5 Návrh počítačom riadeného hráča

Táto kapitola obsahuje návrh počítačom riadeného hráča pre hru TACTIX (ďalej označovaného skratkou AIPT – Artificial Intelligence Player for Tactix). Pri jeho návrhu sa vychádzalo z požiadaviek, definovaných v kapitole 1.3.

AIPT možno chápať ako rozumného agenta, hlavného manažéra, koordinujúceho činnosti jemu podriadených entít. AIPT teda ako agent bude vnímať svoje prostredie pomocou senzorov a konať pomocou efektorov. Senzory sú založené na analýze arény a postavenia jednotlivých entít (svojich aj súperových). Efektory pozostávajú výhradne z príkazov posielaných do `Master` modulu, pomocou ktorých je možné hýbať entitami v aréne a útočiť na súperove entity (pozri kapitolu 3.3).

5.1 Autonómny agent

Výskum rozumných agentov je široká oblasť pokrývajúca rôzne témy. Existuje niekoľko typov rozumných agentov, v závislosti od spôsobu ich fungovania a cieľa, ktorý majú splniť. Pre AIPT je nutné vybrať taký druh rozumného agenta, ktorý bude najvhodnejší vzhľadom na definovaný model hry.

Medzi základné typy rozumných agentov možno počítať nasledovné [7] [8]:

- **Distribuované riešenie problémov.** Koncept takto navrhnutých agentov možno využiť na zjednodušenie riešenia veľkých problémov ich rozdelením a distribúciou medzi samostatné spolupracujúce jednotky, riešiace jednotlivé podproblémy. Každý agent v takomto systéme má svoje individuálne ciele, neexistuje žiaden hlavný spoločný cieľ.
- **Multiagentové systémy.** Výskum v oblasti multiagentových systémov sa zaoberá všeobecnými konceptmi organizácie, distribúcie úloh a dynamických zmien v organizácií (napr. tímové formácie alebo základné komunikačné mechanizmy).
- **Autonómne agenty.** Výskum v oblasti autonómnych agentov je primárne zameraný na realizáciu jedného samostatného agenta. To zahŕňa témy ako vnímanie, výber akcií alebo plánovanie.

AIPT bude navrhovaný ako autonómny agent. Podstatou hry TACTIX je riadenie distribuovaných entít v aréne. Tieto entity sú však všetky bez obmedzenia viditeľné a ovládateľné, čo umožňuje ich centralizované riadenie. Pre ich čo najefektívnejšie riadenie je

teda výhodnejšie použiť „centrálnu moc“, skôr ako každú entitu chápať ako samostatne jednajúceho agenta snažiaceho sa splniť svoj cieľ. Vytvorenie AIPT ako autonómneho agenta z časti vyplýva aj z cieľov projektu (kapitola 1.3), kde je definovaná snaha o vytvorenie hráča (agenta) schopného konať rozumne, plánovať a dynamicky reagovať na vzniknuté situácie.

Nasledujúce podkapitoly klasifikujú najčastejšie používané architektúry pri návrhu autonómneho agenta z pohľadu jeho vynakladania s výpočtovým časom a realizáciou sofistikovaného cieľovo orientovaného správania [9].

5.1.1 „Reactive“ agent

Reaktívne agenty pracujú v napevno nastavenom *„stimul – odpoveď“* štýle. Príkladom takéhoto druhu správania sa agentov sú napr. systémy Eliza [10] alebo Pengi [11]. Pre danú informáciu zo senzorov je vykonaná zodpovedajúca špecifická akcia. Takéto správanie je zvyčajne implementované pomocou množiny jednoduchých *„if – then“* pravidiel.

Ciele agenta sú implicitne reprezentované pravidlami, pričom môže byť komplikované týmito pravidlami zabezpečiť požadované komplexné správanie. Každá situácia musí byť vždy známa už pri návrhu pravidiel. Napríklad situácia, kedy jedna helikoptéra sleduje druhú môže byť realizovaná množinou pravidiel, pričom jedno z pravidiel pre sledujúcu helikoptéru by mohlo vyzerať nasledovne:

```
IF (leading_helicopter == left) THEN
  turn_left
ENDIF
```

Ak však programátor nenapíše reakcie (pravidlá) na všetky možné udalosti, správanie sa agenta (v tomto prípade sledujúcej helikoptéry) prestane vykazovať známky rozumnosti. Ak napríklad v uvedenej množine pravidiel zabudne uviesť pravidlo, ktoré ukončí sledovanie pri havárii sledovanej helikoptéry, bude to mať za následok haváriu aj sledujúcej.

Reaktívne systémy pracujúce v komplexnom prostredí obsahujú bežne stovky pravidiel, čo ich robí veľmi zložitými na naprogramovanie, aj na spravovanie. Pri takomto množstve pravidiel je taktiež veľmi zložité udržať požadované komplexné správanie sa agenta. Z tohto dôvodu model reaktívneho agenta nie je vhodný pre navrhnutie AIPT v potencionálne veľmi zložitom prostredí hry TACTIX. Na druhú stranu, výhoda reaktívneho agenta je v jeho schopnosti reagovať veľmi rýchlo. Táto vlastnosť však sama o sebe pre návrh AIPT nepostačuje.

5.1.2 „Triggering“ agent

Triggering agent oproti reaktívnemu obsahuje navyše vnútorný stav. Príkladmi takýchto agentov sú ALife agentové systémy Creatures [12] alebo Virtual Petz [13]. Upravené pravidlo z predchádzajúcej kapitoly by mohlo vyzerať nasledovne:

```
IF (distribution_mode) AND (leading_helicopter == left) THEN
  turn_right
  trigger_acceleration_mode
ENDIF
```

Tento druh agentov je v počítačových hrách používaný veľmi často, pričom na implementáciu stavov sa využíva konečný stavový automat. Agent dokáže reagovať rovnako rýchlo ako reaktívny agent a taktiež má schopnosť dosahovať dlhodobé ciele. Stále však je založený na napevno definovaných pravidlách a nevie správne reagovať na situácie, ktoré neboli vopred odhalené programátorom. Ďalej stále pretrváva problém s veľkým počtom pravidiel pre komplexné prostredie.

5.1.3 „Deliberative“ agent

Deliberative agent je založený na principiálne odlišnom prístupe. Ciele a model sveta obsahujú informácie o aplikačných požiadavkách a dôsledky akcií sú reprezentované explicitne. Interný plánovací systém, založený na zjemňovaní, používa informácie z modelu sveta na stavbu plánu k dosiahnutiu cieľa.

Takýto druh agentov nemá problém s dosahovaním dlhodobých cieľov. Implementácia všetkých špeciálnych pravidiel môže byť vynechaná, pretože plánovací systém sám dokáže vytvoriť cieľovo riadené plány akcií. Keď má agent vykonať svoju ďalšiu akciu, použije interný plánovací systém:

```
IF (current_plan_is_not_applicable_anymore) THEN
  recompute_plan
ENDIF
execute_plan's_next_action
```

Pomocou všeobecných usudzovacích metód [1] je možné vyriešiť aj nepredvídané situácie vhodným spôsobom. Problém tohto typu agenta je však v jeho nízkej rýchlosti, resp. vo výpadkoch v odozve. Vždy, keď sa nastaná situácia líši od situácie očakávanej plánovacím systémom agenta, je nutné celý plán znova prepočítať. Toto prepočítanie môže byť veľmi časovo náročné, čo spôsobí spomalenie reakcií agenta. V zložitom, stále sa meniacom prostredí, je použitie takéhoto agenta vzhľadom na výpočtové nároky prakticky vylúčené. Tento druh agenta sa tým pádom nehodí pre model pre AIPT. Avšak tvorba plánu ako prostriedku na dosahovanie dlhodobých cieľov je vhodný nástroj, použiteľný aj pre AIPT. Je ale nutné nejakým spôsobom znížiť výpočtovú náročnosť tvorby plánu a zamedziť jeho stálemu prepočítavaniu.

5.1.4 „Hybrid“ agent

Tento druh agentov aplikuje tradičný usudzovací plánovač, popisovaný v predchádzajúcej kapitole, na vysoko-úrovňové plánovanie a necháva rozhodnutia ohľadom vedľajších alternatív v samostatných krokoch plánu na reaktívne komponenty, riedené množinou jednoduchých *„if – then“* pravidiel. Príkladmi takýchto agentov sú T3 robot [14] alebo New Millennium Remote Agent [15].
Vnútorný systém je možné popísať nasledovne:

```
IF (current_plan-step_refinement_is_not_applicable_anymore) THEN

  WHILE (no_plan-step_refinement_is_possible) DO
    recompute_high-level_plan
  ENDWHILE
  use_hard-wired_rules_for_plan-step_refinement

ENDIF
execute_plan-step_refinement's_next_action
```

Je tu jasne vidieť hranica medzi vysoko-úrovňovým plánovaním a napevno definovaným reakčným správaním. Pre komplexné a rýchlo sa meniace prostredie, ako prostredie v počítačových hrách (a teda aj v hre TACTIX), však tento prístup nie je vhodný. Pretrváva tu ten istý problém ako pri deliberative agentoch, teda veľká výpočtová a časová náročnosť tvorby plánu (aj keď agent neprestane reagovať vďaka reaktívnej časti). A ak by aj plánovač dostal dostatok času, výsledkom by bol plán riešiaci situáciu, ktorá sa však už mohla úplne zmeniť.

5.1.5 „Anytime“ agent

Anytime agent je založený na kontinuálnom prechode z reaktívneho správania na plánovanie. Vždy musí mať prístupný plán, bez ohľadu na to, koľko výpočtového času má k dispozícií a koľko z plánu už vypočítal. Toto sa dá dosiahnuť opakovaným postupným vylepšovaním plánu. Ak má agent vykonať akciu, vylepší svoj plán (v rámci stanoveného časového limitu) a potom vykoná nasledujúcu akciu vylepšeného plánu:

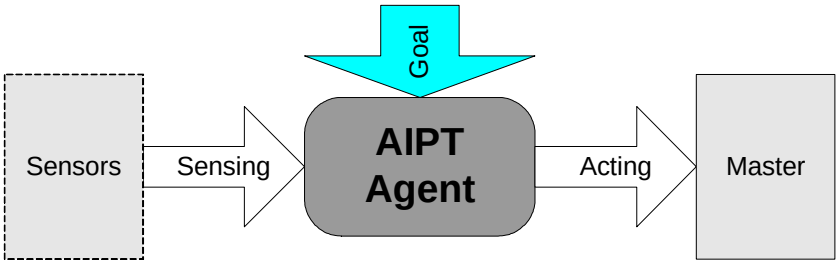
```
WHILE (computation_time_available) DO
  improve_current_plan
ENDWHILE
execute_plan's_next_action
```

Pre krátky výpočtový časový limit sú prístupné iba veľmi primitívne plány – reaktívne správanie. Dlhší časový limit umožňuje agentu vylepšiť, optimalizovať, prípadne pozmeniť plán, resp. časti plánu. Čím viacej výpočtového času je k dispozícií, tým inteligentnejšie správanie je možné od agenta očakávať. Postupné vylepšovanie častí plánu navyše otvára možnosti na jednoduchú adaptáciu plánu na zmenené alebo neočakávané situácie. Táto trieda agentov slúžila ako základná technológia pre projekt EXCALIBUR [9].

Návrh AIPT sa najviac inšpiroval myšlienkou postupného budovania a vylepšovania plánu. Vyjadrenie a tvorenie plánu pomocou grafov je zasa inšpirované projektom EXCALIBUR. Rozhodnutie pre použitie takejto technológie stavby plánu bolo urobené na základe potencionálnej možnosti vytvoriť dostatočne rýchly, komplexný a adaptabilný plánovací systém. Podrobný návrh AIPT agenta sa nachádza v kapitole 5.2. Vo fáze návrhu nie je možné presnejšie odhadnúť konečnú výpočtovú náročnosť plánovacieho systému, ale na základe analýz jednotlivých typov agentov sa model postupného budovania a vylepšovania plánu ukázal ako najviac vyhovujúci požiadavkám na AIPT.

5.2 Model AIPT agenta

Model autonómneho AIPT agenta je na obrázku 3. Do agenta vstupujú vnemy zo senzorov, agent sa snaží splniť cieľ a vytvára plán na základe ktorého koná pomocou efektorov, teda posiela príkazy do `Master` modulu.



Obr. 3 Model autonómneho AIPT agenta

Jadrom AIPT agenta je plánovací systém, ktorý na základe cieľa a informácií zo senzorov vytvorí a postupne modifikuje plán. Princíp fungovania AIPT agenta možno popísať nasledovne:

```
WHILE (current_plan_not_valid)
  improve_current_plan
ENDWHILE
execute_plan's_actual_actions
```

Na tomto pseudokóde je vidieť rozdiel medzi Anytime agentom a AIPT agentom. AIPT agent neberie do úvahy voľný výpočtový čas, ale upravuje plán, pokiaľ nespĺňa podmienky validity. Je teda nutné navrhnúť takú reprezentáciu plánu a mechanizmy na jeho modifikáciu, aby agent dokázal plán kontrolovať a modifikovať s dostatočnou rýchlosťou, a zároveň tak, aby na základe plánu vykazoval požadované správanie. Táto rýchlosť je podmienená dynamikou hry TACTIX. Záleží na reálnej rýchlosti pohybu jednotlivých jednotiek, resp. na rýchlosti zmien stavu hry, na ktoré musí byť AIPT agent schopný na reagovať s dostatočnou malou odozvou. Dynamika hry TACTIX (rýchlosti jednotiek a pod.) je nastavovateľná, takže ju bude možné prípadne prispôsobiť, ak sa ukáže že agent „nestíha“.

Nasledujúce kapitoly popisujú mechanizmy reprezentácie a tvorby plánu, jeho kontinuálnej validácie a modifikácie a prevedenie plánu do konkrétnych akcií (efektorov).

5.3 Reprezentácia plánu

Reprezentácia plánu vychádza z predpokladu, že na cieľ AIPT agenta je možné pozerať ako na problém rozložiteľný na podproblémy. Plán bude reprezentovaný pomocou upraveného A/ALEBO grafu [1].

Graf (strom) plánu bude obsahovať iba hrany A. Koreň stromu zodpovedá začiatočnému stavu (cieľu) a listy elementárnym akciám, resp. príkazom do `Master` modulu (pozri

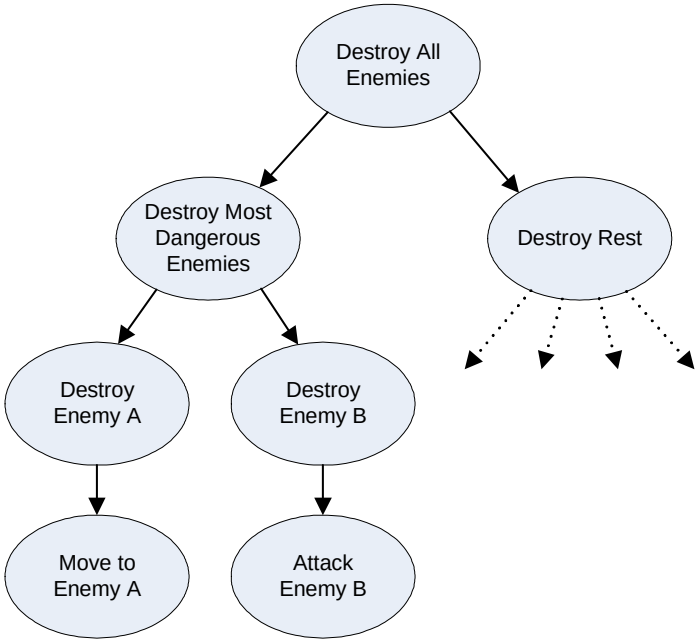
kapitolu 3.3). Hrany ALEBO, spájajúce uzol s alternatívami riešenia, nie sú pre reprezentáciu plánu v dynamicky sa meniacom prostredí zaujímavé, nakoľko tieto alternatívy sú už po veľmi krátkom čase nepoužiteľné.

Strom plánu v konkrétnom diskrétnom časovom okamžiku predstavuje riešenie problému pre aktuálny stav prostredia. Toto riešenie je podgraf v grafe priestoru rozkladov. Graf priestoru rozkladov je implicitne definovaný:

- **Začiatočným stavom.** Začiatočný stav je hlavný cieľ AIPT agenta. Vo všeobecnosti môže nadobúdať hodnoty ako *„vyhrať hru“*,*„poraziť súpera“* alebo *„získať viacej bodov ako súper“*.
- **Množinou rozkladajúcich operátorov.** Tieto operátory rozkladajú problém na podproblémy. Ich reprezentácia je popísaná v kapitole 5.4. Základnou podmienkou kladenou na množinu operátorov je ich schopnosť postupne rozložiť problém až na elementárne akcie – príkazy do `Master` modulu.

Príklad jednoduchého stromu plánu je na obrázku 4. Uzlom stromu je hlavný cieľ agenta – zničiť všetkých nepriateľov. Tento problém je rozložený na dva podproblémy, zničiť najviac nebezpečné jednotky a ostatné jednotky. V tomto prípade boli ako najnebezpečnejšie súperove jednotky určené jednotky A a B. Teda problém zničenia najnebezpečnejších jednotiek je rozložený na podproblémy zničenia jednotky A a zničenia jednotky B. Ak v tomto konkrétnom časovom okamžiku je jednotka A mimo dosahu zničiteľnosti, je nutné sa k nej presunúť. Jednotka B je v dosahu, teda je možné problém jej zničenia rozložiť na akciu zaútočenie. Zničenie ostatných jednotiek je naplánované obdobne.

Popisovaný plán je validný iba pre konkrétny časový okamžik. Ak napríklad v ďalšom priebehu hry súperova jednotka B zmení svoju polohu tak, že zmizne z dosahu zničiteľnosti, je nutné úsek plánu, ktorý je zodpovedný za jej zničenie, preplánovať (v tomto prípade by išlo iba o jeden uzol). Keďže podobné zmeny stavy hry sa budú diať veľmi často, je nevyhnutné návrh rozkladových operátorov koncipovať tak, aby nedochádzalo k častému rušeniu a novému plánovaniu väčšej časti plánu (stromu).



Obr. 4 Príklad jednoduchého plánu AIPT agenta

Uzol stromu plánu predstavuje úlohu (`Task`). Každá úloha obsahuje niekoľko parametrov (slotov), ktorými je definovaná. Nasledovné tri sa nachádzajú v každej úlohe:

- **Názov.** Názov úlohy v skratke vyjadruje jej hlavnú podstatu. Názvu zároveň určuje aj typ úlohy. Tento typ je využívaný pri aplikácií rozkladových operátorov.
- **Jednotky agenta.** Ide o jednotky, ktoré má agent pridelené na túto úlohu. Jedna jednotka môže byť pridelená do viacerých úloh iba vtedy, ak ide o navzájom závislé úlohy (teda ak jedna úloha je podúlohou druhej). Úlohy na rovnakej úrovni nesmú mať spoločné jednotky. Inak povedané, každá jednotka v konkrétnom časovom okamžiku plní jednu úlohu. Implicitne sa v úlohe nachádzajú iba živé jednotky. Akonáhle je jednotka zničená, je odstránená z úlohy, takže s ňou nie je možné ďalej manipulovať a prideľovať ju na podúlohy.
- **Podmienka validity.** Táto podmienka určuje, kedy má zmysel snažiť sa plniť danú úlohu. Ak podmienka validity nie je splnená, úlohu je nutné zrušiť. Príkladom jednoduchého testu validity pre úlohu *„znič súperovu jednotku“* by mohol byť test, či táto súperova jednotka je stále nažive.

Ďalšie sloty sú závislé od typu úlohy. Môže ísť napríklad o jednotky súpera, ktorých sa úloha týka a podobne.

Listy v strome plánu predstavujú elementárne akcie (`Actions`), posielané do `Master` modulu. Sú definované rovnako ako ostatné úlohy, avšak majú svoje špecifické parametre (napr. miesto v aréne kam sa pohnúť alebo zaútočiť).

5.4 Rozkladové operátory

Rozkladové operátory sú hlavnou časťou plánovača. Definujú tvorbu plánu a teda aj správanie sa AIPT agenta. Úplná množina operátorov musí spĺňať niekoľko podmienok:

- **Úplný rozklad.** Operátory musia byť navrhnuté tak, aby postupne rozkladali hlavný cieľ (hlavnú úlohu) na podúlohy až do „jemnosti“ na elementárne akcie. Na každý typ úlohy sa musí dať aplikovať nejaký operátor, pokiaľ sa nejedná o elementárnu akciu.
- **Acyklickosť.** Postupné aplikovanie operátorov na úlohu typu A nesmie viesť opäť na úlohu typu A s rovnakými parametrami. Toto sa dá dosiahnuť napríklad vytvorením úrovní pre typy úloh. Operátor nebude môcť úlohu rozložiť na podúlohy vyššej úrovne ako je tá, ktorú rozkladá.
- **Perzistencia plánu.** Táto podmienka vyjadruje požiadavku na čo najmenšie rušenie úloh a následné preplánovanie. Možností na dosiahnutie tejto podmienky je niekoľko. Napríklad je výhodné v nízkej hĺbke stromu rozkladať úlohy postupne zo všeobecnejších s viacerými jednotkami po konkrétne pre malý počet jednotiek. Konkrétne úlohy pre malý počet jednotiek sú viacej nestabilné (v zmysle dynamiky hry) ako všeobecnejšie. Skúmanie týchto možností a hľadanie optima bude predmetom pokusov pri implementácií plánovača.

Operátory sú definované pomocou grafu. Definícia použitej grafovej syntaxe a aritmetiky sa nachádza v kapitole 5.4.1. V kapitole 5.4.2 sú uvedené príklady rozkladových operátorov a vysvetlené spôsoby ich aplikácie.

5.4.1 Grafová syntax operátorov

Každý operátor sa skladá z dvoch hlavných častí:

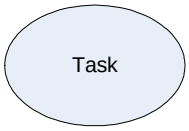
- **Podmienková časť.** Podmienková časť určuje, na akú úlohu a za akých podmienok možno použiť operátor. Podmienková časť je rozdelená na dve podčasti. Prvá, štrukturálna podmienka, je založená na grafovej podobnosti („*match*“) podmienkovej časti operátora a tej časti stromu, na ktorú chceme operátor použiť. Východzí pevný uzol, od ktorého prebieha kontrola zodpovedania stromu a operátora („*matching*“), je

označovaný ako proxy uzol. V grafe operátora musí byť práve jeden proxy uzol. Druhá časť podmienkovej časti je založená na kontrole konkrétnych parametrov proxy uzla (úlohy).

- **Produkčná časť.** Produkčná časť obsahuje uzol, prípadne uzly, ktoré sa po splnení podmienkovej časti pridajú do aktuálneho stromu plánu ako deti proxy uzla operátora. Zároveň obsahuje aj vyjadrenie parametrov pre produkované úlohy (pozri kapitolu 5.3).

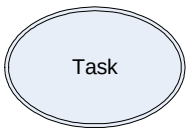
Graf definujúci operátor môže obsahovať nasledovné typov uzlov:

- **Úloha (`Task`).** Vyjadruje úlohu zo stromu plánu (obrázok 5). Obsahuje typ úlohy.



Obr. 5 Úloha

- **Proxy úloha (`Proxy Task`).** Podobne ako úloha, avšak tento uzol označuje proxy úlohu (obrázok 6). Obsahuje typ úlohy.

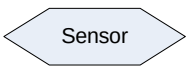


Obr. 6 Proxy úloha

- **Akcia (`Action`).** Vyjadruje elementárnu akciu – príkaz do `Master` modulu (obrázok 7). Obsahuje typ akcie.

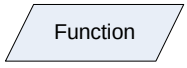
Obr. 7 Akcia

- **Senzor (`Sensor`).** Vyjadruje hodnotu senzoru (obrázok 8). Obsahuje meno senzoru.



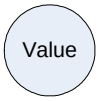
Obr. 8 Senzor

- **Funkcia (`Function`).** Vyjadruje použitie vopred implementovanej funkcie (obrázok 9). Obsahuje názov funkcie.



Obr. 9 Funkcia

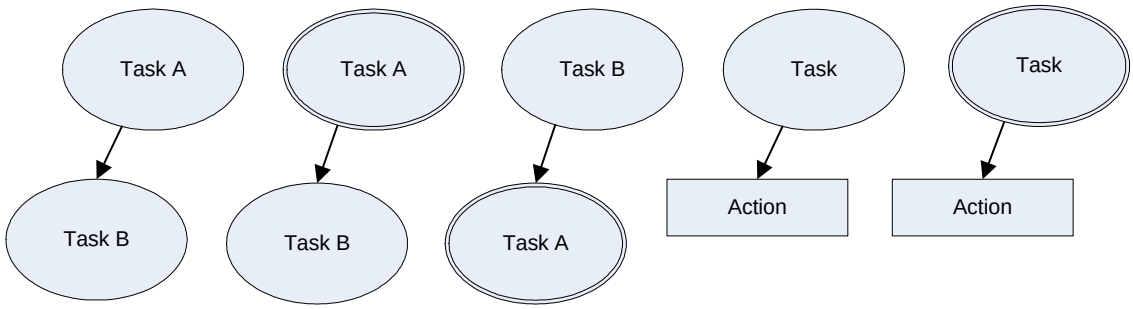
- **Hodnota (`Value`).** Vyjadruje hodnotu objektu s ním stotožnenú (obrázok 10). Môže ísť o ľubovoľný objekt (číselnú hodnotu, zoznam, funkciu, prípadne aj o zložitejšiu štruktúru).



Obr. 10 Hodnota

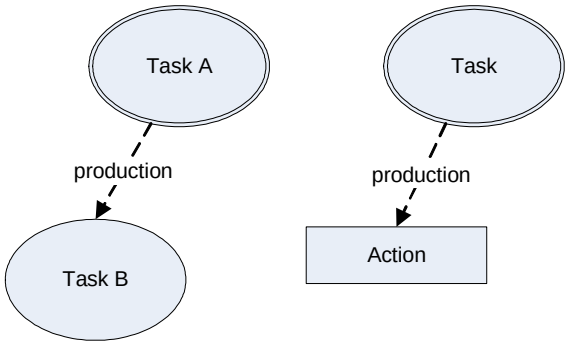
Uzly v grafe operátora je dovolené prepájať nasledovnými spôsobmi a typmi hrán:

- **Vyjadrenie štrukturálnej podmienky.** Štrukturálnu podmienku možno definovať iba na uzloch, ktoré sa vyskytujú v strome plánu, teda na úlohách a akciách. Povolené spôsoby prepojenia sú na obrázku 11.



Obr. 11 Vyjadrenie štrukturálnej podmienky medzi úlohami, proxy úlohami a akciou

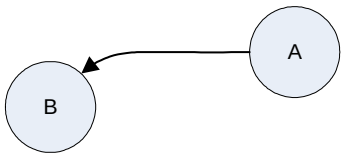
- **Vyjadrenie produkcie.** Operátor môže do stromu plánu produkovať nový plán alebo akciu. Tento nový uzol môže byť pripojený iba k uzlu, ktorý rozvíjame – teda k proxy uzlu. Grafické vyjadrenie produkcie je na obrázku 12.



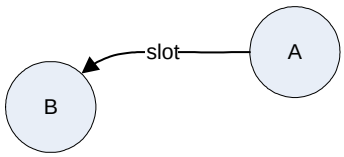
Obr. 12 Vyjadrenie produkcie

- **Stotožnenie.** Stotožnenie je vyjadrené pomocou jednoduchej orientovanej hrany. Orientácia hrany určuje, čo sa stotožňuje na čo. Na obrázku 13 je znázornené stotožnenie objektu B s objektom A. Hrana bez popisu znázorňuje stotožnenie celého

objektu. Ak má hrana popis, stotožňuje sa iba zodpovedajúci slot objektu. Na obrázku 14 je znázornené stotožnenie objektu B so slotom `slot` objektu A.

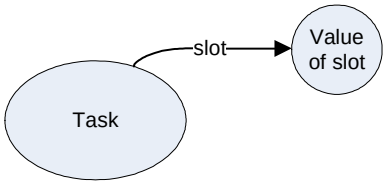


Obr. 13 Stotožnenie objektu

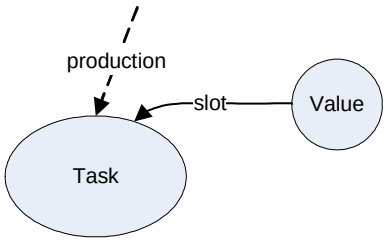


Obr. 14 Stotožnenie slotu

- **Stotožnenie na úlohe.** Ide o prípad stotožnenia slotu úlohy s hodnotou (objektom). Na obrázku 15 je zobrazené stotožnenie hodnoty `Value` so slotom `slot` úlohy `Task`. Na obrázku 16 je opačný prípad, kedy slot `slot` úlohy `Task` je stotožnený s hodnotou `Value`. Toto je však možné spraviť iba s úlohou, ktorá je operátorom produkovaná.

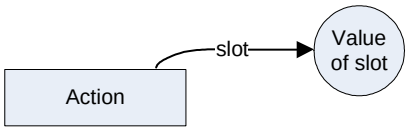


Obr. 15 Stotožnenie hodnoty so slotom úlohy

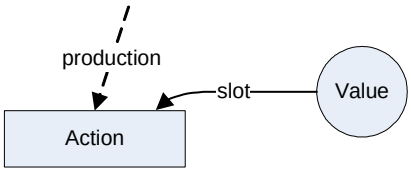


Obr. 16 Stotožnenie slotu úlohy s hodnotou

- **Stotožnenie na akcii.** Rovnaký prípad stotožnenia ako stotožnenie na úlohe (obrázky 17 a 18).

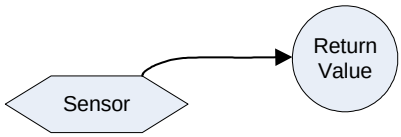


Obr. 17 Stotožnenie hodnoty so slotom akcie

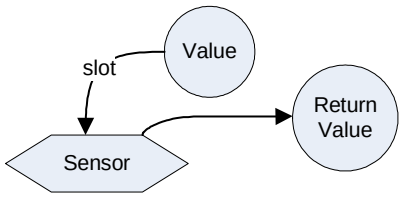


Obr. 18 Stotožnenie slotu akcie s hodnotou

- **Stotožnenie na senzore.** Hodnotu senzora možno stotožniť s objektom (obrázok 19). Ak je senzor parametrický, je nutné stotožniť hodnotu jeho parametra (slotu) s nejakou hodnotou. Na obrázku 20 je parameter `slot` senzora stotožnený s hodnotou `Value`.

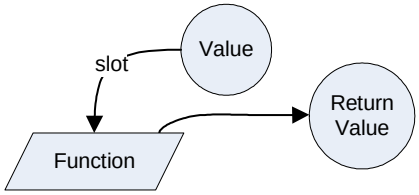


Obr. 19 Stotožnenie hodnoty so senzorom



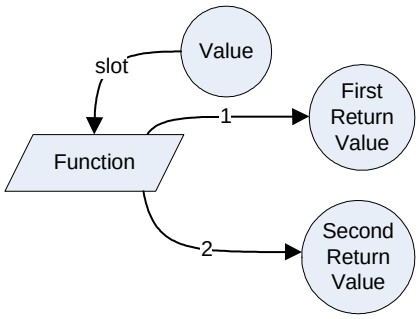
Obr. 20 Stotožnenie hodnoty so senzorom s parametrom

- **Stotožnenie na funkcii.** Rovnaký prípad ako stotožnenie na senzore, avšak funkcia má vždy vstupné parametre, ktoré je nutné s nejakou hodnotou stotožniť (obrázok 21).



Obr. 21 Stotožnenie hodnoty s výstupom funkcie

- **Stotožnenie na funkcii s viacej výstupmi.** Ak má funkcia viacej výstupov, číslo na hrane smerujúcej do hodnoty vyjadruje poradie výstupu s touto hodnotou stotožnenou (obrázok 22).



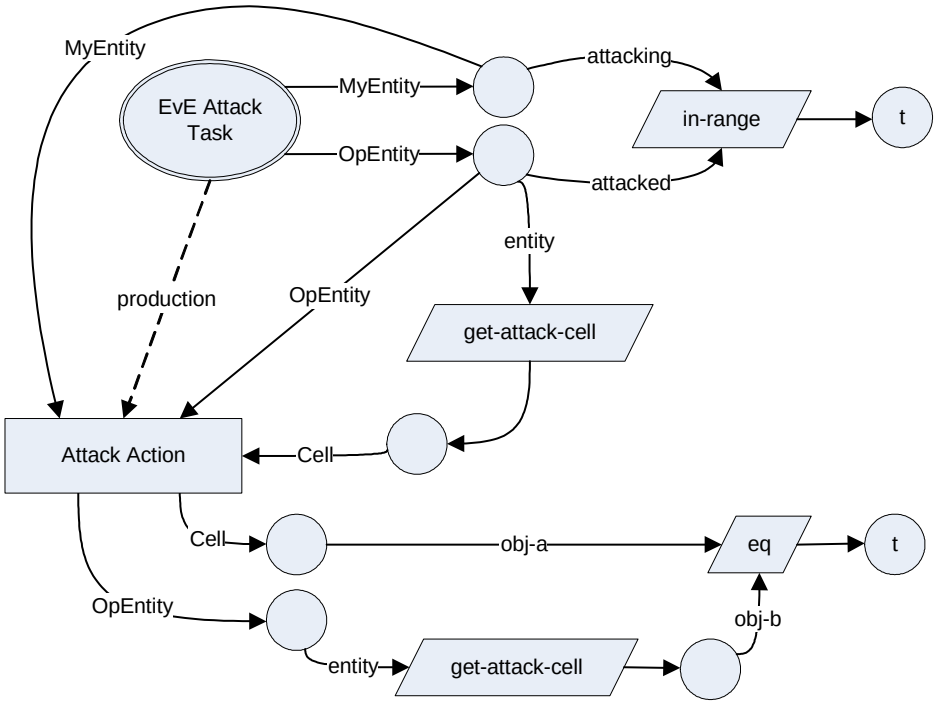
Obr. 22 Stotožnenie hodnôt s výstupmi funkcie

Pomocou uvedených pravidiel sú vytvorené všetky rozkladové operátory plánovacieho systému. V nasledujúcej kapitole sú uvedené a vysvetlené príklady rozkladových operátorov.

5.4.2 Príklady rozkladových operátorov

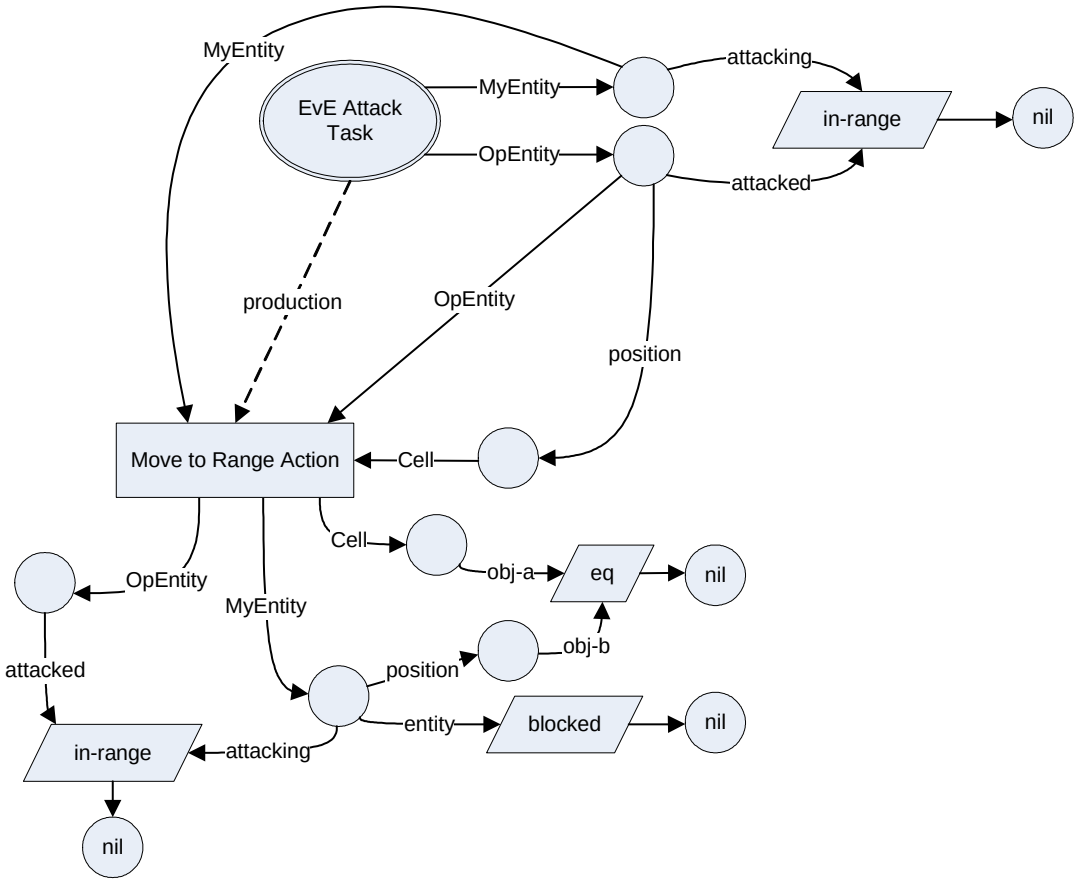
Nasledovné príklady rozkladových operátorov ilustrujú použitie a možnosti definovanej grafovej syntaxe. Sú uvedené od tých nižšie úrovňových po vyššie (abstraktnejšie). Ide iba o príklady, ktoré spolu netvoria úplnú množinu rozkladových operátorov (pozri začiatok kapitoly 5.4).

Na obrázku 23 je príklad rozkladového operátora, ktorý rieši úlohu útoku jednej agentovej entity na jednu súperovu entitu. Podmienka aplikovania tohto operátora je vyjadrená pomocou funkcie `in-range`, ktorá vráti hodnotu `t`, ak je súperova entita v dosahu (dostrele). Ak je táto podmienka splnená, do stromu plánu je pridaná akcia na útok agentovej entity na súperovu. Zároveň je v operátore aj vyjadrená podmienka, dokedy má táto akcia zmysel. Táto akcia má zmysel, dokedy je súperova entita na mieste, na ktoré útočíme. Funkcia `get-attack-cell` vráti miesto, kam treba zaútočiť, aby bola entita zasiahnutá (môže počítať aj s jej pohybom a podobne).



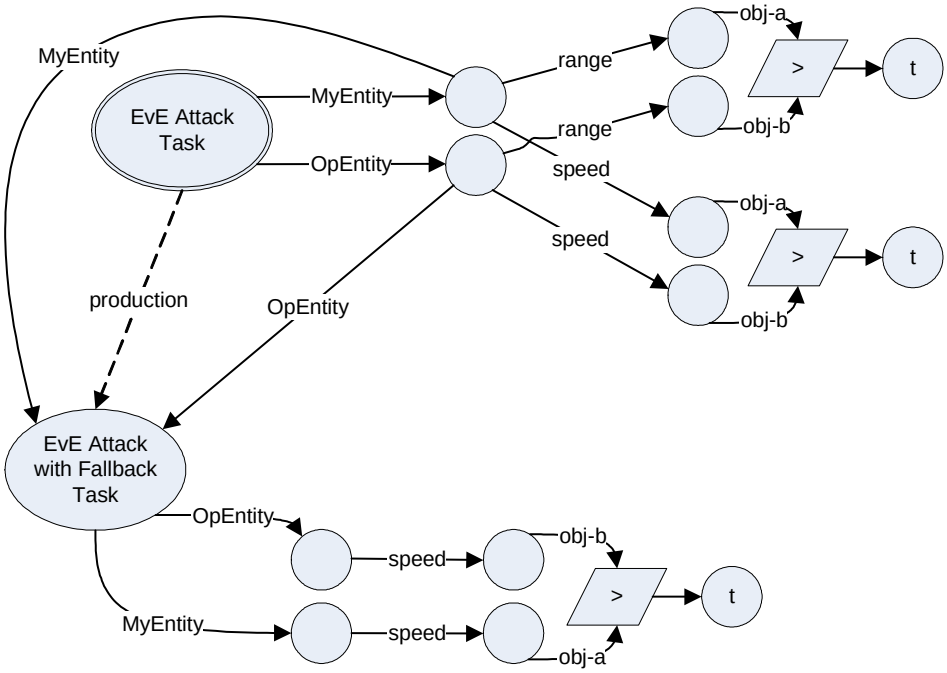
Obr. 23 Operátor rozkladu úlohy EvE Attack na akciu Attack

Na obrázku 24 je príklad rozkladového operátora, ktorý rozkladá ten istý typ úlohy, ale v prípade, že súperova jednotka nie je v dosahu (dostrele) agentovej jednotky. Úloha útoku na jednotku je potom rozvinutá na akciu presunu k súperovej jednotke. Zmysel tejto akcie pretrváva dovtedy, kým agentova jednotka buď prišla na zadané miesto, alebo sa nezablokovala (nemôže sa na dané miesto z rôznych príčin dostať), alebo sa nedostala na dostrel súperovej jednotky (súperova jednotka sa teoreticky mohla pohybovať proti agentovej jednotke, takže agentova jednotka je v dostrele skôr, ako sa predpokladalo pri aplikácií operátora).



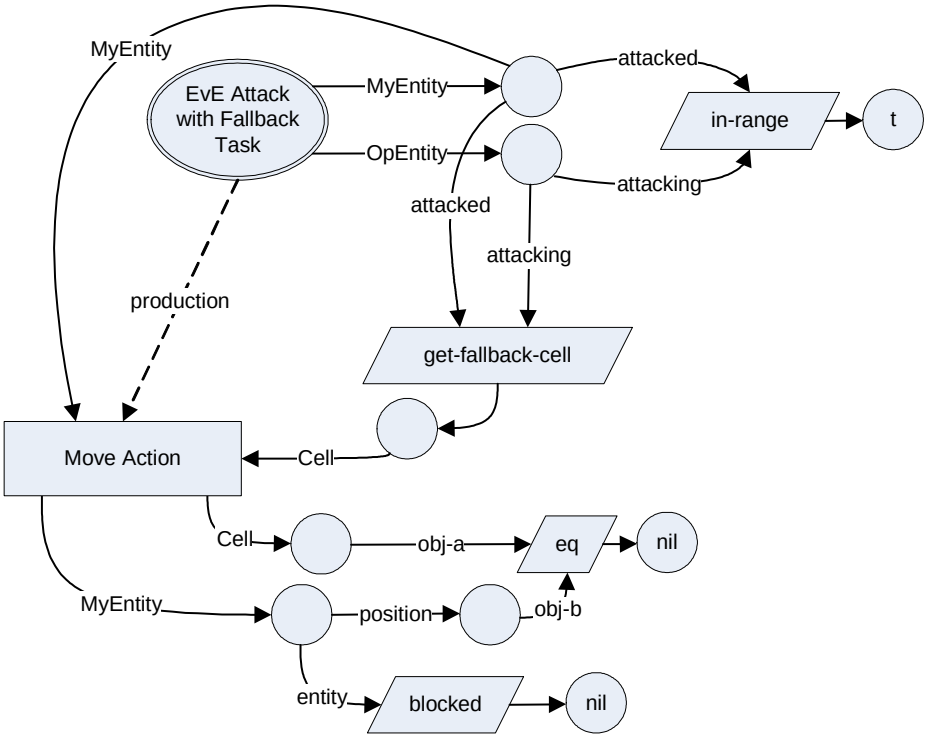
Obr. 24 Operátor rozkladu úlohy EvE Attack na akciu Move to Range

Takto vytvorené operátory útoku jednej entity na druhú možno rozšíriť o nové operátory, ktoré budú zohľadňovať výhodu väčšieho dostrelu a vyššej rýchlosti agentovej entity oproti súperovej. Na obrázku 25 je príklad rozkladového operátora, ktorý úlohu útoku na súperovu entitu rozloží na úlohu útoku z ústupom. Entita plniaca túto úlohu sa bude snažiť držať si súperovu entitu v dostrele, ale zároveň byť mimo dostrelu tejto entity. Tento operátor okrem toho, že vytvorí novú podúlohu, nastaví tejto podúlohe test validity tak, aby mala zmysel iba ak je agentova entita stále rýchlejšia ako súperova (vychádza sa z predpokladu, že rýchlosť entity sa môže meniť, napr. jej poškodením, ale dostrel nie).

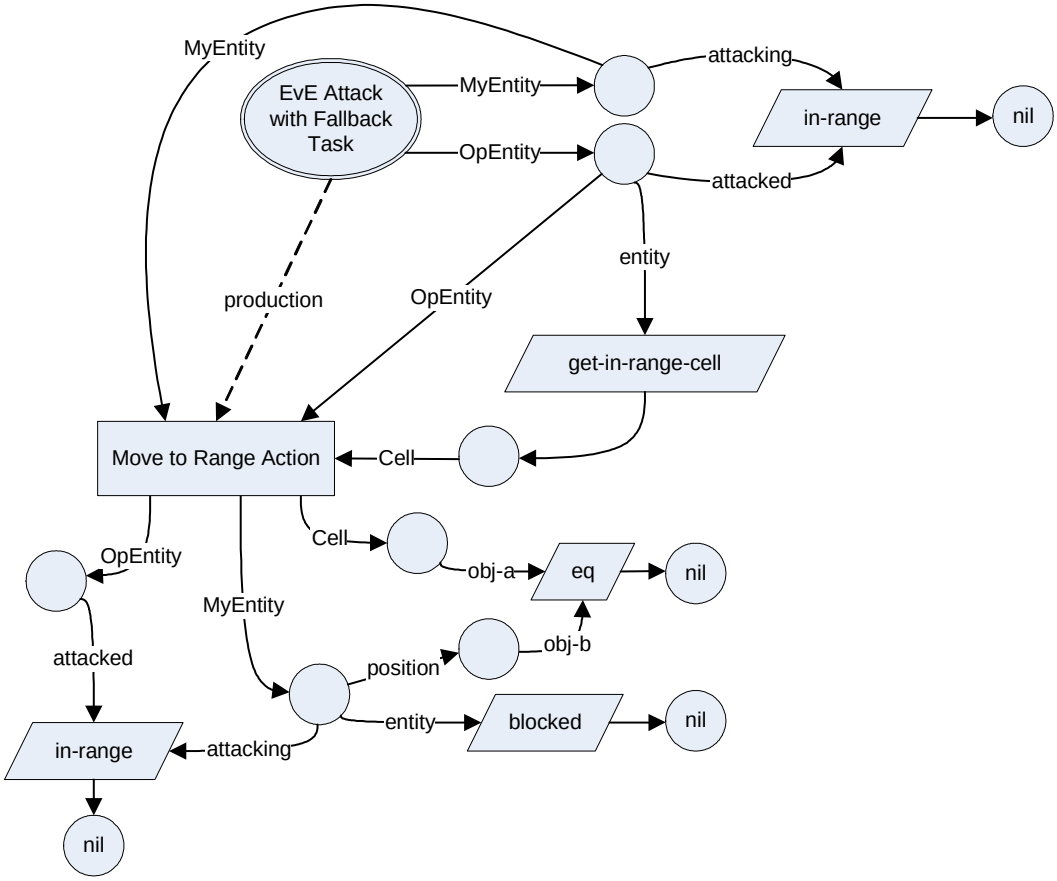


Obr. 25 Operátor rozkladu úlohy EvE Attack na úlohu EvE Attack with FallBack

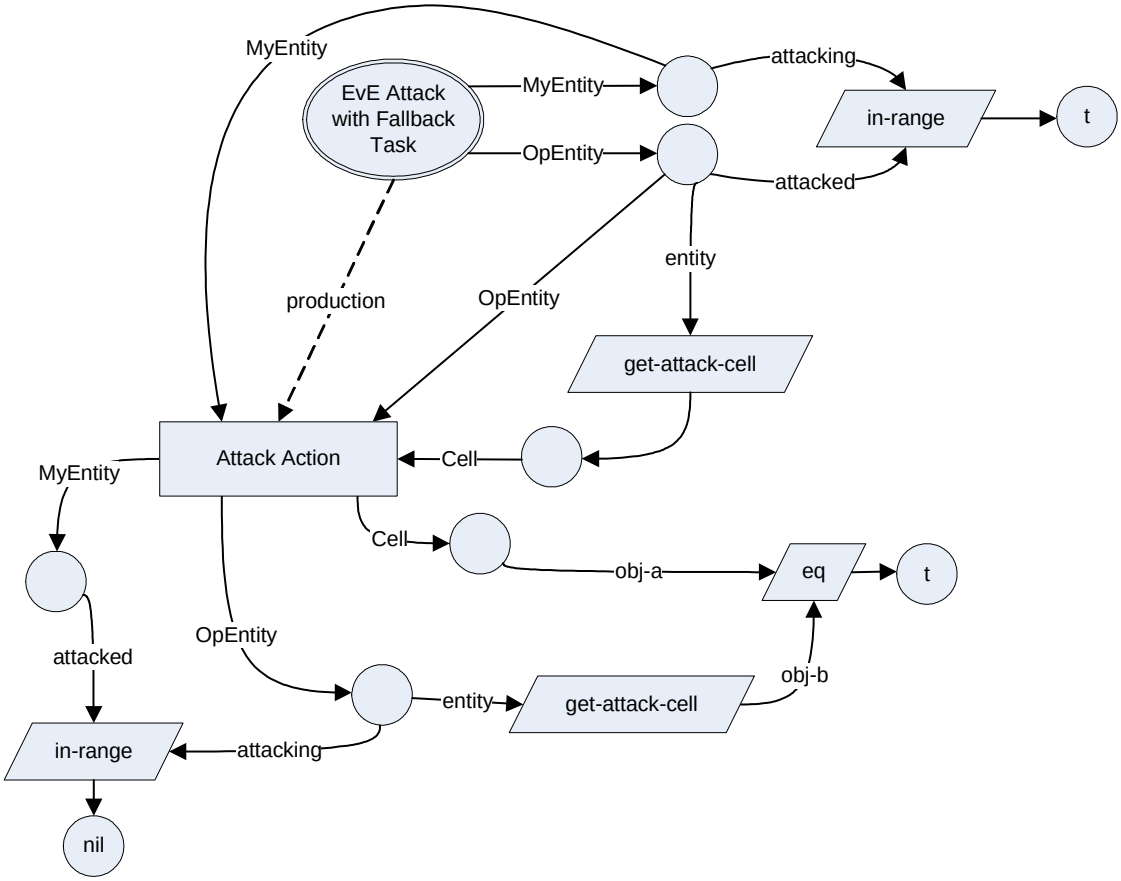
Novú úlohu útoku s ústupom potom možno rozkladať nasledovnými operátormi. Na obrázku 26 je operátor, ktorý je použiteľný, ak sa agentova entita nachádza v dostrele súperovej. Vtedy je úloha rozložená na podúlohu ústupu – akciu pohybu na nové miesto. Funkcia `get-fallback-cell` vráti najvodnejšie miesto pre ústup agentovej entity. Na obrázku 27 je naopak operátor, ktorý úlohu útoku s ústupom rozloží na podúlohu (akciu) priblíženia sa k súperovej entite, ak táto nie je v dostrele agentovej entity. Tieto typy operátorov uzatvára operátor na obrázku 28, ktorý je použiteľný, ak je súperova entita v dostrele agentovej, a spôsobí vytvorenie akcie útoku na súperovu entitu (s ohľadom na to, či sa agentova entita dostala do dostrelu súperovej). Tieto operátory musia mať uvedené poradie, resp. priority, aby bolo dosiahnuté požadované správanie.



Obr. 26 Operátor rozkladu úlohy EvE Attack with Fallback na akciu Move

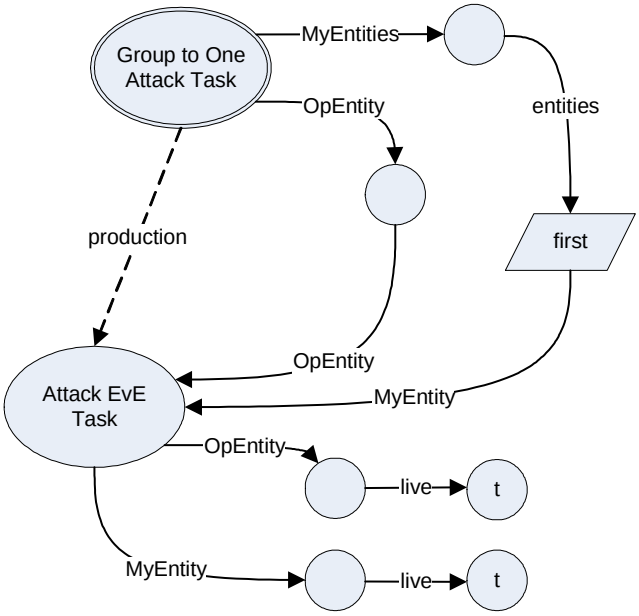


Obr. 27 Operátor rozkladu úlohy EvE Attack with Fallback na akciu Move to Range



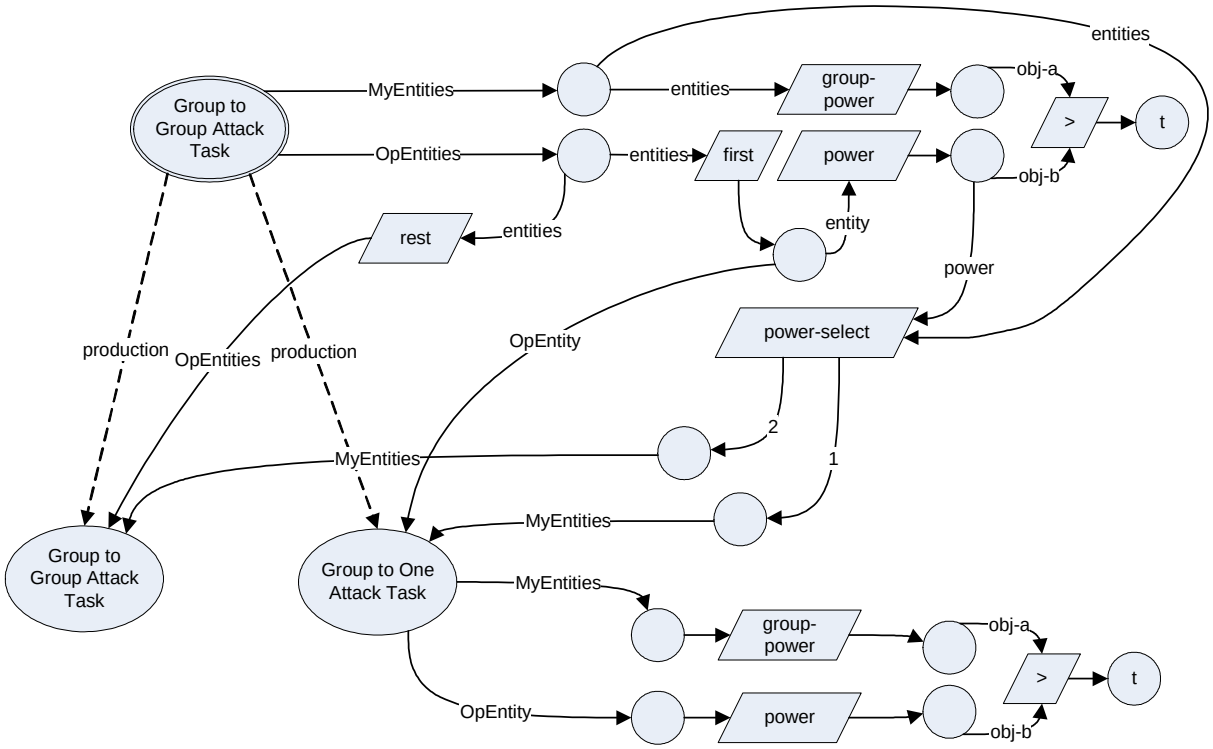
Obr. 28 Operátor rozkladu úlohy EvE Attack with Fallback na akciu Attack

Na obrázku 29 je príklad operátora, ktorý rozkladá úlohu útoku skupiny agentových entít na jednu súperovu. Úlohu rozkladá na pod úlohu útoku jednej entity na jednu. Táto pod úloha má zmysel iba vtedy, ak sú obe entity „nažive“.



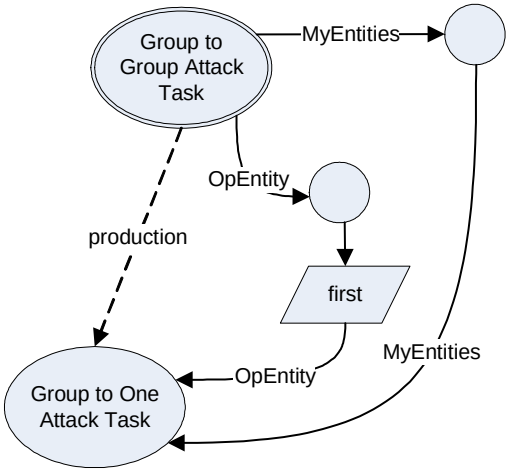
Obr. 29 Operátor rozkladu úlohy Group to Group Attack na úlohu Attack EvE

Relatívne komplikovanejší operátor na obrázku 30 rozkladá úlohu útoku skupiny agentových entít proti skupine súperových entít. Je použiteľný iba vtedy, ak má skupina agentových entít väčšiu „silu“ ako prvá zo skupiny súperových entít. Táto sila môže byť chápaná ako kumulovaná útočná sila entít spolu s ich stupňom poškodenia a podobne. Je vypočítaná vo funkcii `power` pre jednu entitu, resp. `group-power` pre skupinu entít. Pomocou funkcie `power-select` je vybraná taká skupina agentových entít, ktorá má väčšiu silu ako prvá súperova entita (prvá návratová hodnota). Operátor rozloží úlohu útoku skupiny na skupinu na dve nové pod úlohy. Jednou z nich je opäť rovnaká úloha, avšak bez vybraných agentových entít a so zvyškom entít súperových. Ide vlastne o určitý druh rekurzie. Druhou vytvorenou podúlohou je už popisovaná úloha útoku skupiny entít na jednu entitu. Táto podúloha má zmysel iba vtedy, ak je zachovaný predpoklad s akou bola vytvorená, teda že skupina agentových entít má väčšiu silu ako súperova entita.



Obr. 30 Operátor rozkladu úlohy Group to Group Attack na rovnaký typ úlohy a úlohu Group to One Attack

Operátor na obrázku 30 však nemusí byť vždy použiteľný. Môže vzniknúť (napr. postupným aplikovaním práve tohto operátora) úloha, kedy skupina agentových entít bude „slabšia“ ako jediná súperova entita. Aby táto skupina neostala v nečinnosti, je výhodné pridať ďalší operátor, ktorý nemá žiadne podmienky na aplikovanie a pošle zostávajúce agentove jednotky na súperovu (bez ohľadu na to, že je silnejšia). Tento operátor je na obrázku 31. Musí mať nižšiu prioritu ako operátor na obrázku 30, lebo tento by sa inak nikdy neaplikoval.



Obr. 31 Operátor rozkladu úlohy Group to Group Attack na úlohu Group to One Attack

5.5 Mechanizmus tvorby, validácie a modifikácie plánu

Tvorba plánu je založená na aplikácií rozkladových operátorov na úlohy. Tieto operátory sa aplikujú dovtedy, kým sa rozkladom nedospeje k akciám. Na akcie sa ďalej nedajú aplikovať žiadne operátory (pozri kapitolu 5.4.1, vyjadrenie produkcie).

Počas hry prebieha neustála kontrola validity a modifikácia plánu (v každom diskrétnom časovom okamžiku hry, kedy dostane AIPT výpočtový čas). Pri inicializácii AIPT vytvorí kompletný plán z jednej hlavnej úlohy (*„poraz súpera“* a podobne). Toto začiatočné vytvorenie plánu je založené na rovnakom mechanizme ako jeho postupná modifikácia.

Mechanizmus tvorby, resp. modifikácie plánu sa skladá z nasledovných dvoch krokov:

1. **Kontrola validity úloh.** Strom plánu sa postupne prechádza od koreňa až po listy. Metóda prechádzania je typu *„preorder“*. Najskôr sa skontroluje (zvaliduje) aktuálny uzol (aktuálna úloha) a potom jej potomkovia (jej rozklad na podúlohy). Ak test validity úlohy nie je úspešný, úloha je zrušená – je zmazaná so stromu spolu so všetkými jej potomkami až po listy. Teda je zmazaná celá vetva stromu od invalidného uzla vrátane tohto uzla. Úloha musí mať ďalej aspoň jednu pridelenú živú jednotku. Ak nemá, je rovnako zrušená.
2. **Rozkladanie úloh.** Strom sa prechádza rovnako ako v prvom kroku od koreňa až po listy. Ak sa nájde úloha, ktorej jednotky agenta nie sú všetky pridelené do podúloh, aplikuje sa na prvý vyhovujúci rozkladový operátor. Úloha je kompletne rozložená a ďalej sa nerozkladá iba vtedy, ak všetky jej pridelené agentove jednotky sú použité (pridelené) do jej priamych potomkov (podúloh). Hľadanie vyhovujúceho operátora prebieha ako kontrola zhody (*„matching“*) medzi operátorom a rozkladanou úlohou. Poradie skúšania operátorov je určené ich prioritou. Priorita operátora je vyjadrená jednoducho jeho poradím v zozname všetkých operátorov. Ak je úloha validná a obsahuje voľné jednotky, ale nepodarí sa na ňu aplikovať žiaden rozkladový operátor, ide o chybu v návrhu operátorov.

Listy plánu predstavujú akcie, ktoré má agent vykonať v konkrétnom časovom okamžiku. Keďže niektoré akcie pretrvávajú dlhší čas, bolo by zbytočné ich po každom prechode znova posielať do `Master` modulu. Stačí posielať iba novovzniknuté akcie. Toto správanie bude zabalené do samotnej funkcie akcie, neovplyvní popísaný mechanizmus tvorby a modifikácie plánu.

6 Kompilátor rozkladových operátorov

Táto kapitola obsahuje popis kompilátora rozkladových operátorov. Jeho hlavnou úlohou je previesť definíciu operátora ako grafu do jazyka, ktorý je možno skompilovať do natívneho kódu počítača. Návrh aj implementácia kompilátora vychádzali z použitia jazyka Common Lisp [16]. Kompilátor prevedie definíciu operátora, ktorá je vo forme grafu (vhodne vnútorne reprezentovaného) do kódu, ktorý je potom skompilovaný Common Lisp kompilátorom do natívneho kódu počítača.

Kompilátor rozkladových operátorov spolu so stromom plánu (kapitola 5.3) a mechanizmom vytvárania a modifikáciu plánu (kapitola 5.5) predstavuje jadro AIPT agenta. Je vystavaný pomocou knižnice PATG (Patterns On Graphs, http://common-lisp.net/project/patg/). PATG je Common Lisp knižnica na hľadanie vzorov (podgrafov) v grafoch, ako aj na modifikáciu týchto grafov. Je vytvorená ako doménovo špecifický jazyk, ktorý používa Lisp makrá na preklad deklaratívnych grafových vzorov na rýchly Common Lisp kód.

Kompilácia rozkladového operátora sa skladá z nasledovných krokov:

1. **Lexikálna analýza** (kapitola 6.2). V tomto prípade je triviálna, nakoľko sú lexikálne jednotky jasné už z vnútornej reprezentácie operátora (kapitola 6.1).
2. **Syntaktická analýza** (kapitola 6.3). Ide o kontrolu základných syntaktických pravidiel, definovaných v kapitole 5.4.1.
3. **Sémantická analýza** (kapitola 6.4). V tomto kroku sa skontroluje logická správnosť operátora a vytvoria sa pomocné podgrafy pre jeho jednotlivé časti (podmienková časť a produkčné časti).
4. **Generácia kódu** (kapitola 6.5). V tomto kroku je vygenerovaný samotný kód, ktorý predstavuje rozkladový operátor.

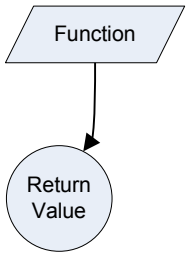
Jednotlivé kroky sú na sebe úzko závislé, a to v tom zmysle, že každý krok (algoritmus tohto kroku) predpokladá úspešné splnenie predchádzajúceho (samozrejme okrem prvého kroku). Kompilácia operátora sa preto zastaví na prvej nájdenej chybe.

6.1 Vnútorná reprezentácia rozkladových operátorov

Vnútorná reprezentácia rozkladového operátora vychádza z knižnice PATG. Táto umožňuje zadefinovať si vlastné typy uzlov a hrán, z ktorých je potom možné vytvoriť požadovaný

rozkladový operátor. Definícia uzlov a hrán vychádzala z kapitoly 5.4.1. Počas implementácie kompilátora sa však ukázalo výhodné urobiť niekoľko zmien oproti danému návrhu:

- **Odstránenie špeciálneho uzla pre senzor.** Ako sa ukázalo, senzor je z praktického hľadiska v podstate funkcia, ktorá nemá žiadne vstupné parametre (viditeľné z grafu operátora). Na základe tejto skutočnosti bol uzol senzor odstránený, ako aj zodpovedajúce syntaktické pravidlá. Namiesto toho bolo pridané syntaktické pravidlo, ktoré umožňuje v grafe rozkladového operátora použiť aj funkciu, ktorá nemá žiadne vstupné parametre. Toto pravidlo je znázornené na obrázku 32.



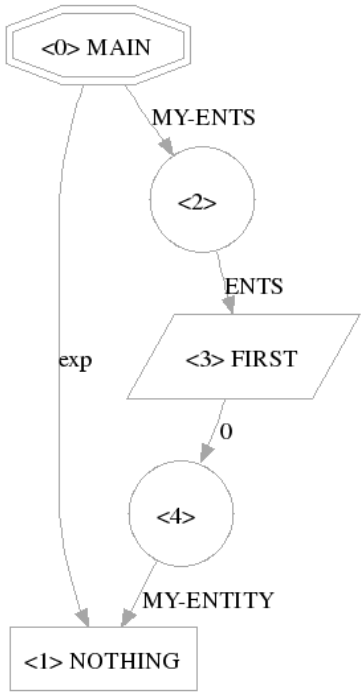
Obr. 32 Syntaktické pravidlo pre funkciu bez vstupného parametra

- **Číslovanie výstupných hrán funkcie od nuly.** Takéto číslovanie je bližšie zaužívaným štandardom v programovacích jazykoch.
- **Vizuálne zmeny v operátoroch.** Na generovanie grafickej reprezentácie rozkladového operátora bola použitá knižnica Graphviz. Z tohto dôvodu bolo treba urobiť niekoľko vizuálnych zmien. Tá najpodstatnejšia sa týka proxy úlohy. Nakoľko knižnica Graphviz nepodporovala zobrazenie elipsy vytvorenej pomocou dvojitej čiary, bolo nutné použiť namiesto elipsy osemuholník. Ďalšia zmena sa týka vyjadrenia produkcie, ktorá je ďalej reprezentovaná ako orientovaná hrana s neprerušovanou čiarou a s popisom `exp`. Všetky nasledujúce obrázky rozkladových operátorov sú generované pomocou knižnice Graphviz, takže obsahujú tieto vizuálne zmeny.

Rozkladový operátor je vnútorne zadefinovaný pomocou špecifického doménového jazyka. Tento jazyk je nadstavba nad doménovým jazykom, použitým v knižnici PATG. Na obrázku 33 je grafická reprezentácia rozkladového operátora, ktorého vnútorná deklaratívna reprezentácia je nasledovná:

```
((0 ref main -> (1 exp) (2 slot my-ents))
 (1 action nothing)
 (2 val -> (3 arg ents))
 (3 fun first -> (4 res 0))
 (4 val -> (1 slot my-entity)))
```

Syntax tohto doménového jazyka je relatívne jednoduchá. Ide o zoznam uzlov, pričom definícia každého uzla je zoznam pozostávajúci z dvoch častí, ktoré sú od seba oddelené symbolom „->“. Prvá časť obsahuje identifikačné číslo uzla (toto je kvôli prehľadnosti zobrazené aj pri grafickej reprezentácii, a to v každom konkrétnom uzle ohraničené symbolmi „<“ a „>“) a popis parametrov uzla, druhá časť obsahuje popis vychádzajúcich hrán. Konkrétny popis vychádzajúcej hrany je zoznam, kde prvý prvok je identifikačné číslo uzla, do ktorého hrana vstupuje. Zvyšok zoznamu predstavuje parametre hrany. Podrobnejší popis syntaxe nebudem uvádzať, keďže sa v podstate jedná o interný zápis, ktorý je generovaný editorom rozkladových operátorov (pozri kapitolu 9.4).



Obr. 33 Grafická reprezentácia rozkladového operátora

Rozkladový operátor, zadefinovaný pomocou uvedeného doménového jazyka je najskôr prevedený do inštancie objektu PATG grafu. V takejto forme je potom spracovaný kompilátorom.

6.2 Lexikálna analýza

Cieľom lexikálnej analýzy je vo všeobecnosti určiť postupnosť lexikálnych jednotiek. V kompilátore bežného programovacieho jazyka ide zväčša o určenie lexikálnych jednotiek z postupnosti ASCII znakov, teda o určenie čo je číslo, čo je kľúčové slovo jazyka a podobne.

Rozkladový operátor je definovaný pomocou grafu. Za jeho lexikálne jednotky možno považovať rôzne typy uzlov a hrán. Vnútorne je však reprezentovaný takým spôsobom, že samotná reprezentácia už zo svojej podstaty obsahuje identifikované lexikálne jednotky. Takže prevádzať nejakú dodatočnú lexikálnu analýzu nie je nutné.

Implementovať lexikálnu analýzu by bolo nutné napríklad v prípade, že by bol rozkladový operátor definovaný ako bitmapový alebo vektorový obrázok. V takomto prípade by bolo nutné rozpoznať, čo je uzol, čo je hrana a aké sú ich typy. Algoritmus rozpoznania takýchto lexikálnych jednotiek by vôbec nebol jednoduchý, pravdepodobne by išlo o nejaký formu OCR rozpoznávania. Aj z tohto dôvodu bolo výhodnejšie vnútorne reprezentovať rozkladový operátor pomocou doménového jazyka, ktorý je omnoho horšie vizuálne čitateľný ako obrázok grafu. Táto zlá čitateľnosť vnútornej reprezentácie je však v konečnom dôsledku nepodstatná, nakoľko bolo veľmi jednoduché implementovať jej prevod do bitmapového obrázka.

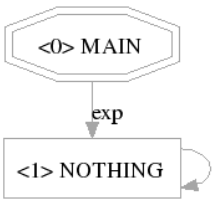
6.3 Syntaktická analýza

Úlohou syntaktickej analýzy je skontrolovať, či definícia rozkladového operátora spĺňa syntaktické pravidlá, definované v kapitole 5.4.1. Táto kontrola prebieha pomocou hľadania preddefinovaných vzorov na grafe rozkladového operátora. Tieto preddefinované vzory zodpovedajú definovaným syntaktickým pravidlám. Ich vyhľadávanie je realizované pomocou knižnice PATG.

Syntaktická kontrola rozkladového operátora prebieha nasledovne. Na začiatku sú všetky uzly aj hrany označené ako syntakticky nesprávne. Potom sú postupne v grafe operátora vyhľadávané všetky výskyty jednotlivých syntaktických konštrukcií. Každá nájdená syntaktická konštrukcia je v grafe označená za správnu. Po vyhľadaní všetkých výskytov všetkých konštrukcií ostanú tie syntaktické konštrukcie, ktoré sú chybné.

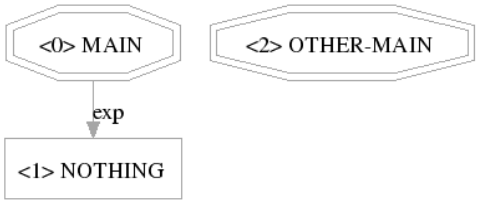
Kompletná syntaktická analýza pozostáva z nasledovných krokov:

1. **Kontrola jednoduchých cyklov**. Skontroluje sa, či sa v grafe nenachádzajú cykly v rámci jedného uzla, teda hrany, ktoré vychádzajú aj vchádzajú do toho istého uzla. Na obrázku 34 je znázornená takáto syntaktická chyba na uzle 1 (akcii `NOTHING`).



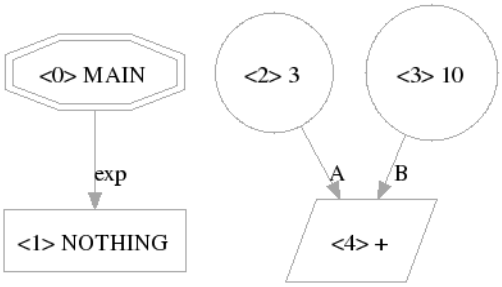
Obr. 34 Jednoduchý cyklus na uzle 1

2. **Kontrola unikátnosti referenčného uzla.** Referenčný (proxy) uzol môže byť v grafe iba jeden jediný. Na obrázku 35 je znázornená syntaktická chyba, kde uzol 0 aj uzol 2 sú referenčné.



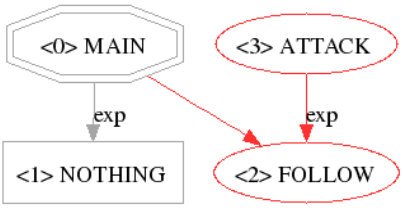
Obr. 35 Dva referenčné uzly

3. **Kontrola jednotnosti.** V grafe operátora sa nesmú nachádzať množiny uzlov, ktoré by neboli navzájom prepojené aspoň jednou hranou. Graf musí byť tvorený jedinou množinou prepojených uzlov. Na obrázku 36 je znázornená takáto syntaktická chyba (dve samostatné množiny uzlov).

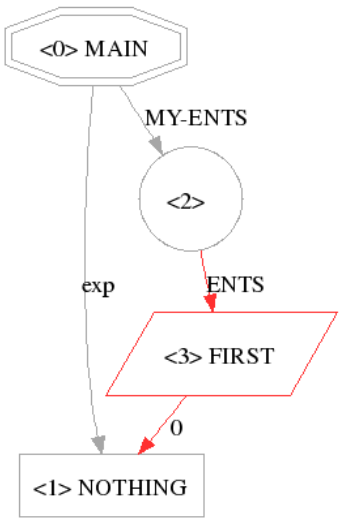


Obr. 36 Dve samostatné množiny uzlov

4. **Kontrola štruktúry uzlov.** Ide o kontrolu jednotlivých prepojení uzlov. Dovolené spôsoby prepojenia rôznych typov uzlov sú popísané v kapitole 5.4.1. Na obrázku 37 je znázornené nesprávne prepojenie úloh a na obrázku 38 nesprávne prepojenie funkcie a akcie.



Obr. 37 Chybná štruktúra uzlov typu úloha



Obr. 38 Chybná štruktúra uzlov typu akcia a funkcia

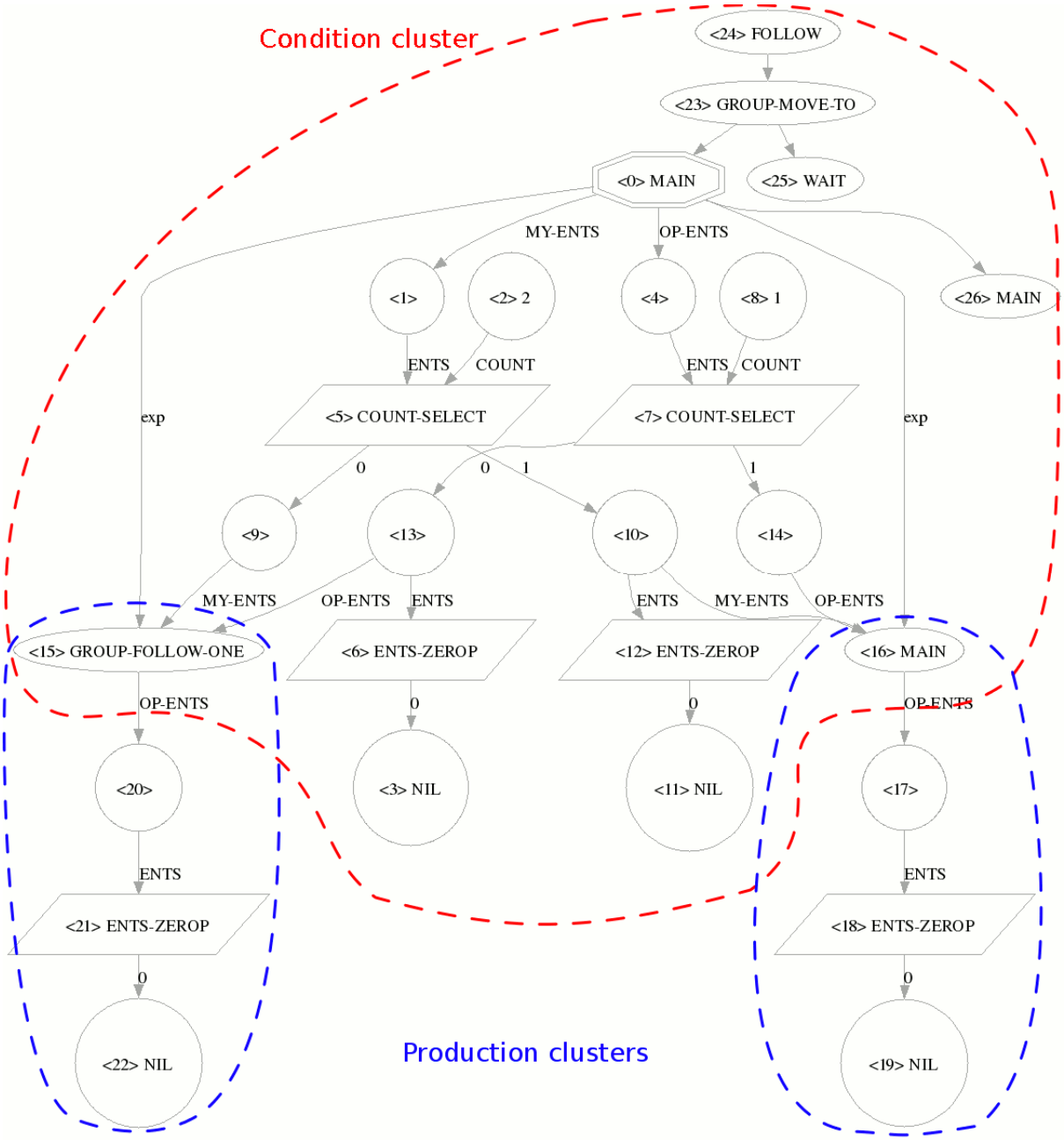
6.4 Sémantická analýza

Úlohou sémantickej analýzy je rozdeliť graf rozkladového operátora na podgrafy (ďalej nazývané ako sémantické klastre) a skontrolovať ich základnú logickú štruktúru (ucelenosť, autonómnosť a pod.). Počas sémantickej analýzy sa tiež kontroluje, či sa v grafe operátora nenachádzajú zložité cykly (cez viacero uzlov). Výstup sémantickej analýzy slúži ako základ pre fázu generovania výsledného kódu.

Sémantická analýza pozostáva z nasledovných krokov:

1. **Vytvorenie sémantických klastrov.** Rozkladový operátor je rozdelený na samostatné podgrafy – sémantické klastre. Toto rozdelenie predstavuje základ sémantickej analýzy, ako aj nasledovného generovania kódu. Sémantické klastre zodpovedajú štruktúre rozkladového operátora, popísanej v kapitole 5.4.1. Na obrázku 39 je naznačené rozdelenie na príklade zložitejšieho operátora. Operátor je najskôr rozdelený na podmienkový klaster a produkčný klaster, resp. produkčné klastre, ak je pomocou operátora produkovaných viacero nových úloh alebo akcií do stromu plánu. Podmienkový klaster je rozpoznaný podľa referenčného uzla a k nemu prislúchajúcich uzlov. Produkčné klastre sú rozpoznané rovnakým algoritmom, ale pre produkované uzly. Produkovaný uzol (ktorý môže byť len typu úloha alebo akcia) je rozpoznaný tak, že do neho vstupuje `exp` hrana z referenčného uzla. Podmienkový klaster je ďalej rozdelený na dve časti – štrukturálny a parametrický podmienkový klaster. Štrukturálny podmienkový klaster obsahuje uzly, vyjadrujúce štrukturálnu podmienku rozkladového operátora, obdobne parametrický klaster pre parametrickú podmienku.

Pri rozpoznaní a analýze štrukturálneho podmienkového klastra je vytvorená pomocná forma, ktorá zjednodušuje generovanie kódu štrukturálnej podmienky rozkladového operátora (pozri kapitolu 6.5.1).



Obr. 39 Rozdelenie operátora na sémantické klastre

2. **Kontrola autonómie sémantických klastrov.** V tomto kroku sa skontroluje, či vytvorené klastre nemajú spoločné uzly. Jediné dovolené spoločné uzly sú uzly, ktoré reprezentujú produkované úlohy alebo akcie. Tieto uzly v podstate predstavujú logické spojenie podmienkového a produkčných klastrov. Chyba autonómie sémantických klastrov môže nastať napríklad v prípade, že existuje hrana medzi uzlom z podmienkovej časti a uzlom z produkčnej časti. Prvý zo spomínaných uzlov je

súčasťou podmienky, vyjadrujúcej kedy je možné aplikovať operátor, pričom druhý zo spomínaných je súčasťou podmienky, dokedy je produkovaná úloha validná. Tieto dve kontroly podmienok sa vykonávajú úplne oddelene (pozri kapitolu 5.5) a sú teda nezlučiteľné akýmkoľvek priamym prepojením.

3. **Vytvorenie podgrafov.** Z rozpoznaných sémantických klastrov sa vytvoria podgrafy, ktoré slúžia ako vstupy pre algoritmy generujúce výsledný kód. Tieto podgrafy majú vnútornú reprezentáciu zhodnú s grafom rozkladového operátora (teda s grafom PATG knižnice). Spolu s vytvorením podgrafov je zaznamenané aj mapovanie uzlov z podgrafu do celkového grafu rozkladového operátora.

6.5 Generácia kódu

Generovanie kódu je poslednou fázou kompilácie rozkladového operátora. Výsledný vygenerovaný kód je lambda funkcia, skompilovaná Common Lisp kompilátorom. Kostra generovaného kódu je vždy rovnaká. Najskôr je vygenerovaný kód, vyjadrujúci podmienku aplikovateľnosti operátora – štrukturálnu (ak existuje) a parametrickú. Algoritmus generovania parametrickej podmienky dostane ako parametre dve formy, obsahujúce kód úspechu a kód neúspechu parametrickej podmienky, a podgraf obsahujúci parametrický podmienkový klaster operátora. Kód úspechu parametrickej podmienky obsahuje vytvorenie nového uzla (novej úlohy alebo akcie), nastavenie jeho parametrov a pripojenie do stromu plánu. Parametre nového uzla závisia od jeho konkrétneho typu. Každý však obsahuje lambda funkciu, vyjadrujúcu podmienku validity tohto uzla. Táto je vytvorená rovnakým algoritmom ako parametrická podmienka celého rozkladového operátora, avšak daný algoritmus pracuje na produkčnom klastri tohto uzla (a taktiež má iné formy pre úspech a zlyhanie). Forma pre zlyhanie v tomto prípade obsahuje kód, ktorý zmaže daný uzol zo stromu plánu (resp. celý podstrom, ktorého koreň je daný uzol).

Generátor kódu rozkladového operátora je teda tvorený dvoma hlavnými časťami:

- **Generátor kódu štrukturálnej podmienky.** Jeho úlohou je vygenerovať kód funkcie, ktorá po zavolaní vráti hodnotu pravda alebo nepravda podľa toho, či je štrukturálna podmienka na aktuálnom proxy uzle splnená alebo nie.
- **Generátor kódu parametrickej podmienky.** Jeho úlohou je vygenerovať zo zadaného podgrafu a kódu úspechu a neúspechu funkciu, ktorá po splnení parametrických podmienok vykoná kód úspechu, a pri zlyhaní parametrických podmienok vykoná kód neúspechu.

Pre pochopenie fungovania generátora kódu je nutné si uvedomiť, že v jazyku Lisp nie je rozdiel medzi dátami a programom. Na manipuláciu s programom možno použiť rovnaké „nástroje“ ako na manipuláciu s dátami. Napríklad formu

```
(+ 1 2)
```

možno chápať ako zoznam o troch prvkoch, kde prvý prvok je symbol `+`, druhý symbol `1` a tretí symbol `2`. S touto formou (dátami) možno pracovať ako s ľubovoľným zoznamom. Napríklad funkciou `first` získať jeho prvý prvok:

```
(first '(+ 1 2)) => +
```

Túto istú formu však možno chápať ako kód programu. Jeho prvý prvok je názov funkcie (v tomto prípade funkcie sčítania) a zvyšné prvky parametre. Vyhodnotenie takéhoto programu je potom podľa očakávania číslo 3:

```
(eval (+ 1 2)) => 3
```

6.5.1 Generátor kódu štrukturálnej podmienky

Generovanie kódu štrukturálnej podmienky vychádza z formy vytvorenej pri sémantickej analýze rozkladového operátora. Ide v podstate o hľadanie podstromu v strome plánu, pričom je známy jeden pevný uzol, od ktorého prebieha hľadanie. Tento uzol je referenčný proxy uzol rozkladového operátora. Vygenerovaný kód má hľadať práve požadovaný podstrom v strome plánu, a to práve z referenčného uzla. Nájsť tento podstrom znamená, že strom plánu bude obsahovať rovnako poprepájané uzly (úlohy a akcie) rovnakého typu.

Prehľadávanie stromu má už z podstaty stromu ako dátovej štruktúry rekurzívny charakter. Kvôli výslednej rýchlosti je však výhodné tento rekurzívny charakter previesť do algoritmu generovania kódu. Samotný vygenerovaný kód už neobsahuje rekurziu, ale priame iteratívne prehľadanie jednotlivých uzlov. Takto vygenerovaný kód vyhľadáva priamo požadovaný tvar podstromu. Algoritmus generovania kódu je pomerne komplikovaný (aj keď veľmi krátky) meta-program. Jeho výsledkom pre rozkladový operátor na obrázku 39 je nasledovný kód:

```
(LAMBDA (ACTUAL-AI-NODE)
  (AND
    (BIND ((ACTUAL-AI-NODE (AI-NODE-PARENT ACTUAL-AI-NODE)))
      (AND ACTUAL-AI-NODE (TYPEP ACTUAL-AI-NODE 'AI-TASK-GROUP-MOVE-TO)
        (BIND ((ACTUAL-AI-NODE (AI-NODE-PARENT ACTUAL-AI-NODE)))
          (AND ACTUAL-AI-NODE (TYPEP ACTUAL-AI-NODE 'AI-TASK-FOLLOW)))
            (ITERATE #:main-iter2290
              (FOR #:CANDIDATE2291 :IN-AI-NODE-CHILDS ACTUAL-AI-NODE)
                (WHEN (AND (TYPEP #:CANDIDATE2291 'AI-TASK-WAIT) T)
                  (RETURN-FROM #:main-iter2290 T)))))
    (ITERATE #:main-iter2293
      (FOR #:CANDIDATE2294 :IN-AI-NODE-CHILDS ACTUAL-AI-NODE)
        (WHEN (AND (TYPEP #:CANDIDATE2294 'AI-TASK-MAIN) T)
          (RETURN-FROM #:main-iter2293 T)))))
```

6.5.2 Generátor kódu parametrickej podmienky

Generovanie kódu parametrickej podmienky je založené na postupnom stotožňovaní hodnôt (uzlov typu `Value`). Najskôr je však potrebné určiť, v akom poradí sa majú jednotlivé hodnoty stotožňovať. Nie je možné napríklad stotožniť hodnotu A s výsledkom sčítania 1 a B, pokiaľ nie je známa hodnota B.

Algoritmus určenia poradia stotožňovania hodnôt pracuje na princípe zoznamu uzavretých a otvorených uzlov hodnôt. Začína sa s uzlami, ktoré môžeme stotožniť s hodnotou, ktorú už poznáme (so zadanou konštantnou hodnotou alebo s výstupom funkcie, ktorá nemá vstupné parametre). Tieto uzly sa presunú do uzavretých. Potom sa postupuje ďalej v zozname otvorených uzlov, pričom uzol presunieme do uzavretého, ak poznáme hodnotu, s akou ho máme stotožniť (všetky uzly, ktoré túto hodnotu určujú, sú v zozname uzavretých uzlov). Takýmto spôsobom zostane v zozname uzavretých uzlov požadované poradie uzlov.

Generovanie samotného kódu potom prebieha tak, že sa v stanovenom poradí prechádza po uzloch, a podľa typov uzlov a hrán medzi mini sú generované formy, zodpovedajúce programovej reprezentácií danej relácie dvoch uzlov. Ide napríklad o vytvorenie formy na zavolanie požadovanej funkcie, pričom je skontrolovaná existencia všetkých vyžadovaných parametrov a automaticky určené ich poradie z definície funkcie (niektoré parametre sú interné a neprístupné z grafu operátora). Ak uzol typu `Value` už pri vytvorení obsahuje nejakú konštantnú hodnotu X a je stotožňovaná s nejakou inou hodnotou, vygenerovaná zodpovedajúca forma obsahuje podmienku, že hodnota, ktorá má byť stotožnená s týmto uzlom musí byť ekvivalentná zadanej konštantnej hodnote X, inak parametrická podmienka zlyhá (na toto miesto sa umiestni kód zlyhania).

Tieto formy sú postupne vnárané do seba, až vytvoria výsledný kód, ktorý je nakoniec doplnený o kód (formu) úspechu.

Nasledovná funkcia predstavuje kompletne vygenerovaný kód pre rozkladový operátor na obrázku 40.

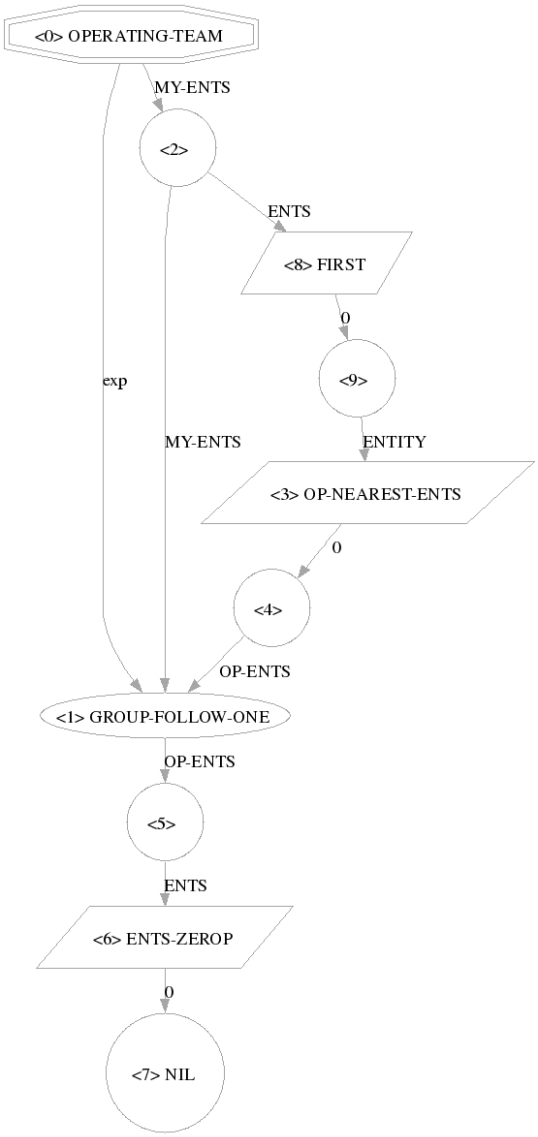
```
(LAMBDA (ACTUAL-AI-NODE MASTER MYSELF OPPONENT)
  (BLOCK OPERATING-TEAM
    (WHEN
      (BIND ((VALUE-1 (SLOT-VALUE ACTUAL-AI-NODE 'MY-FREE-ENTS)))
        (BIND ((VALUE-5 (AI-FN-FIRST VALUE-1)))
          (BIND ((VALUE-4 (AI-FN-OP-NEAREST-ENTS MASTER OPPONENT VALUE-5)))
            (MAKE-AI-TASK-GROUP-FOLLOW-ONE
                (NODES-CACHE-OF MYSELF)
                ACTUAL-AI-NODE
                (LAMBDA (ACTUAL-AI-NODE MASTER MYSELF OPPONENT)
                  (BLOCK GROUP-FOLLOW-ONE
                    (BIND ((VALUE-3 (SLOT-VALUE ACTUAL-AI-NODE 'OP-ENTS)))
                      (BIND ((VALUE-1 (AI-FN-ENTS-ZEROP VALUE-3)))
                        (UNLESS (AND (EQL VALUE-1 'NIL))
                          (UNMAKE-AI-NODE (NODES-CACHE-OF MYSELF)
                                          ACTUAL-AI-NODE)
                            (RETURN-FROM GROUP-FOLLOW-ONE NIL))
                        T))))
                VALUE-1 VALUE-4)
          T)))
    T)))
```

Tento príklad rozkladového operátora pre jednoduchosť neobsahuje žiadnu štrukturálnu podmienku. Výsledná lambda funkcia je zavolaná vždy, keď sa AIPT hráč pokúsi použiť tento operátor na rozvitie konkrétneho uzla v strome plánu. Parametre tejto funkcie majú nasledovný význam:

- `actual-ai-node` – referencia na uzol, na ktorý sa aplikuje tento operátor,
- `master` – referencia na inštanciu `Master` modulu (pozri kapitolu 3.2),
- `myself` – referencia na inštanciu tohto hráča,
- `oponent` – referencia na inštanciu oponenta.

Vo výslednom kóde možno vidieť postupnosť `BIND` foriem, ktoré reprezentujú stotožňovanie hodnôt. Produkcia nového uzla funkciou `MAKE-AI-TASK-GROUP-FOLLOW-ONE` je vykonaná ako posledná forma (úspechu). Jeden z parametrov tejto funkcie je aj lambda

funkcia, obsahujúca podmienku validity novovytvorenej úlohy. Znova je tvorená postupnosťou `BIND` foriem, pričom `VALUE-1` (v grafe rozkladového operátora na obrázku 40 ju predstavuje uzol 7) je podmienená na hodnotu `NIL`. Neúspešná forma v tomto prípade obsahuje funkcie zmazania podstromu uzla a uzla samotného funkciou `UNMAKE-AI-NODE`.



Obr. 40 Príklad rozkladového operátora bez štrukturálnej podmienky

7 Experimenty

Funkčnosť AIPT agenta bola overená na vytvorení dvoch rôznych množinách rozkladových operátorov. Nakoľko rozkladové operátory určujú kompletné správanie sa AIPT agenta, možno tieto množiny chápať ako dva rôzne agenty. Experimenty prebiehali ako súboj týchto dvoch AIPT agentov. Agenty boli testované aj v hre proti ľudskému hráčovi. Cieľom experimentov bolo dokázať funkčnosť návrhu AIPT agenta, ako aj použiteľnosť grafového jazyka na zápis rozkladových operátorov.

Z výsledkov experimentov je vo všeobecnosti možné povedať, že samotný priebeh hry je závislý nielen na samotných hráčoch, ale aj na hernom prostredí ako takom. Dynamika hry a rôznorodosť prostredia (hernej arény, druhov jednotiek a druhov zbraní) výrazne vplýva aj na dôležitosť taktického správania sa hráča (AIPT agenta). Jednoduché prostredie znižuje dôležitosť taktického správania, v extrémnych prípadoch do takej miery, že hráč je úspešný aj s úplne jednoduchým správaním sa bez náznaku akejkoľvek komplexnejšej taktiky alebo stratégie.

V experimentoch bolo síce použité relatívne jednoduché prostredie, avšak dostatočne rôznorodé, že umožnilo sledovať dôležitosť taktického správania sa. Vzhľadom na danú komplexnosť vytvoreného prostredia boli navrhnuté aj spomínané dve množiny rozkladových operátorov. V čase písania tejto diplomovej práce bolo implementovaných niekoľko ďalších faktorov zvyšujúcich rôznorodosť prostredia (nových jednotiek a zbraní), avšak z časových dôvodov neboli vykonané ďalšie experimenty (bolo by totiž nutné prispôsobiť aj agentov na nové faktory).

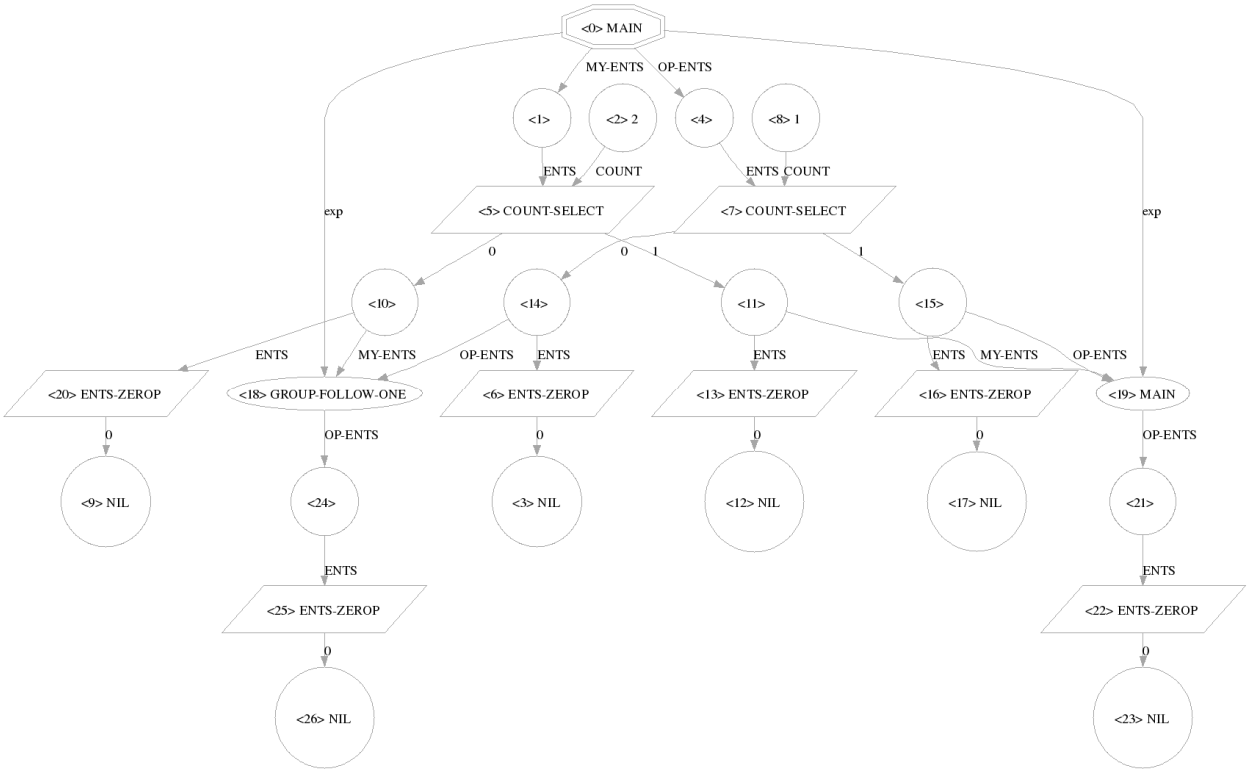
7.1 Súboj počítačom riadených hráčov

Súboj prebiehal medzi dvoma hráčmi s vývojovými označeniami `basic-ai-player` a `caesar-ai-player`, pričom obaja mali rovnaký počet a druhy jednotiek a zbraní. Obsahovali rozdielne množiny rozkladových operátorov. Nasledovné podkapitoly popisujú správanie sa hráčov na základe ich rozkladových operátorov.

7.1.1 AIPT agent `basic-ai-player`

Tento agent vznikol pôvodne z dôvodu oživovania a testovania kompilátora a základných mechanizmov AIPT agenta (tvorba plánu a pod.). Jeho správanie je založené na jednoduchom

princípe, že rozdeľuje svoje entity na dvojice, ktoré postupne posiela na jednu súperovu entitu, pričom nezohľadňuje žiadne ďalšie faktory (silu alebo vzdialenosť od súperovej entity). Ak je súperova entita zničená, posiela konkrétnu dvojicu na nasledovnú entitu (poradie súperových entít je dané ich identifikačným číslom). Ak má agent viacej entít ako súper, posiela všetky zvyšné na zostávajúcu súperovu entitu. Toto správanie je vyjadrené v rozkladových operátoroch na obrázkoch 41 a 42. Operátor na obrázku 41 má vyššiu prioritu ako operátor na obrázku 42. Tieto dva operátory možno chápať ako klasický model *first/rest* rekurzie, pričom operátor na obrázku 42 je v takomto chápaní ukončovacia podmienka rekurzie. Operátor na obrázku 41 logicky zodpovedá operátoru na obrázku 39, ktorý bol v kapitole 6.4 použitý na vysvetlenie sémantickej analýzy rozkladového operátora.



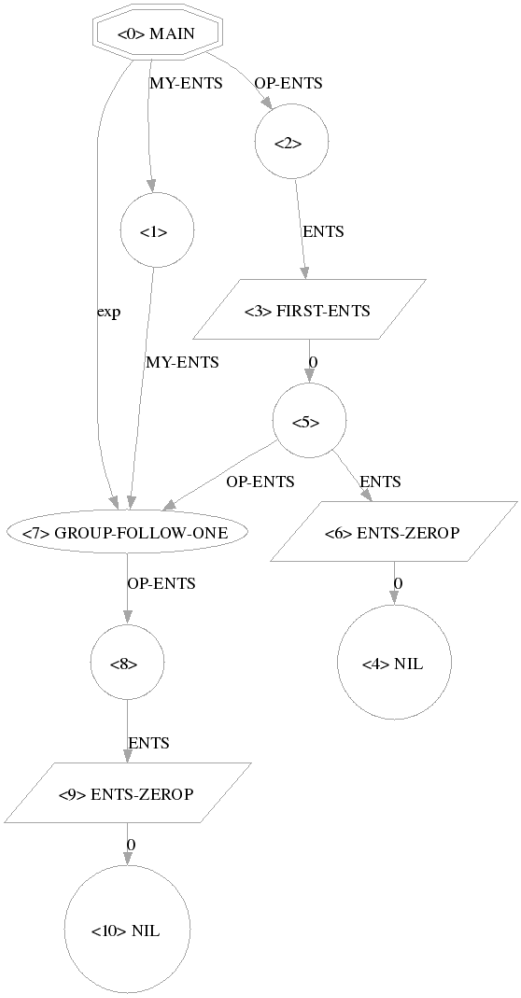
Obr. 41 Operátor rozkladu úlohy `Main` na úlohu `Group-Follow-One` a opäť `Main`

Funkcia `Couns-Select` vyberie z množiny entít `Ents`, ktorú dostane ako vstupný parameter, prvých `Count` entít (ďalší vstupný parameter). Vráti množinu entít, obsahujúcu vybrané entity, prípadne množinu bez entít, ak vstupná množina neobsahuje požadovaný počet entít (overované funkciou `Ents-Zerop`).

Úloha `Group-Follow-One` je ďalej rozkladaná rôznymi operátormi, ktoré v podstate riešia priblíženie entity agenta na dostrel k vybranej súperovej entite, útok na túto entitu a prípadné

opätovné priblíženie sa. Všetky rozkladové operátory agenta možno zobraziť funkciou `show-player-operators`:

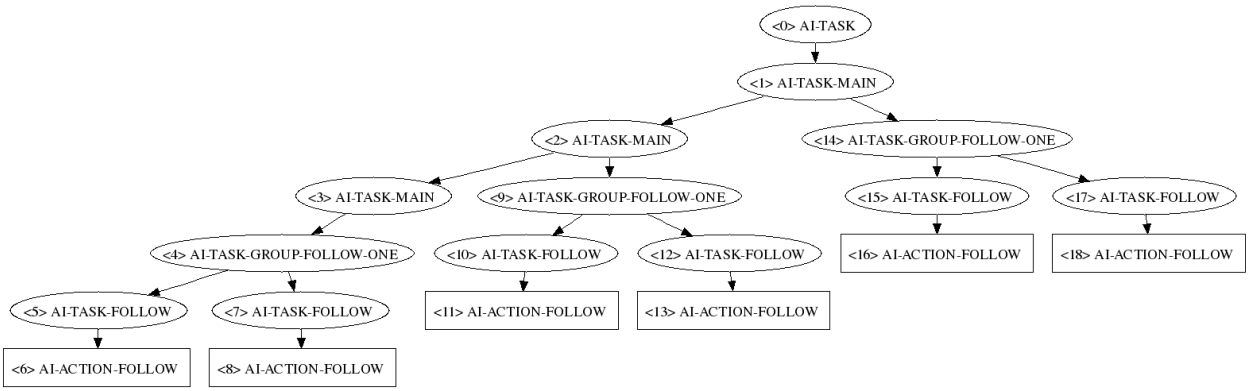
```
TACTIX> (show-player-operators 'basic-ai-player)
```



Obr. 42 Rozkladový operátor úlohy `Main` na úlohu `Group-Follow-One`

Úspešnosť tohto AIPT agenta bola pomerne vysoká, vzhľadom na jednoduchosť jeho konania. V experimentoch však nikdy nevyhral nad komplexnejším AIPT agentom `caesar-ai-player`. Toto je v podstate pozitívny a očakávaný výsledok, dokazujúci správnosť navrhnutého prostredia hry (pre splnenie požiadavky na nutnosť taktického správania sa hráča). Avšak výsledok boja agentov bol vždy veľmi tesný (a nie rovnaký, pozri kapitolu 7.1.3). Predpokladám však, že pri vyššej komplexnosti prostredia by bol aj výsledok boja výraznejší.

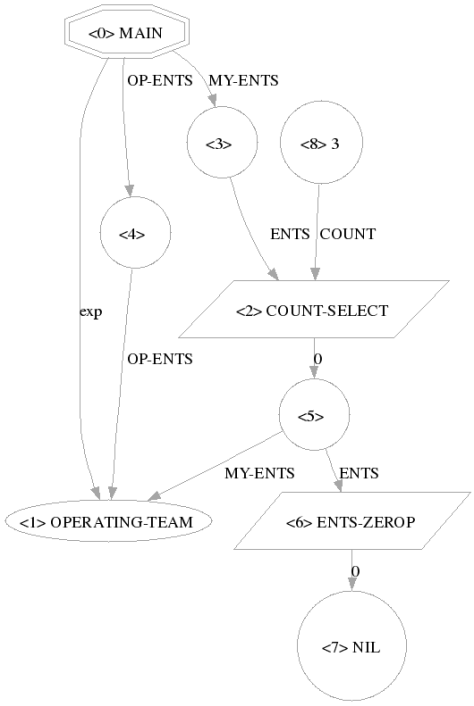
Na obrázku 43 je zobrazený strom plánu tesne (približne sekundu) po spustení hry. Oba agenty ovládali armádu tvorenú šiestimi entitami. Na obrázky je vidieť rozdelenie entít na skupiny po aplikovaní rozkladových operátorov na obrázkoch 41 a 42.



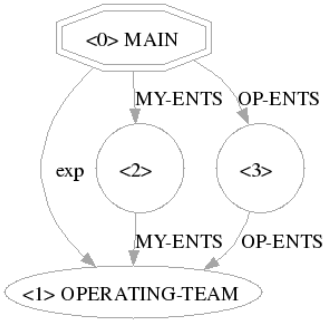
Obr. 43 Strom plánu AIPT agenta `basic-ai-player`

7.1.2 AIPT agent `caesar-ai-player`

Agent pracuje so svojimi entitami na podobnom princípe ako `basic-ai-player`. Na začiatku si rozdelí entity na malé tímy, ktoré majú svojho veliteľa (spôsob *rozdeľ a panuj*). Toto rozdelenie je vykonané pomocou rozkladových operátorov na obrázkoch 44 a 45. Tieto dva operátory možno chápať ako iteráciu nad množinou entít, pridelených k úlohe `Main`. Operátor na obrázku 44 je aplikovaný dokedy je možné vytvoriť skupinu entít s požadovanou veľkosťou (v tomto prípade tri). Operátor 45 možno chápať ako ukončenie iterácie (aplikuje sa, pokiaľ nie je možné aplikovať operátor na obrázku 44). Tvorba plánu pre tieto samostatné skupiny je komplexnejšia a jej výsledkom je taktickejšie správanie sa agenta v porovnaní s predchádzajúcim.



Obr. 44 Rozkladový operátor úlohy `Main` na úlohu `Operating-Team`

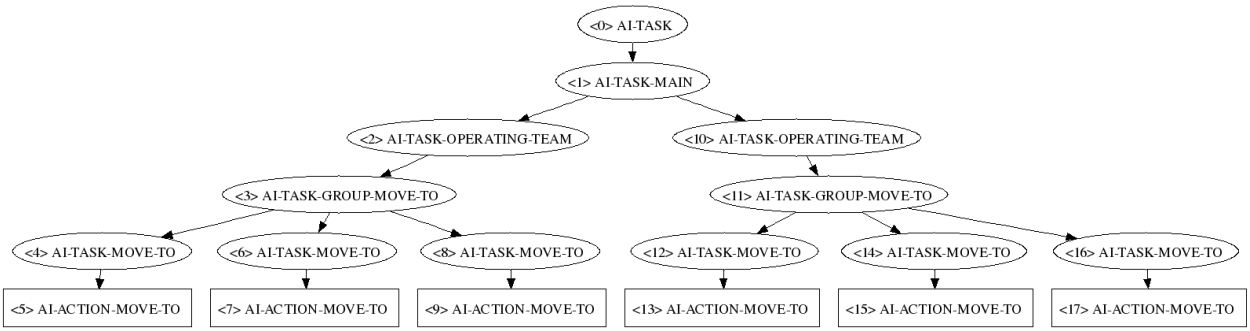


Obr. 45 Rozkladový operátor úlohy `Main` na úlohu `Operating-Team` pre zvyšok entít

Samotné správanie sa agenta určujú operátory rozkladajúce úlohu `Operating-Team`. Tieto rozkladové operátory (popísané podľa poradia ich priority) vytvárajú nasledovné podplány:

- **Rozptyl skupín.** Ak je konkrétny operačný tím príliš „nakope“ s iným, je od neho poslaný ďalej, aby sa zbytočne neblokovali a neboli jednoduchým terčom pre súpera.
- **Opatrný útok.** Skupine sa vyberie najbližší súper, na ktorého zaútočí (prenasleduje ho a ak je v dostrele tak strieľa na neho). Ak však ohrozenie veliteľa prekročí určitú hodnotu, je táto úloha zrušená. Ohrozenie je jeden zo senzorov `Master` modulu. Je vypočítané ako celková potencionálna sila útoku súperových entít na danú bunku, na ktorej sa entita nachádza (v tomto prípade veliteľ skupiny).
- **Ústup do bezpečia.** Ak ohrozenie na veliteľa skupiny prekročí určitú hranicu, skupina ustúpi od súperov do bezpečnej vzdialenosti.
- **Slepý útok.** Ak skupina nemá kam ustúpiť, teda je zahnaná do rohu, útočí na najbližšiu entitu.

Operátory vytvárajúce a rozkladajúce popísané úlohy možno zobraziť funkciou `show-player-operators`. Strom plánu vytvorený rozkladovými operátormi na obrázkoch 44 a 45 je zobrazený na obrázku 46 (ide o strom plánu približne po sekunde trvania hry, teda v podstate na začiatku boja).



Obr. 46 Strom plánu AIPT agenta `caesar-ai-player`

7.1.3 Nedeterministickosť súbojov

Počas experimentov so vzájomným súbojom dvoch AIPT agentov sa ukázalo, že žiadne dva súboje neprebiehali úplne rovnako, a približne už po troch sekundách boli pozorované rozdiely v pohybe a umiestnení entít (tento časový údaj treba chápať ako relatívny, je totiž plne závislý od nastavenej dynamiky hry).

Princíp fungovania AIPT agenta je plne deterministický. Domnievam sa, že nedeterministickosť bola do hry vnesená počas implementácie modulov `Master`, `Arena` a `Move Manager`, ktoré sa starajú o základné princípy fungovania jadra hry, a to hlavne o pohyb entít v aréne. Aréna je v podstate štvorcová mriežka, v ktorej sa pohybujú entity (kapitola 4). Počas implementácie sa však ukázalo, že entity, ktoré sa pohybujú iba do štyroch alebo ôsmych smerov vyzerajú veľmi umelo. Preto bol vytvorený relatívne zložitý systém pohybu entít v mriežkovej aréne pod ľubovoľným uhlom. Tento systém je veľmi závislý na časových intervaloch obnovovania polohy entít. Takže akýkoľvek rozdiel v postupnosti týchto intervalov má už po malom čase za následok malé rozdiely v polohách entít. Tieto rozdiely sú postupne znásobované, čo v konečnom dôsledku vedie k rôznym vzájomným súbojom tých istých súperiacich agentov v rovnakej aréne a s rovnakými entitami.

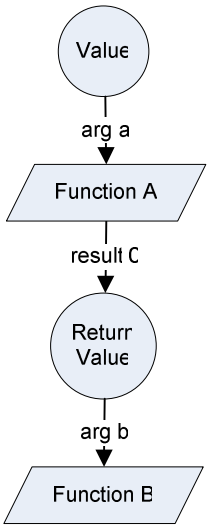
Aj keď faktor nedeterministickosti nebol v pôvodnom návrhu nijako uvažovaný, jeho objavenie sa považujem za kladný jav, ktorý zvýšil zaujímavosť a istým spôsobom aj vierohodnosť súbojov počítačom riadených hráčov. Možno tak s vyššou pravdepodobnosťou povedať, že víťazstvá konkrétneho počítačom riadeného hráča sa zakladajú na jeho lepšom ovládaní vlastných entít viacej ako na štartovacom rozložení jeho jednotiek, prípadne na iných konštantne daných štartovacích faktoroch.

7.2 Použiteľnosť grafového jazyka a možnosti rozšírenia

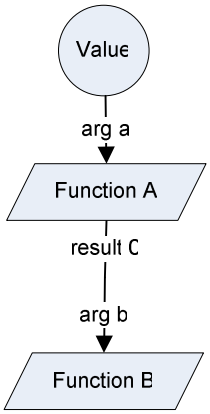
Použiteľnosť navrhnutého grafového jazyka bola predvedená v predchádzajúcej kapitole. Na rýchlu orientáciu v grafovom vyjadrení operátora je však nutná určitá prax, ako aj na vytváranie kompletných množín rozkladových operátorov tak, aby vyjadrovali požadované správanie. Myslím si však, že navrhnutý grafový jazyk splnil očakávania a výrazne zjednodušil navrhovanie rozkladových operátorov, ktoré by inak museli byť písané priamo v zdrojovom kóde. Celkovú prehľadnosť množiny rozkladových operátorov by zvýšil špecializovaný editor, ktorý by umožňoval jednoducho vytvárať a modifikovať rozkladové operátory, zobrazoval by ich logickú štruktúru a prepojenia a pod. Základ takéhoto editora bol implementovaný (kapitola 9.4), avšak slúži len na vytváranie a modifikáciu konkrétneho rozkladového operátora a neumožňuje prehľadne pracovať s kompletnou množinou rozkladových operátorov AIPT agenta. Taktiež je ovládaný výhradne pomocou textových príkazov.

Možnosti rozšírenia grafového jazyka spočívajú predovšetkým v zjednodušení niektorých syntaktických konštrukcií, resp. v rozšírení jazyka o nové. Asi najviditeľnejším zdrojom „nadbytočnosti“ uzlov je vidieť pri nútenom používaní uzlov `Value` pre akúkoľvek manipuláciu s parametrami (hodnotami) v grafe. Navrhnutá syntax je správna a jazykovo korektná, avšak by mohla byť v niektorých prípadoch schovaná a zjednodušená. Graf by sa zapísal jednoduchšie a potom by prebehla fáza rozloženia grafu do korektnej syntaxe grafového kompilátora (teda niečo ako grafové makro).

Na obrázku 47 je zobrazený príklad často sa opakujúcej konštrukcie v operátoroch, tvorený dvoma funkciami, pričom výstup jednej z nich je vstupom druhej. Ak by sa vytvoril nový druh hrany, ktorá by mala dva parametre (vstupný a výstupný) bolo by možné príklad zjednodušiť tak, ako je zobrazené na obrázku 48. Všeobecne možno povedať, že ak hodnotu potrebujeme použiť iba na jednom mieste, je to možné dosiahnuť bez vytvárania uzla `Value` pomocou dvojparametrovej hrany (myslím, že analógia s každým klasickým programovacím jazykom je úplne zrejmá). V čase písania tejto práce však takéto rozšírenie nebolo implementované.



Obr. 47 Príklad častej syntaktickej konštrukcie



Obr. 48 Zjednodušenie syntaktickej konštrukcie

7.3 Adaptívny AIPT agent

Jedným z cieľov diplomovej práce bolo vniesť do počítačom riadeného hráča schopnosť adaptívnosti (kapitola 1.3). Navrhnutý model AIPT agenta však spĺňa iba požiadavku adaptívnosti počas súboja samotného, teda že reaguje na správanie sa súpera a snaží sa modifikáciou plánu naučenými prostriedkami (vopred zadefinovanými rozkladovými operátormi) adaptovať plán na aktuálne vzniknutú situáciu. Adaptívnosť vychádzajúca z nejakej formy strojového učenia však nie je v navrhnutom modeli prítomná.

V čase písania tejto diplomovej práce však existoval iba hrubý návrh, ako vniesť do AIPT agenta faktor adaptívnosti, resp. strojového učenia sa. Nakoľko však nebol implementovaný, neboli vykonané ani žiadne experimenty na overenie jeho funkčnosti.

Návrh strojového učenia sa vychádzal z možnosti meniť prioritu rozkladových operátorov, a tým aj meniť spôsob tvorby plánu. Takýmto spôsobom by si mohol AIPT agent vytvoriť upravené množiny rozkladových operátorov, ktoré by vyjadrovali adaptovanie sa na konkrétneho súpera. S nastavovateľnou prioritou operátorov by bolo spojených niekoľko problémov, ktoré by bolo treba vyriešiť:

- **Závislosť návrhu operátorov od vzájomného poradia aplikovania.** V niektorých prípadoch sú rozkladové operátory navrhnuté tak, že implicitne predpokladajú, že obdobný rozkladový operátor (rozkladajúci rovnakú úlohu) s vyššou prioritou zlyhal a teda neplatí podmienka, ktorá viedla k jeho zlyhaniu. Táto závislosť by sa dala

odstrániť vhodnejším návrhom rozkladových operátorov, ktoré by s nemenným poradím nepočítali, ako aj využitím štrukturálnej podmienky.

- **Vyhodnotenie úspechu aplikácie operátora.** Je nutné navrhnúť metódu, pomocou ktorej sa určí, či aplikácia daného operátora viedla k úspechu alebo nie, a na základe tejto informácie (ktorá nemusí nevyhnutne nadobúdať iba dve hodnoty – áno a nie) upraviť prioritu rozkladového operátora.

8 Záver

Po preštudovaní zadania som navrhol modelovú oblasť, v ktorej sa pokúsim zrealizovať experiment s podporou riadenia a rozhodovania. Ako modelovú oblasť som si vybral strategickú hru v reálnom čase. Výber tejto oblasti som vysvetlil v úvode tohto dokumentu.

V prvom semestri som sa venoval hlavne návrhu a implementácii základného modelu systému hry, teda jadru servera a klientovi. Podarilo sa mi vyriešiť problémy so synchronizáciou servera a klienta. Pri hre v reálnom čase je totiž bezpodmienečne nutné, aby zobrazovaný stav hry na klientovi čo najpresnejšie zodpovedal skutočnému stavu hry (ktorý je celý počítaný na serveri).

V druhom semestri som dokončil implementáciu základných modulov servera, hlavne `Master` modulu a k nemu patriacich modulov `Pathfinder` a `Move Manager`. Veľkú časť práce som venoval práve modulu `Pathfinder`, resp. hľadaniu optimálnej cesty v relatívne komplikovanom teréne arény. Implementoval som viacero prototypov A* algoritmu. Nakoniec sa ako najviac výhodné ukázalo použitie binárnej haldy a priame prehľadávanie v statickej mriežke arény (žiadna alokácia nových uzlov a podobne). Toto riešenie však aj tak nebolo dostačujúce pre väčšie množstvá entít (náhodne roztrúsené po aréne), ak ich hráč pošle všetky naraz do jedného miesta. Dochádzalo k nepríjemným oneskoreniam na serveri, čo sa na klientovi prejavovalo ako neskoré reakcie entít a celkové „sekanie“ hry. Preto som navrhol a implementoval riešenie s predpočítavaním ciest a ich použitím pri hľadaní optimálnej cesty. Toto riešenie sa ukázalo ako dostačujúce.

Podstatnú časť práce v druhom semestri som venoval návrhu počítačom riadeného hráča. Rozhodol som sa ho navrhnúť ako autonómneho agenta, ktorý riadi ostatné entity. Entita sama osebe nebola chápaná ako agent. Toto rozhodnutie som zdôvodnil v kapitole 5.1. Preštudoval som základné prístupy a modely pre navrhovanie autonómnych agentov. Na základe tejto štúdie som navrhol vlastný model fungovania agenta, nazvaný AIPT agent. Model sa zakladá na inkrementálnom tvorení a modifikovaní plánu. Jadrom modelu sú grafovo vyjadrené operátory na rozkladanie úloh, a strom úloh ako vyjadrenie rozkladov.

V treťom semestri som dokončil implementáciu základných modulov servera. Pri implementácií počítačom riadených hráčov (ako aj modulu `Client Support Inteligence`) bolo nutné urobiť niekoľko implementačných zmien v `Master` module, ktoré súviseli s rozhraním modulu na prijímanie príkazov od hráčov. Ďalej bolo nutné implementovať rôzne senzory na sledovanie stavu arény (ohrozenosť bunky konkrétnym hráčom a pod.). Poslednou podstatnou úpravou `Master` modulu bolo pridanie možnosti

pozastaviť a znova spustiť hru. Táto funkcionalita slúžila ako pomocný nástroj pri ladení AIPT agentov.

Hlavnou náplňou práce v treťom semestri bola implementácia kompilátora rozkladových operátorov (kapitola 6), vytvorenie testovacích AIPT agentov a vykonanie experimentov na overenie funkčnosti celého modelu hry TACTIX. Taktiež som relatívne veľa času venoval vylepšovaniu vizuálnej stránky klienta a jeho ovládania. Podarilo sa mi navrhnúť a otestovať funkčnosť dvoch AIPT agentov. Tieto agenty obsahovali odlišné množiny rozkladových operátorov, čo sa prejavilo aj v ich odlišnom správaní sa. Výsledky experimentov (vzájomných súbojov) týchto agentov sú zhrnuté v kapitole 7. Vo všeobecnosti možno povedať, že vykonané experimenty potvrdili funkčnosť navrhnutého modelu hry a AIPT agenta.

Dokončil som implementáciu rôznych podporných nástrojov pre pohodlnejšiu prácu s prostredím hry. Medzi tieto nástroje patria hlavne:

- editor arény (kapitola 9.3),
- editor rozkladových operátorov (kapitola 9.4) a
- funkcie na zobrazovanie stavu hráča (stromu plánu a rozkladových operátorov).

Po overení funkčnosti grafového jazyka a kompilátora som navrhol možnosti jeho rozšírenia, ktoré by spočívali v zjednodušení niektorých, často sa vyskytujúcich, syntaktických konštrukciách. Toto možné rozšírenie som popísal v kapitole 7.2.

Z časových dôvodov sa mi nepodarilo podrobne navrhnúť a implementovať schopnosť AIPT agenta adaptovať sa na protihráča, ktorá by vychádzala z nejakej formy strojového učenia sa. Agent dynamicky reaguje na zmenu stavu hry a adaptuje svoj plán pomocou vopred daných rozkladových operátorov. Neobsahuje však žiadnu formu prispôsobovania a zefektívňovania tejto adaptácie plánu na konkrétneho protihráča. Kapitola 7.3 obsahuje základný hrubý návrh jednej z možností, ako vniesť faktor adaptívnosti do AIPT agenta.

9 Technická dokumentácia

Táto kapitola obsahuje stručný popis implementácie servera a klienta a krátke príklady zdrojových kódov. Ďalej obsahuje základný popis implementovanej vývojovej verzie editora arény, editora rozkladových operátorov a používateľskú príručku k celému systému TACTIX.

9.1 Implementácia modulov servera

Všetky moduly servera sú implementované v jazyku Common Lisp (štandardizovaný v ANSI X3.226-1994), s využitím základných vlastností CLOS (Common Lisp Object System) [17]. Program je kompilovaný pomocou SBCL (Steel Bank Common Lisp, http://www.sbcl.org).

Jednoduchý príklad použitia metódy `find-path` z modulu `Pathfinder` je uvedený v nasledujúcom príklade:

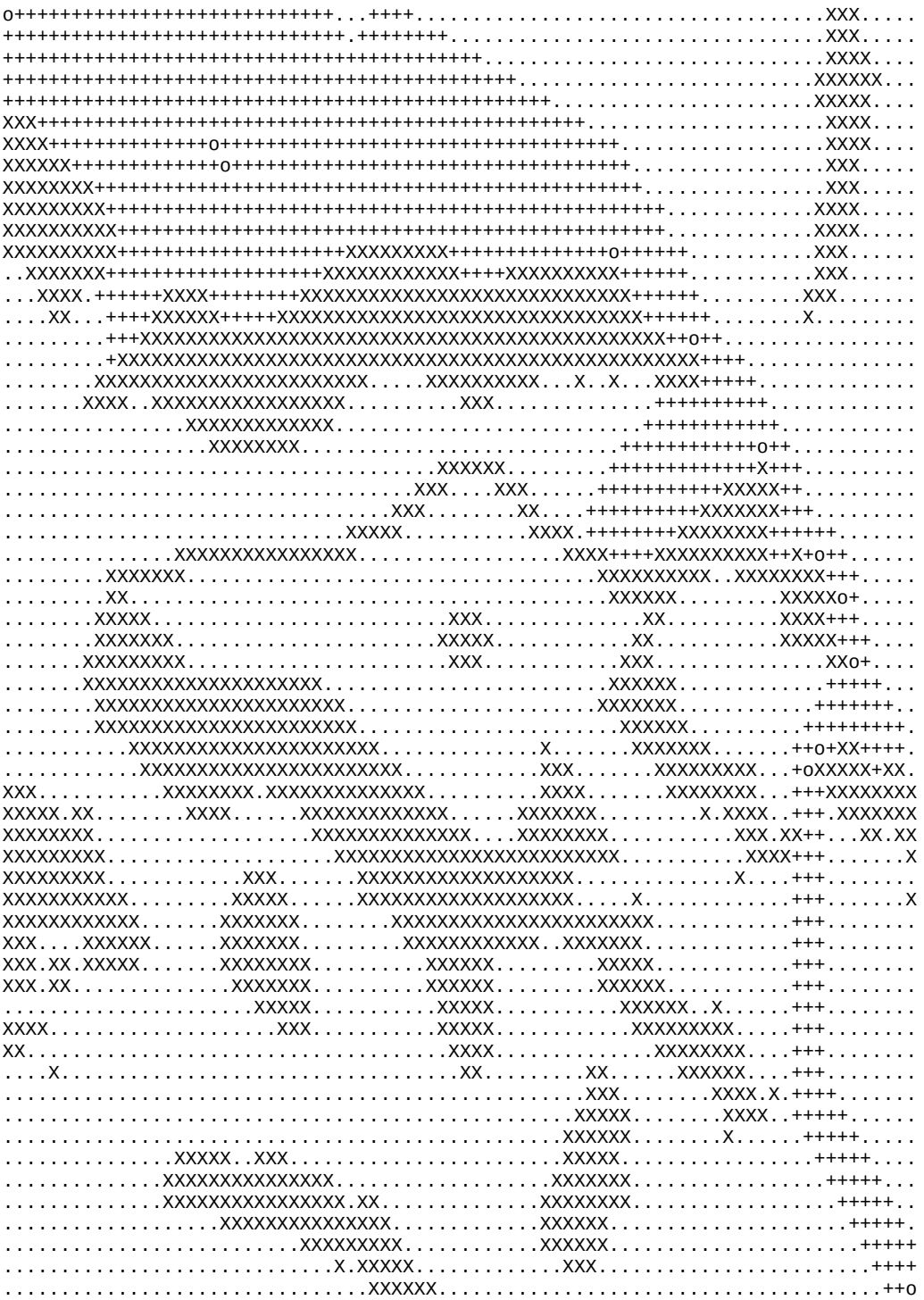
```
TACTIX> (let* ((g (make-instance 'game :level "battle-1"))
               (pf (pathfinder-of g))
               (pb (make-array 100 :element-type 'fixnum
                                   :fill-pointer 0)))
           (time
              (find-path pf 0 0 79 59 pb
                         :distance-fn #'distance-heuristics
                         :cost-fn #'terrain-cost-of
                         :path-border-width 0
                         :normalize t))
           (show-grid pf :pf-index 1))

INIT    load-arena: dimensions [80 60]
INIT    prepare-arena: 2 players and 0 entities
INIT    pathfinder: grid dimensions [80 60]
INIT    dummy player initialized
INIT    dummy player initialized

Evaluation took:
  0.01 seconds of real time
  0.008 seconds of user run time
  0.0 seconds of system run time
  0 calls to %EVAL
  0 page faults and
  77,928 bytes consed.
```

Najskôr sa vytvorí inštancia celej hry. Tá vytvorí ostatné triedy servera a načíta všetky potrebné informácie o aréne `battle-1` (mriežku buniek a predpočítané cesty). Metóda `find-path`, ktorá pomocou implementovaného A* algoritmu nájde cestu (bez použitia predpočítanej cesty) z horného ľavého rohu arény do dolného pravého, je vložená do funkcie `time`. Táto funkcia vypíše informácie o spotrebovanom výpočtovom čase (`Evaluation took`). Pomocou tejto funkcie boli porovnávané jednotlivé experimenty s prototypmi A* algoritmu. Výsledná cesta je uložená vo vektore `pb`.

Na obrázku 49 je zobrazený výstup funkcie `show-grid`, ktorý bol kvôli prehľadnosti vynechaný z predchádzajúceho príkladu. Zobrazuje nájdenú cestu v aréne. Znak `'X'` predstavuje nepriechodnú bunku, znak `'.'` predstavuje priechodnú bunku, znak `'o'` bod cesty a znak `'+'` bunku, ktorá bola pri hľadaní cesty „navštívená“.



Obr. 49 Zobrazenie nájdenej cesty funkciou `show-grid`

9.2 Implementácia klienta

Klient je implementovaný ako Java Applet s využitím knižnice GTGE (Golden T Game Engine, http://www.goldenstudios.or.id/products/GTGE/). Klient slúži výhradne ako „konzola“, teda zobrazuje stav hry, resp. arény v reálnom čase. Ďalej umožňuje pozastaviť

hru a znovu ju nechať pokračovať (čo je veľmi výhodné pri ladení počítačom riadeného hráča). Ak je jeden z hráčov človek, tomuto umožní ovládať jeho jednotky. Celá logika hry sa teda nachádza na serveri.

Kód klienta je však napísaný v jazyku LINJ (Lisp Is Not Java, http://www.evaluator.pt/). LINJ je jazyk napísaný v Common Lispe, ktorý sa snaží poskytnúť čo najviac vyjadrovacích prostriedkov Common Lispu a zároveň byť prekompilovateľný do *„human-readable“* Java kódu. Tento Java kód je potom skompilovaný klasickým `javac` kompilátorom do Java class súboru.

Nasledujúci ilustračný príklad obsahuje kód napísaný v LINJ:

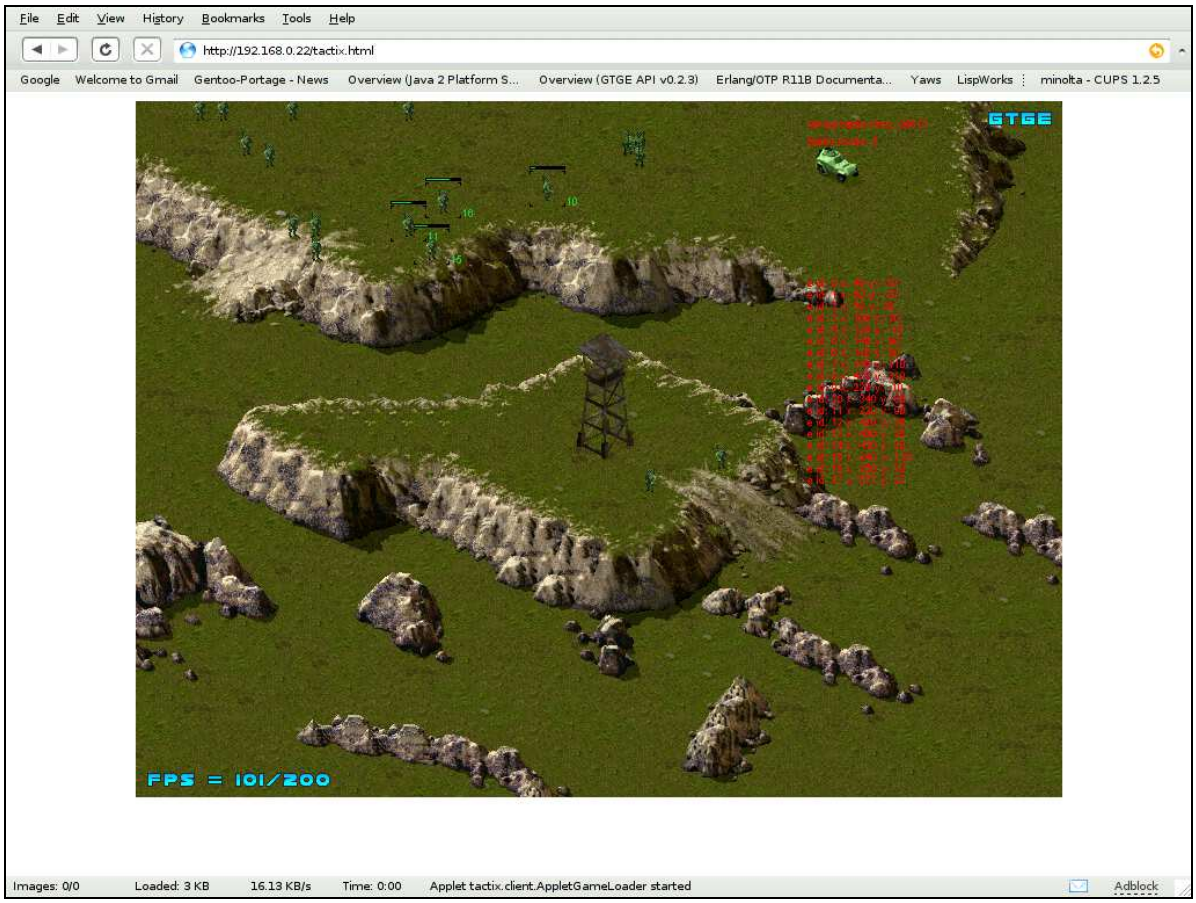
```
(defmacro handle-mouse-buttons (mouse-buttons handle-test handler)
  `(progn
       ,@(loop :for button :in mouse-buttons
               :collect `(when (,@handle-test ,button)
                               (,@handler ,button
                                          (get-mouse-x bs-input)
                                          (get-mouse-y bs-input))))))

(defmethod update ()
  (handle-mouse-buttons (+mouse-button-left+ +mouse-button-middle+
                         +mouse-button-right+)
       (is-mouse-pressed bs-input)
       (mouse-press-handler))
  (handle-mouse-buttons (+mouse-button-left+ +mouse-button-middle+
                         +mouse-button-right+)
       (is-mouse-released bs-input)
       (mouse-release-handler)))
```

Tento kód po prekompilovaní do Javy vyzerá nasledovne:

```
public void update() {
        if (bsInput.isMousePressed(MOUSE_BUTTON_LEFT)) {
            mousePressHandler(MOUSE_BUTTON_LEFT, bsInput.getMouseX(),
            bsInput.getMouseY());
        }
        if (bsInput.isMousePressed(MOUSE_BUTTON_MIDDLE)) {
            mousePressHandler(MOUSE_BUTTON_MIDDLE, bsInput.getMouseX(),
            bsInput.getMouseY());
        }
        if (bsInput.isMousePressed(MOUSE_BUTTON_RIGHT)) {
            mousePressHandler(MOUSE_BUTTON_RIGHT, bsInput.getMouseX(),
            bsInput.getMouseY());
        }
        if (bsInput.isMouseReleased(MOUSE_BUTTON_LEFT)) {
            mouseReleaseHandler(MOUSE_BUTTON_LEFT, bsInput.getMouseX(),
            bsInput.getMouseY());
        }
        if (bsInput.isMouseReleased(MOUSE_BUTTON_MIDDLE)) {
            mouseReleaseHandler(MOUSE_BUTTON_MIDDLE, bsInput.getMouseX(),
            bsInput.getMouseY());
        }
        if (bsInput.isMouseReleased(MOUSE_BUTTON_RIGHT)) {
            mouseReleaseHandler(MOUSE_BUTTON_RIGHT, bsInput.getMouseX(),
            bsInput.getMouseY());
        }
    }
```

Na obrázku 50 je zobrazený klient ako Java Applet v prehliadači Mozilla Firefox.



Obr. 50 Java Applet klienta v prehliadači Mozilla Firefox

9.3 Editor arény

Editor arény slúži na špecifikovanie vlastností mriežky arény, resp. na špecifikovanie vlastností jednotlivých buniek. Ako grafický podklad arény načíta GIF obrázok rovnakého mena ako aréna (napr. `battle-1.gif`). Tento podklad musí následne spracovať užívateľ, tzn. označiť všetky nepriechodné bunky, cenu cesty priechodných buniek a *„pseudo-z“* (hodnota, ktorá určuje výšku bunky).

Editor arény je súčasťou servera. Je rovnako implementovaný v jazyku Common Lisp s využitím knižnice LTK (The Lisp Toolkit, http://www.peter-herth.de/ltk/). Umožňuje exportovať definované vlastnosti buniek, resp. celú mriežku do súboru. Tento súbor je nutný pre spustenie hry TACTIX, resp. zodpovedajúcej úrovni (levelu). Editor ďalej umožňuje do toho istého súboru exportovať predpočítané cesty. Zobrazuje rozmiestnenie vlajok pre zvolenú granularitu. Na obrázku 51 je okno editora, na ktorom vidieť označené nepriechodné bunky a rozmiestnenie vlajok, v tomto prípade pre granularitu 7.

Editor možno spustiť nasledovným spôsobom:

```
TACTIX> (level-editor)
LEVEL-EDITOR    level /home/miso/tactix/server/resources/battle-1.level
                imported
INIT            load-arena: dimensions [80 60]
INIT            prepare-arena: 2 players and 0 entities
INIT            pathfinder: grid dimensions [80 60]
INIT            dummy player initialized
INIT            dummy player initialized
LEVEL-EDITOR    created 142 flags for border-width 0
```

Obr. 51 Hlavné okno editora arény

9.4 Editor rozkladových operátorov

Editor rozkladových operátorov slúži ako jednoduchý nástroj na vytváranie a modifikáciu rozkladových operátorov. Ide v podstate o jednu funkciu, spustiteľnú z REPLu. Komunikuje s užívateľom pomocou nasledovných textových príkazov:

- Príkaz: q. Ukončenie práce s editorom. Editor vráti vnútornú formu predstavujúcu vytvorený rozkladový editor (pozri kapitolu 6.1).

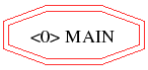
- Príkaz: `an <popis uzla>`. Vytvorenie nového uzla podľa popisu (napr. `an val 4` – pridá do grafu uzol typu `Value` s konštantnou hodnotou 4).
- Príkaz: `ae <id začiatočného uzla> <id koncového uzla> <popis hrany>`. Vytvorí hranu zo začiatočného do koncového uzla, s parametrami podľa popisu (napr. `ae 0 1 slot my-ents` – vytvorí hranu z uzla 0 do uzla 1, predstavujúcu stotožnenie slotu `my-ents`).
- Príkaz: `dn <id uzla>`. Zmaže daný uzol zo stromu, spolu s prislúchajúcimi vstupnými a výstupnými hranami.
- Príkaz: `de <id začiatočného uzla> <id koncového uzla>`. Zmaže danú hranu.
- Príkaz: `cn <id uzla> <nový popis uzla>`. Zmení typ daného uzla podľa nového popisu.
- Príkaz: `ce <id začiatočného uzla> <id koncového uzla> <nový popis hrany>`. Zmení danú hranu podľa nového popisu.

Editor funguje tak, že inkrementálne vytvára nové rozkladové operátory podľa postupnosti príkazov zadávaných užívateľom. Po každom novom príkaze sa pokúsi vytvoriť a skompilovať nový operátor, ako aj vygenerovať jeho grafickú reprezentáciu (syntaktické chyby sú farebne zvýraznené). Chybové výstupy z kompilácie sú zobrazované užívateľovi. Grafická reprezentácia, teda vygenerovaný obrázok, je otvorená v zadefinovanom grafickom prehliadači. Je výhodné, ak tento prehliadač podporuje automatické načítanie obrázka po jeho modifikácii. Výber prehliadača obrázkov je na užívateľovi. Taktiež je dobré, ak obsahuje funkcie priblíženia a podobne.

Nasledujúci príklad obsahuje ukážku vytvorenia jednoduchého rozkladového operátora:

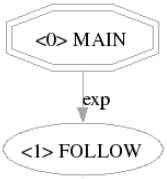
1. **Spustenie editora.** Editor sa spúšťa funkciou `ai-operator-editor`, pričom ako vstupný parameter sa zadáva začiatočná vnútorná definícia rozkladového operátora. Ak chce užívateľ vytvoriť nový rozkladový operátor, začiatočná forma je prázdny zoznam. Po spustení editor vypíše výzvu na zadanie vstupu „`#>` “. Prázdny rozkladový operátor nie je validný operátor.
2. **Editovanie operátora.** Editovanie prebieha zadávaním príkazov. Nasledujúce obrázky a príkazy ilustrujú postupné vytvorenie jednoduchého operátora.

```
#> an ref main
invalid nodes: (0)
```

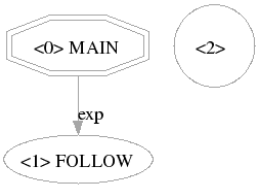



```
#> an task follow
failed validating using CHECK-SEPARATE-
COMPONENTS
```

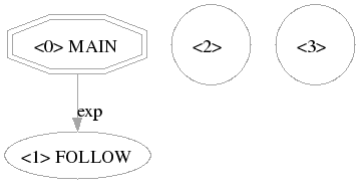

```
#> ae 0 1 exp
required slots (MY-ENTS OP-ENTS)
are missing in FOLLOW
```



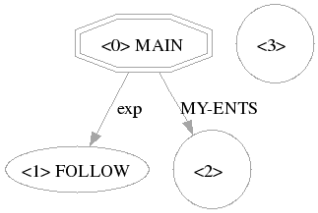

```
#> an val
failed validating using CHECK-
SEPARATE-COMPONENTS
```



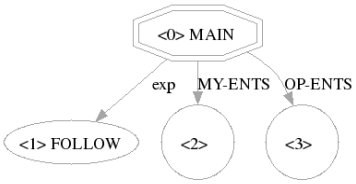

```
#> an val
failed validating using CHECK-
SEPARATE-COMPONENTS
```



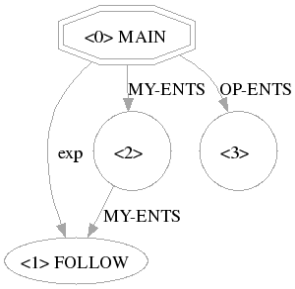

```
#> ae 0 2 slot my-ents
failed validating using CHECK-SEPARATE-
COMPONENTS
```



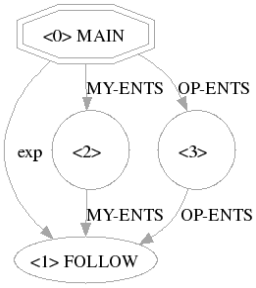

```
#> ae 0 3 slot op-ents
required slots (MY-ENTS OP-ENTS)
are missing in FOLLOW
```




```
#> ae 2 1 slot my-ents
required slots (OP-ENTS) are missing in
FOLLOW
```




```
#> ae 3 1 slot op-ents
```



3. **Ukončenie editora.** Po ukončení práce editor vráti výslednú formu, zodpovedajúcu vnútornej reprezentácii rozkladového operátora.

```
#> q

((0 REF MAIN -> (1 EXP) (2 SLOT MY-ENTS) (3 SLOT OP-ENTS)) (1 TASK FOLLOW)
(2 VAL -> (1 SLOT MY-ENTS)) (3 VAL -> (1 SLOT OP-ENTS)))
```

9.5 Inštalácia a spustenie systému TACTIX

Inštalácia systému TACTIX pozostáva z dvoch častí, inštalácie a nastavenia klienta a z inštalácie a nastavenia servera. Klient môže fungovať ako samostatná Java aplikácia alebo ako Java Applet v prehliadači.

9.5.1 Systémové požiadavky

Pre spustenie klienta je nutné mať nainštalované:

- Java™ SE Runtime Environment verzie 1.5 a vyššie,
- GTGE knižnicu,
- LINJ rozšírenie jazyka,
- pre spustenie klienta v Java Applet móde príslušný modul v prehliadači (závisí od konkréteho prehliadača a implementácie Javy).

Pre spustenie servera je nutné mať nainštalované:

- Steel Bank Common Lisp verzie 1.0.12 a vyššie,
- príslušné knižnice popísané v `tactix.asd` súbore.

Všetky knižnice potrebné pre kompiláciu a spustenie servera je nutné nainštalovať do systému podľa inštalačného návodu príslušnej knižnice (prípadne podľa zaužívaného postupu pre daný operačný systém a jeho distribúciu). Všetky použité Common Lisp knižnice sú inštalovateľné pomocou ASDF (Another System Definition Facility, http://www.cliki.net/ASDF-Install), ako aj samotný systém TACTIX.

Zoznam potrebných knižníc:

- metabang-bind (http://common-lisp.net/project/metabang-bind/),
- iterate (http://common-lisp.net/project/iterate/),
- cl-utilities (http://common-lisp.net/project/cl-utilities/),
- cl-store (http://common-lisp.net/project/cl-store/),
- Tcl/Tk (http://www.tcl.tk/),
- LTK (http://www.peter-herth.de/ltk/),
- PATG (http://common-lisp.net/project/patg/) a
- Graphviz (http://www.graphviz.org/).

9.5.2 Inštalácia a spustenie servera

Server sa kompiluje a nahráva do Lisp stroja nasledovnou funkciou:

```
CL-USER> (asdf:oos 'asdf:load-op :tactix)
```

Server je zabalený do balíka menom TACTIX. REPL možno prepnúť do tohto balíka pomocou funkcie `in-package`:

```
CL-USER> (in-package :tactix)
#<PACKAGE "TACTIX">
```

Základná konfigurácia servera (podobne ako aj klienta) sa nachádza v súbore `config`. Nahratie aktuálnej konfigurácie po jej zmene je možné vykonať funkciou `load-config`:

```
TACTIX> (load-config)
CONFIG              server-address: #(127 0 0 1)
CONFIG              server-port: 9123
CONFIG              roundtrip-requests-count: 50
CONFIG              tactix-directory: /home/miso/tactix
```

Server možno spustiť napríklad nasledovným spôsobom:

```
TACTIX> (start-server-for-clients :clients-count 2 :worker #'game-worker)
```

Voľba štartovacej funkcie, ako aj :worker funkcia závisí od experimentu, resp. typu hry, ktorý chceme spustiť (dvaja človekom ovládaní hráči, jeden hráč AIPT alebo obaja hráči AIPT). Funkcia :worker tiež obsahuje definíciu armády každého hráča.

Celý server možno skompilovať do jedného spustiteľného súboru taxtix-server pomocou nasledovnej funkcie (server v tomto príklade možno spustiť len pre jedného klienta):

```
(defun build-tactix-server ()
  (sb-ext:save-lisp-and-die (concatenate 'string (sb-posix:getenv "HOME")
                                         "/tactix/tactix-server")
            :toplevel (lambda ()
                 (start-server-for-one-client :worker #'game-worker)
                 (sb-ext:quit))
            :executable t))
```

9.5.3 Inštalácia a spustenie klienta

Keďže klient je naprogramovaný v Jave, inštalácia pozostáva v podstate z inštalácie a konfigurácie virtuálneho stroja Javy. Spustenie klienta možno vykonať príkazom:

```
$ java tactix.client.Main
```

Základná konfigurácia sa nachádza v súbore config.

9.6 Vytvorenie AIPT hráča

Kompletné vytvorenie nového AIPT hráča pozostáva z piatich krokov (nie všetky sú nevyhnutné, pokiaľ sa použijú časti z už existujúceho hráča):

- **Definícia chýbajúcich funkcií.** Ide o definíciu funkcií, ktoré sa vyskytujú v operátoroch ako uzly typu Function. Tento krok je nutný iba ak požadovaná funkcia medzi aktuálne zadefinovanými chýba. V aktuálnej verzii servera sú implementované všetky funkcie, použité v rozkladových operátoroch pri

experimentoch. Nasledujúci príklad ilustruje vytvorenie funkcie `game-time`, ktorá slúži na zistenie aktuálneho času hry:

```
(define-ai-function game-time (master) :inline
  (time-of master))
```

- **Definícia funkcie akcií.** Vykonanie každej akcie musí byť zadefinované zodpovedajúcou funkciou (rovnaké meno funkcie ako meno akcie, resp. uzla typu `Action`). Funkcia akcie je definovaná pomocou anaforického makra `define-ai-action-function`. Príklad definície funkcie pre akciu `Attack`:

```
(define-ai-action-function attack
  (bind ((my-entity (ai-action-attack-my-entity action)))
    (unless (is-aiming-p my-entity)
      (bind ((op-entity (ai-action-attack-op-entity action)))
        (attack-command (master-of myself) myself (x-of op-entity) (y-of
op-entity) my-entity +primary-weapon+)))))
```

- **Definícia uzlov.** Ide o definíciu použitých uzlov typu `Task` a `Action`. Každá úloha a akcia automaticky obsahuje základné parametre (ako množinu hráčových entít a množinu súperových entít a pod...), prípadne je možné dodefinovať ďalšie dodatočné parametre (sloty), s ktorými sa dá potom manipulovať v rozkladových operátoroch. Nasledujúci príklad obsahuje definíciu uzla typu `Action` menom `wait`. Definuje aj dodatočný slot `expiration-time`:

```
(define-ai-node action wait
  (expiration-time 0 :type fixnum))
```

- **Definícia operátorov.** Rozkladový operátor, vytvorený napríklad pomocou editora rozkladových operátorov (kapitola 9.4), sa zadefinuje pomocou makra `define-ai-operator`. Toto makro obsahuje kód, ktorý skompiluje danú definíciu rozkladového operátora, a zaradí ho do množiny rozkladových operátorov. Rozkladový operátor je identifikovaný dvojicou mien, pričom prvé meno musí byť názov uzla ktorý rozkladá a druhý unikátne meno pre rozklad tohto konkrétneho uzla. Táto dvojica mien potom jednoznačne identifikuje rozkladový operátor. Príklad jednoduchej definície rozkladového operátora, ktorý rozkladá úlohu `main`:

```
(define-ai-operator (main task-operating-team/rest)
    ((0 REF MAIN -> (1 EXP) (2 SLOT MY-ENTS) (3 SLOT OP-ENTS))
     (1 TASK OPERATING-TEAM)
     (2 VAL -> (1 SLOT MY-ENTS)) (3 VAL -> (1 SLOT OP-ENTS))))
```

- **Definícia hráča.** Hráč je definovaný pomocou makra `define-ai-player`. Ako prvý parameter je názov hráča. Potom nasleduje špecifikácia množiny rozkladových operátorov, ktorá je podmnožinou všetkých zadefinovaných operátorov. Špecifikácia množiny je tvorená zoznamom úloh, ku ktorým sú priradené mená operátorov, ktoré tieto úlohy rozkladajú. Poradie mien rozkladových operátorov určuje aj ich prioritu. Príklad definície AIPT agenta `caesar-ai-player`:

```
(define-ai-player caesar-ai-player
    ((main (task-operating-team/full
            task-operating-team/rest))
     (operating-team (task-group-move-to/diffusion
                      task-group-move-to/follow-and-attack-carefully
                      task-group-move-to/run-to-safety
                      task-group-move-to/follow-and-attack-blind))
     (group-move-to (task-move-to))
     (move-to (task-wait/on-destination
               action-move-to
               action-wait/failed-move))
     (wait (action-attack
            action-wait))
     (group-follow-one (task-follow))
     (follow (action-attack
              action-follow
              action-wait))))
```

10 Literatúra

[1] NÁVRAT, P. et al.: Umelá inteligencia. 1.vyd. Bratislava: STU, 2002. 405 s. ISBN 80-227-1645-6.

[2] RABIN, S. et al.: AI Game Programming Wisdom. 1. vyd. Hingham: Charles River Media, 2002. 672 s. ISBN 1-58450-077-8.

[3] KVASNIČKA, V., POSPÍCHAL, J., TIŇO, P.: Evolučné algoritmy. 1. vyd. Bratislava: STU, 2000. 223 s. ISBN 80-227-1377-5.

[4] WOODCOCK, S. M.: Games Making Interesting Use of Artificial Intelligence Techniques. http://www.gameai.com/, 2006 (11. 12. 2006).

[5] KOENIG, S., LIKHACHEV, M., FURCY, D.: Lifelong Planning A*. Computer Science Department, USC, Los Angeles. 2005.

[6] MANLEY, K.: Pathfinding: From A* to LPA. University of Minnesota. 2003.

[7] STUART, J. R., NORVING, P.: Artificial Intelligence: A Modern Approach. 1. vyd. New Jersey: Prentice Hall, 1995. 946 s. ISBN 0-13-103805-2.

[8] WOOLDRIDGE, M., JENNINGS, N. R.: Intelligent Agents: Theory and Practice. The Knowledge Engineering Review 10(2): 115-152. 1995.

[9] NAREYEK, A. et al.: Excalibur: Adaptive Constraint-Based Agents in Artificial Environments. http://www.ai-center.com/projects/excalibur/, 2007 (21.4 2007).

[10] WEIZENBAUM, J.: Eliza – A Computer Program For the Study of Natural Language Communication Between Man and Machine. Communications of the ACM 9(1): 36-45. 1966.

[11] AGRE, P., and CHAPMAN, D.: Pengi: An Implementation of a Theory of Activity. In Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), 268-272. 1987.

[12] GRAND, S., CLIFF, D., MALHOTRA, A.: Creatures: Artificial Life Autonomous Software Agents for Home Entertainment. In Proceedings of the First International Conference on Autonomous Agents (Agents '97), 22-29. 1997.

[13] STERN, A., FRANK, A., RESNER, B.: Virtual Petz: A Hybrid Approach to Creating Autonomous Lifelike Dogz and Catz. In Proceedings of the Second International Conference on Autonomous Agents (AGENTS98), 334-335. 1998.

[14] BONASSO, R. P. et al.: Experiences with an Architecture for Intelligent, Reactive Agents. Journal of Experimental and Theoretical Artificial Intelligence 9(1). 1997.

[15] PELL, B. et al.: A Remote Agent Prototype for Spacecraft Autonomy. In Proceedings of the SPIE Conference on Optical Science, Engineering, and Instrumentation. 1996.

[16] GRAHAM, P.: On Lisp: Advanced Techniques for Common Lisp. 1. vyd. Prentice Hall, 1993. 432 s. ISBN 0-13-370875-6.

[17] SEIBEL, P.: Practical Common Lisp. 1. vyd. New York: Apress, 2005. 528 s. ISBN 1-59059-239-5.

Príloha: Elektronické médium

Obsah priloženého elektronického nosiča:

contents.txt	- obsah elektronického nosiča
anotacia.txt	- anotácia v slovenskom jazyku
annotation.txt	- anotácia v anglickom jazyku
/sources/client	- zdrojové texty klienta
/sources/server	- zdrojové texty servera
/papers/diplom.doc	- tento dokument (formát MS Word)
/papers/diplom.pdf	- tento dokument (formát PDF)
/papers/pictures.vsd	- vytvorené obrázky, diagramy a grafy použité v tomto dokumente (formát MS Visio)
/pictures/operators	- obrázky generované knižnicou Graphviz použité v tomto dokumente
/pictures/game	- ukážky z tvorby grafiky pre hru