stále je na ňom veľa práce. Je isté, že bez toho, aby sa knižnica začala používať na 2-3 ďalších projektoch, neodladí sa množstvo chýb. Myslíme si však, že PoJACTR by mohol zaujímať relatívne veľkú cieľovú skupinu. Počet užívateľov Pogamutu rastie každý rok, a o teórii ACT-R sa ročne publikujú desiatky textov. O zaujímavé projekty na platforme PoJACTR tiež nie je núdza.

Najprv spomenieme jedno rozšírenie ladiacich nástrojov, ktoré sme pôvodne chceli implementovať v rámci tejto práce. Tým je synchronizované zastavenie servera aj PoJACTR Bota. Zaujímavou možnosťou by bolo simultánne krokovanie UT2004 a PoJACTR bota, napríklad po sekundách, a súčasné logovanie daných časových okien.

Z vedeckého hľadiska by bolo zaujímavé implementovať kognitívne plausibilný modul. Na výber sa ponúka hneď niekoľko možností. Medzi nimi spomenieme epizodickú pamäť, učiaci sa procedurálny modul alebo vizuálny modul. Prípadne plausibilná navigácia by bola na takých veľkých mapách rozhodne výzvou.

Nakoniec ešte spomenieme súťaž BotPrize, kde súťažia boti v tom, ktorý je najpodobnejší človeku. Aby sme podrobili PoJACTR, ale aj samotnú ACT-R teóriu poriadnej skúške, stálo by za to urobiť komplexného PoJACTR bota, ktorý by sa súťaže zúčasnil.

13. Záver

V rámci tejto práce sme vytvorili novú verziu knižnice PoJACTR. Identifikovali najväčšie nedostatky pôvodnej verzie a navrhli riešenie - ladiace nástroje. Následne sme implementovali komplexné ladiace nástroje na platformu Eclipse IDE. Okrem toho sme implementovali navigačný modul, CTF modul a komunikačný modul.

Pre validáciu PoJACTRv2 sme vybrali problém Capture The Flag, ktorý sme riešili pomocou Java botov a PoJACTR botov. Porovnávali sme dva aspekty programovania a to náročnosť implementácie a výkon výsledných botov. Subjektívne sme zhodnotili, že náročnosť vývoja je porovnateľná. Oboch botov sme vyvíjali 25-30 hodín. Pre porovnanie výkonu sme použili Pogamut framework pre automatické spúšťanie zápasov. Napriek tomu, že PoJACTR bot nevyhral jeho výkon bol porovnateľný s výkonom Java bota.

Validácia riešenia dopadla pozitívne. Preto môžeme konštatovať, že sme vytvorili použiteľnú knižnicu rozširujúcu platformu Pogamut. Veríme, že pomôže výskumíkom modelovať kognitívne plauzibilných botov v prostredí UT2004.

Zoznam použitej literatúry

[1] Zemcak, L.: ACT-R in Pogamut. Bachelor's thesis, Computer Science Department, Charles University of Prague, Czech Republic, 2009.

[2] Gemrot, J., Kadlec, R., Bída, M., Burkert, O., Píbil, R., Havlíček, J., Zemčák, L., Šimlovič, J., Vansa, R., Štolba, M., Plch, T., Brom C.: Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. In: Agents for Games and Simulations, LNCS 5920, Springer(external link), 2009, pp. 1-15.

[3] Gemrot, J., Brom, C., Kadlec, R., Bída, M., Burkert, O., Zemčák, M., Píbil, R., Plch, T. Pogamut 3 – Virtual Humans Made Simple. In: Advances in Cognitive Systems, Nefti-Meziani, S. and Gray, J.O. eds, The Institution Of Engineering And Technology, pp. 211-243, 2010.

[4] Kadlec, R., Gemrot, J., Bída, M., Burkert, O., Havlíček, J., Zemčák, L., Pibil, R., Vansa, R., Brom, C.: Extensions and applications of Pogamut 3 platform. In: Proc. 9th IVA. LNAI, vol. 5773, pp.506–507.

[5] Kadlec, R.: Evolution of intelligent agent behaviour in computer games. Master thesis, Computer Science Department, Charles University of Prague, Czech Republic, 2008.

[6] Burkert, O.: Connectionist Model of Episodic Memory for Virtual Humans. Master thesis, Computer Science Department, Charles University of Prague, Czech Republic, 2009.

[7] Kadlec, R., Brom, C.: DyBaNeM: Bayesian Framework for Episodic Memory Modelling.

[8] Gemrot, J.: Joint behaviour for virtual humans. Master thesis, Computer Science Department, Charles University of Prague, Czech Republic, 2009.

[9] Brom, C., Gemrot, J., Bída, M., Burkert, O., Partington S. J., and Bryson, J. J.: POSH Tools for Game Agent Development by Students and Non-Programmers, In: Proceedings of CGAMES 06, Dublin, Ireland (2006) 126 — 135

[10] Stenberg, R. J., *Kognitivní psychologie.* , Portál, Praha, (2002).

[11] Fodor, J. A., *The Modularity of Mind* , MIT/Bradford Books, Cambridge, (1983).

[12] Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., Qin, Y., *An integrated theory of the mind* , Psychological Review 111, (2004) 1036-1060.

[13] Anderson, J. R., *How can the human mind occur in the physical universe?*, Oxford University Press, New York, (2007).

[14] Duchi, J. C. , Laird, J. E., *Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot* , AAAI Press 2000, (2000) 75-79.

[15] Best, B. J., Lebiere, C., *Cognitive agents interacting in real and virtual worlds.* , Cambridge University Press, New York, (2006) 186-218.

[16] Franklin, S., *Artificial Minds* , MIT Press, Cambridge, (1995).

[17] Best, B., Lebiere, C., Scarpinatto, C. *A model of synthetic opponents in MOUT training simulations using the ACT-R cognitive architecture.* , Orlando, FL. (2002).

[18] Best, B., Lebiere, C. *Spatial Plans, Communication, and Teamwork in Synthetic MOUT Agents.* , In Proceedings of the 12th Conference on Behavior Representation In Modeling and Simulation (2003).

[19] LehMan, J.F., Laird, J., Rosenbloom, P. *A gentle Introduction to Soar, an Architecture for Human Cognition: 2006 Update.* , University of Michigan (2006).

[20] Laird, J. E., Assanie, M., Bachelor, B., Benninghoff, N., Enam, S., Jones, B., Kerfoot, A., Lauver, C., Magerko, B., Sheiman, J., Stokes, D., Wallace, S. *A Test Bed for Developing Intelligent Synthetic Characters.* , AAAI Technical Report (2002).

[21] Lewinskim, M., Demir, C., Anantharam, R. *Incorporating Team Strategies in Bots for Capture the Flag mode of Unreal-Tournament 2004, URL: http://mlewi.com/portfolio.php*

[22] Fu, W. T. *An ACT-R adaptive planner in a simple map-navigation task.*, Proceedings of the Fifth International Conference on Cognitive Modeling (2003).

[23] Dye, H. A. *A diagrammatic reasoning: Route planning on maps with ACT-R.*, Proceedings of the Eigth International Conference on Cognitive Modeling, Ann Arbor, MI. (2007).