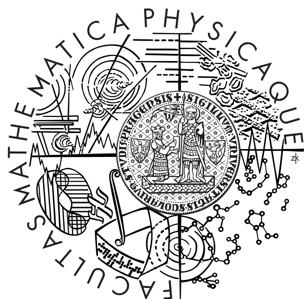
Univerzita Karlova v Prahe

Matematicko-fyzikálna fakulta

BAKALÁRSKA PRÁCA



Tomáš Mikula

Hierarchické reaktívne plánovanie s prechodmi

Kabinet software a výuky informatiky

Vedúci bakalárskej práce: Mgr. Cyril Brom

Študijný program: Informatika, obecná informatika

2006

Chcem sa poďakovať vedúcemu práce Mgr. Cyrilovi Bromovi za jeho konzultácie, cenné pripomienky a za to, že si pri množstve projektov, ktoré vedie, vždy našiel čas venovať sa každému individuálne.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej zverejňovaním.

V Prahe dňa 10. augusta 2006 Tomáš Mikula

Obsah

1 Úvod **6**
1.1 Štruktúra práce . . . 6
1.2 Terminologická dohoda . . . 6

2 Problém výberu akcie **7**
2.1 Reaktívnosť vs. plánovanie . . . 7

3 Hierarchické reaktívne plánovanie **9**
3.1 Reaktívny plán . . . 9
3.2 Správanie pomocou reaktívneho plánu . . . 10
3.3 Hierarchia reaktívnych plánov . . . 11
3.4 Lezenie po stromoch . . . 13
3.5 Nezodpovedané otázky . . . 13
3.6 Nedostatky HRP . . . 16

4 Prechodné správania **17**
4.1 Pokus o implementáciu čisto v HRP . . . 18
4.2 Rozšírenie HRP o prechodné správania . . . 18

5 Odloženie správania **20**
5.1 Prečo nie dynamická zmena priority . . . 20
5.2 Rozšírenie HRP o odloženie správania . . . 21
5.3 Formálne rozšírenie reaktívneho plánu . . . 23

6 Implementácia v Projekte ENTi **24**
6.1 Jazyk E . . . 24
6.2 Rozvrhovač . . . 26
6.3 Ukážka správania . . . 28

7 Príbuzné práce **29**
7.1 POSH reaktívne plány . . . 29
7.2 Expressivator . . . 30

8 Záver **32**

A Prechodné správania a odloženie správania v Projekte ENTi **33**
A.1 Kto sú Enti? . . . 33
A.2 Prechodné správania (transition behaviors) . . . 33
A.3 Odložené spustenie správania . . . 34
A.4 Niečo málo o jazyku E . . . 35
A.5 Výber akcií u Entov . . . 37
A.6 Rozvrhovač . . . 38
A.7 Rozšírenie o prechodné správania a odložené spustenie . . . 39
A.8 Podrobnosti implementácie rozvrhovača . . . 40
A.9 Ukážka správania . . . 43
A.10 Zásahy do interpretu jazyka E . . . 45

B Inštalácia a spustenie Projektu Enti **46**
B.1 Čo budeme potrebovať 46
B.2 Postup inštalácie 46
B.3 Spustenie 47

Literatúra **48**

Názov práce: *Hierarchické reaktívne plánovanie s prechodmi*
Autor: *Tomáš Mikula*
Katedra: *Kabinet software a výuky informatiky*
Vedúci bakalárskej práce: *Mgr. Cyril Brom*
e-mail vedúceho: *brom@ksvi.mff.cuni.cz*

Abstrakt: *Hierarchické reaktívne plánovanie (HRP) je obľúbená metóda pre riadenie správania umelých bytostí. Výhodou HRP je, že sa v ňom i komplexné správania zapisujú pomerne jednoducho. HRP má však v určitých situáciach problémy s biologickou vierohodnosťou. Toto je zčasti spôsobené tým, že sa v HRP obtiažne zapisujú tzv. prechodné správania a odloženie správania. Prechodné správania sú krátke činnosti, ktoré by simulované bytosti mali vykonávať medzi dvoma hlavnými správaniami a zabezpečiť hladký prechod medzi nimi. Odloženie správania je vhodné v prípade, keď bežiace správanie bude čoskoro končiť a preto by nemalo byť prerušené. V tejto práci rozšírime model HRP o prechodné správania a odloženie správania a popíšeme implementovaný prototyp.*

Kľúčové slová: *hierarchické reaktívne plánovanie, prechodné správania, odloženie správania, umelá inteligencia*

Title: *Hierarchical reactive planning with transions*
Author: *Tomáš Mikula*
Department: *Department of Software and Computer Science Education*
Supervisor: *Mgr. Cyril Brom*
Supervisor's e-mail address: *brom@ksvi.mff.cuni.cz*

Abstract: *Hierarchical reactive planning (HRP) is a popular method for controlling virtual beings' behaviour. The advantage of HRP is that complex behaviours can be described relatively easily. However, in particular situations problems with biological plausibility arise. This is partially caused by the fact that so called transition behaviours and postponement of behaviour are hard to express in HRP. Transition behaviours are short actions that the simulated beings should engage in between two main behaviours in order to ensure a smooth transition between them. Postponement of behaviour is desirable in case the running behaviour is approaching the end and therefore should not be interrupted. In the present work we incorporate transition behaviours and postponement of behaviour into the model of HRP and describe the implemented prototype.*

Keywords: *hierarchical reactive planning, transition behaviours, postponement of behaviour, artificial intelligence*

1. Úvod

Hierarchické reaktívne plánovanie (HRP) je populárnou metódou riadenia správania umelých bytostí. Obľúbenosť plynie najmä z toho, že pri zachovaní jednoduchosti návrhu možno v HRP dosiahnuť plnenie komplexných úloh v dynamicky sa meniacom prostredí. HRP však má svoje nedostatky, ktoré sa týkajú hlavne nedostatočnej biologickej vierohodnosti v určitých situáciach. Príčinou je aj to, že sa v HRP nedajú jednoducho vyjadriť tzv. prechodné správania a odloženie činnosti. Prechodné správania by mali byť spúšťané medzi dvoma hlavnými správaniami a postarať sa o hladký prechod medzi nimi. Ak, napríklad, človek počas varenia ide zdvihnúť telefón, prechodným správaním môže byť stlmenie sporáka. Toto sa však v HRP zapisuje obtiažne. Odloženie správania má za úlohu potlačiť spustenie nového správania, ak je to práve prebiehajúce už pred koncom. Ak, napríklad, záhradník navrhnutý pomocou HRP zalieva záhony, zostáva mu zaliať už len jediný záhon a dostane hlad, je zalievanie prerušené a odíde sa najesť. Odloženie správania spočíva v tom, aby keď už zalievanie nebude trvať dlho, jedenie počkalo, kým zalievanie neskončí.

1.1 Štruktúra práce

V 2. kapitole je jemný úvod do problému výberu akcie, v ktorom si pripravíme pôdu pre uvedenie HRP v kapitole 3. Potom sa budeme zaoberať analýzou a riešením predložených problémov. V kapitole 4 HRP rozšírime o prechodné správania. V 5. kapitole sa venujeme odloženiu správania. V 6. kapitole potom popíšeme implementáciu navrhnutých rozšírení v Projkte ENTi.

1.2 Terminologická dohoda

Agentom sa väčšinou rozumie program, ktorý je situovaný v nejakom prostredí a je schopný v ňom konať *autonómne*, v záujme plnenia svojich cieľov.

Umelou bytosťou rozumieme agenta, ktorý „žije“ v prostredí podobnom skutočnému (alebo kľudne i fantastickému) svetu, má simulované telo podliehajúce zákonom prostredia a snaží sa napodobniť svoj vzor z reálneho (fantastického) sveta.

My budeme tieto pojmy niekedy zamieňať. Agentom budeme väčšinou myslieť umelú bytosť, zvlášť keď v niektorých úvahách rozdiel nebude dôležitý. Keď budeme hovoriť o simulovanej ľudskej bytosti, budemie niekedy používať slovo *postavička*.

2. Problém výberu akcie

Od umelých bytostí obyčajne požadujeme, aby sa správali rozumne[1], pôsobili autenticky (tj. konali podobne ako ich vzory z reálneho sveta) a dosahovali dobré výsledky pri plnení svojich cieľov. Ciele, ktoré sa snaží bytosť dosiahnuť, jej určí autor. Zvyčajne je medzi nimi okrem iných uspokojenie základných životných potrieb, napríklad príjem potravy, ak je v danom svete simulovaný hlad.

Systém, v ktorom správanie bytostí modelujeme, poskytne každej bytosti množinu jednoduchých (atomických, primitívnych) fyzických akcií, ktoré môže vo virtuálnom vykonávať. Vykonanie fyzickej akcie sa prejaví navonok a spôsobí zmenu prostredia (i keď niekedy len nepatrnú). Úlohou autora bytosti je, aby z daných atomických akcií „poskladal“ správanie. Musí navrhnúť mechanizmus, ktorý bude vyberať vhodné akcie, aby výsledné konanie vyhovovalo požiadavkám z predchádzajúceho odstavca.

2.1 Reaktívnosť vs. plánovanie

Aby bytosť pôsobila vierohodne, musí reagovať na zmeny svojho okolia, a to dostatočne skoro—musí byť dostatočne *reaktívna*.

Uvažujme plánujúcu bytosť. Predstavme si, že sa naša bytosť rozhodla nakupovať. Túto činnosť si dopredu naplánuje a potom už len vykonáva naplánované kroky. Cestou do obchodu stretne kamaráta, ktorý sa jej prihovorí. Ona sa však teraz venuje nakupovaniu a kamaráta si nevšíma. Až donakupuje, vytvorí si plán, čo bude robiť ďalej. Potom možno na kamaráta zareaguje, ak tam stále bude.

Asi nikomu takéto správanie nepripadá vierohodné. Bytosť nie je dostatočne reaktívna. Pretože je prostredie nedeterministické, stav prostredia po vykonaní akcie sa nedá s určitosťou predvídať a nastávajú nepredvídané udalosti. Ako správanie vylepšiť? Je možné zakomponovať do plánu reakciu na každú možnú udalosť, ktorá by prípadne mohla nastať? Možné to pravdepodobne je, ale neprehľadnosť popisu každej činnosti by prudko stúpala s každou ďalšou možnou situáciou, ktorú by bolo treba riešiť. Navyše, po každej takejto nečakanej udalosti by bolo nutné preplánovanie, ktoré je výpočetne náročné.

Okrem reakcie na zmeny okolia potrebuje bytosť reagovať na svoje vnútorné zmeny (napr. hlad), ktoré sú, z pohľadu autora správania, väčšinou tiež nedeterministické.

Rovnako, zámery a ich dôležitosť sa môžu časom meniť a je treba sa včas vzdať utlmujúceho sa zámeru.

Mali by sme však myšlienku plánovania opustiť úplne?

Uvažujme reaktívnu bytosť. Tá na základe svojho vnútorného stavu a stavu vnímaného prostredia vyberie vždy len jednu akciu, neplánuje na dlhšie obdobie. Ako takúto bytosť donútiť vykonať nejakú netriviálnu činnosť,

[1] Keď modelujeme napríklad opicu, slovom „rozumne“ myslíme niečo iné, ako keď modelujeme človeka.

ktorá je časovo náročná a na dokončenie vyžaduje množstvo primitívnych akcií? Potrebujeme, aby zvolená fyzická akcia napomohla splneniu aktuálneho cieľa a zároveň spôsobila taký stav prostredia a agentovho vnútra, že v ďalšom kroku bude opäť vybraná akcia smerujúca k dokončeniu činnosti.

Potrebujeme metódu, ktorá sa za normálnych okolností správa ako plán, ale v prípade potreby dokáže pružne reagovať na nečakané udalosti. Jednu takúto metódu predstavíme v nasledujúcej kapitole.

3. Hierarchické reaktívne plánovanie

Hierarchické reaktívne plánovanie (HRP) je prístup k výberu akcie, ktorý je dostatočne reaktívny a zároveň sa ním dobre modelujú komplexné správania.

Pri jeho použití bytosť vždy „plánuje“ iba najbližšiu fyzickú akciu, avšak týmto uvažovaním o ďalšej akcii vznikne taký vnútorný stav, že po úspešnom vykonaní zvolenej akcie bude mať bytosť tendenciu vybrať ďalšiu akciu vedúcu k splneniu aktuálneho cieľa. Ak však nastane nepredvídaná udalosť, je bytosť schopná reagovať okamžite. A to všetko pri zachovaní jednoduchosti návrhu.

Z predchádzajúcej kapitoly máme dojem, že slovné spojenie „reaktívne plánovanie“ je tak trochu protichodné. HRP v skutočnosti žiadne plánovanie nie je, aspoň nie v klasickom zmysle.

Náš výklad HRP bude založený na tom, čomu J. Bryson v [4] hovorí *basic reactive plan* (BRP). My ho budeme nazývať jednoducho reaktívny plán.

3.1 Reaktívny plán

Základnou stavebnou jednotkou HRP je *reaktívny plán*. Reaktívny plán je súbor podmienok a im priradených akcií, ktoré sa majú pri splnení podmienky vykonať. Ak je súčasne splnených viac podmienok, vyberie sa medzi príslušnými akciami na základe priority.

Formálne, reaktívny plán je množina trojíc (ϱ, π, α), kde ϱ je podmienka, π je priorita, α je akcia (teraz nemáme na mysli atomickú akciu). Trojicu (ϱ, π, α) nazývame *krok reaktívneho plánu*. Ak je splnená podmienka ϱ_i kroku $(\varrho_i, \pi_i, \alpha_i)$, nazveme tento krok aktívny. Zo všetkých aktívnych krokov daného reaktívneho plánu sa vyberie ten s najvyššou prioritou a jeho akcia je následne vykonaná. Ak je aktívnych viac krokov s rovnakou najvyššou prioritou, vyberie sa medzi nimi náhodne.

Podmienka je booleovský výraz, ktorý je vyhodnotený na `true` alebo `false`. Obyčajne nič nebráni tomu, aby sa v podmienkach použili ľubovoľné výpočty, ktoré nám náš programovací jazyk poskytuje (napr. aritmetické výrazy, volanie funkcií, ...). Reaktívny agent bude mať niekoľko reaktívnych plánov, ktorých podmienky testujú senzorické vstupy. Ďalším typickým predmetom testovania je pamäť, ak ňou bytosť disponuje. (Niekedy sa pamäť dokonca používa ako rozhranie medzi senzormi a mysľou. Umožňuje to väčšiu nezávislosť rozhodovacieho mechanizmu od podložného systému a tým jeho lepšiu prenositeľnosť na iné platformy. Senzorický vstup „*vidím jablko*“ sa prejaví prítomnosťou tejto informácie v pamäti.)

Prioritou je typicky číslo. Obecnejším prípadom je priorita ako funkcia času. Mohli by sme ísť ešte ďalej a definovať prioritu ako funkciu času, stavu prostredia a vnútorného stavu agenta. Príklad: keď vidím hračku, mala by stúpnuť priorita kroku s akciou „*hraj sa*.“ Tu by sme sa mali zaraziť. Veď stav prostredia a vnútra už predsa môžeme kontrolovať v podmienke. Vlastne by sme týmto mohli eliminovať podmienku (považovať ju za vždy

splnenú) a celú ju schovať do priority. Potom, keby v danej situácii pôvodná podmienka bola vyhodnotená na `false`, priorita by vrátila 0 (príp. $-\infty$). Napriek tomu sa tento prístup nepoužíva. Dôvodom je pravdepodobne fakt, že precízny popis priority by bol zbytočne komplikovaný (treba uvažovať viac faktorov), ale tiež to, že pri pôvodnom prístupe dokážeme ušetriť nejaký výpočetný čas. Pri vopred známej priorite (prípadne vypočítanej ako funkcia času) môžeme totiž testovať podmienky v jednom reaktívnom pláne v zostupnom poradí podľa priority. Keď natrafíme na prvú splnenú podmienku ϱ_i, ostatné už nemusíme vôbec vyhodnocovať, lebo už vieme, že vykonaná bude akcia α_i. Dôvod, prečo sa o tom teraz zmieňujeme, je, že sa k podobnej myšlienke na chvíľu vrátime v kapitole 5.1.

Akciou α je buď sekvencia primitívnych akcií alebo iný reaktívny plán. Sekvenciou akcií je, samozrejme, aj jediná primitívna akcia. Sekvencia viac ako jednej akcie by mala byť použitá iba v prípade, že jej akcie môžu nasledovať bezpečne za sebou. Navyše, dlhé sekvencie znižujú agentovu reaktívnosť, preto by mali byť používané s rozvahou.

V sekvencii sa okrem fyzických akcií môžu vyskytovať aj akcie mentálne (napr. hľadanie cesty na základe znalostí o svete).

3.2 Správanie pomocou reaktívneho plánu

Objasníme princíp návrhu správania pomocou RP na príklade. Budeme modelovať oberanie jahôd v záhrade. Reaktívny plán pre zbieranie jahôd zobrazuje tabuľka 3.1.

podmienka	**priorita**	**akcia**
„*v záhrade nie sú jahody*“	6	„*úloha splnená*“
„*máš plný košík*“	5	„*odnes košík do kuchyne*“
„*máš košík*“ & „*si v záhrade*“	4	„*zober najbližšiu jahodu*“
„*máš košík*“	3	„*príď do záhrady*“
„*vidíš prázdny košík*“	2	„*vezmi košík*“
$\emptyset$	1	„*hľadaj prázdny košík*“

Tab. 3.1 Reaktívny plán zbierania jahôd

Jednotlivé riadky tabuľky sú kroky reaktívneho plánu. Postup plnenia plánu je zrejmý:

- ak v záhrade nie sú jahody[2], oberanie končí (úspešne), inak sa skúsi ďalší krok s nižšou prioritou
- ak má agent plný košík, odnesie ho do kuchyne, inak sa skúsi ďalší krok s nižšou prioriotou

⋮

[2] Táto podmienka je trochu komplikovaná. Nedá sa vyhodnotiť s istotou a pritom jednoducho. Bytosť obyčajne nemá možnosť zistiť stav jahôd v záhrade, zvlášť ak sa v záhrade práve nenachádza. Pre jednoduchosť predpokladajme, že ak je v záhrade, dokáže podmienku vyhodnotiť správne a ak tam nie je, dokáže výsledok odhadnúť na základe svojej pamäte a znalostí o raste jahôd.

∅ znamená prázdnu podmienku.

Všimnime si, že napr. v podmienke kroku s akciou „*zober najbližšiu jahodu*“ už nemusíme testovať, či vôbec nejaké jahody sú a či košík nie je náhodou plný. Ak sa totiž dostane na testovanie podmienky tohto kroku, znamená to, že neboli splnené podmienky krokov s vyššou prioritou.

Bežný priebeh správania by bol nasledovný: postavička by našla a vzala košík, prišla do záhrady, zbierala jahody, kým by nebol košík plný alebo neobrala všetky. Keby naplnila košík a ešte zostali nejaké jahody v záhrade, košík by odniesla do kuchyne. Potom by znovu začala hľadať prázdny košík... Skončila by, keď by boli všetky jahody obraté.

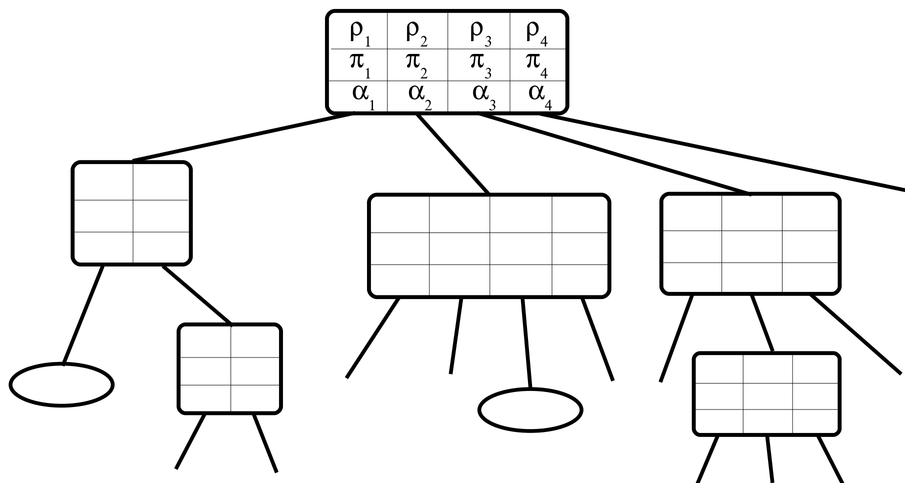
Akcia „*úloha splnená*“ je špeciálny príkaz, ktorý spôsobí (úspešné) ukončenie správania. Ak žiadna podmienka nie je splnená, správanie tiež končí (tentokrát neúspešne). Väčšinou medzi úspešným a neúspešným ukončením správania potrebujeme rozlišovať (viď časť 3.5.ii).

Existuje jednoduchý návod, ako pre činnosť napísať reaktívny plán. Tento návod pochádza z [4]:

> Predstavíme si najhorší možný priebeh činnosti (tj. nič nemáme splnené vopred). Napíšeme si jeho akcie, s vynechaním opakujúcich sa akcií. Pridáme akciu pre splnenie cieľa. Priradíme priority tak, že posledná akcia má najvyššiu, predposledná nižšiu... Potom nastavíme podmienky, počnúc od najvyššej priority po najnižšiu, podľa toho, či môže byť príslušná akcia vykonaná. Pritom v žiadnom kroku nemusíme opakovať podmienku z kroku s vyššou prioritou.

3.3 Hierarchia reaktívnych plánov

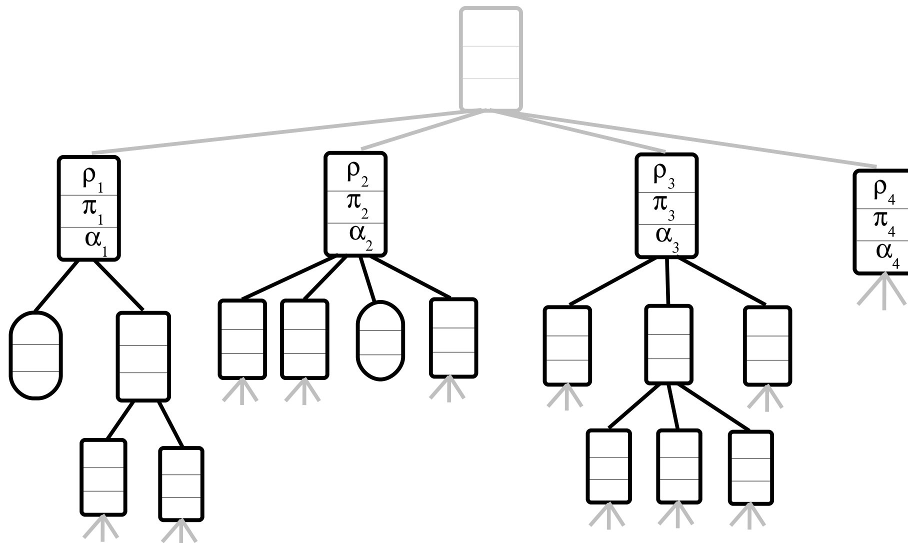
V našom príklade so zbieraním jahôd je akcia každého kroku (okrem splnenia úlohy) zložitejšia než jediná atomická akcia alebo pevne daná sekvencia akcií. Každá z nich bude ďalší reaktívny plán. Zbieranie jahôd môže byť zároveň súčasťou nejakého „vyššieho“ reaktívneho plánu, napríklad starania sa o záhradu. Vznikne tak hierarchia reaktívnych plánov. Vrcholom hierarchie môže byť plán „*ži*“, ktorý riadi celý život simulovanej bytosti.



Obr. 3.2 Hierarchia reaktívnych plánov

Schematické znázornenie stromu hierarchie reaktívnych plánov vidíme na obrázku 3.2. Toto znázornenie dostaneme priamym prepisom podľa definície. Jeden uzol odpovedá jednému reaktívnemu plánu alebo jednej sekvencii jednoduchých akcií. Reaktívny plán má tvar tabuľky, sekvencia má tvar elipsy. Pre prehľadnosť niektoré vetvy nie sú dokončené. V uzle reaktívneho plánu sú kroky, trojice (podmienka, priorita, akcia), znázornené v stĺpcoch. Je to vlastne tabuľka 3.1 otočená o 90° po smere hodinových ručičiek. Z políčka pre akciu vedie hrana k ďalšiemu RP alebo sekvencii. Koreňom stromu je plán „*ži*“. Správania odpovedajúce jeho krokom nazývame *top-level správania.* Ciele, ktoré sú týmito správaniami plnené, nazývame *top-level ciele.*

V našej definícii reaktívneho plánu má prioritu a podmienku priradenú krok reaktívneho plánu, nie reaktívny plán samotný. Zo stromu reaktívnych plánov vidíme, že každému RP okrem koreňa by sme mohli priradiť podmienku a prioritu, totiž tú, ktorú má odpovedajúci krok nadradeného plánu. Rovnako môžeme priradiť podmienku a prioritu každej sekvencii. V danej hierarchii si potom môžeme sekvenciu predstaviť ako trojicu (ϱ, π, σ), kde ϱ je podmienka, π je priorita a σ je samotná sekvencia akcií, ako sme ju chápali predtým. Reaktívny plán môžeme chápať ako trojicu $(\varrho, \pi, \mathcal{A})$, kde $\mathcal{A}$ je množina reaktívnych plánov a sekvencií, ako ich chápeme podľa tejto novej „definície“. Čo týmto pohľadom získame? Možno trochu prehľadnejšie znázornenie hierarchie (viď obr. 3.3). U jediného koreňa hierarchie by potom podmienka mala byť prázdna (vždy splnená) a na priorite vôbec nezáleží. Obyčajne potom tento koreň hierarchie vôbec neuvažujeme a strom sa nám rozpadne na les top-level správaní/cieľov. Na obr. 3.3 je znázornený nový pohľad na hierarchiu z obr. 3.2.



Obr. 3.3 Iný pohľad na hierarchiu reaktívnych plánov

V ďalšom sa budeme držať skôr tohto nového pohľadu na hierarchiu. Pôvodnú definíciu reaktívneho plánu však určite meniť nebudeme. Počas návrhu správania pomocou reaktívneho plánu tento plán totiž žiadnu prioritu nemá. Tú dostane až v hierarchii, keď sa stane súčasťou nejakého

nadradeného plánu. Priorita krokov plánu je však pri návrhu plánu podstatná.

3.4 Lezenie po stromoch

Popíšeme teraz výber akcie v hierarchii reaktívnych plánov. Na hierarchiu sa dívame ako na les top-level správaní (obr. 3.3).

Postupne prechádzame korene stromov v zostupnom poradí podľa priority a testujeme ich podmienky. Akonáhle natrafíme na prvú podmienka, ktorá je splnená, vieme, že bude vykonávaná akcia uzlu s touto podmienkou. Akcia je množina reaktívnych plánov a sekvencií. Prechádzame jej prvky podľa priority a testujeme podmienku, kým nenatrafíme na prvú splnenú. Ak sa jedná o sekvenciu atomických akcií, vykonáme ich. Ak sa jedná o reaktívny plán, pokračujeme rekurzívne na prvkoch akcie tohto reaktívneho plánu.

Výber akcie (sekvencie akcií) si môžeme predstaviť ako cestu od (pomyselného) koreňa hierarchie k nejakému listu. Musíme testovať podmienky všetkých uzlov cesty, plus podmienky súrodencov každého uzlu cesty, ktorí majú vyššiu prioritu.

3.5 Nezodpovedané otázky

Počas čítania predchádzajúcich odstavcov čitateľovi asi napadlo mnoho otázok, ktoré sme neriešili. A definitívnu odpoveď nedáme ani na tomto mieste. Hierarchické reaktívne plánovanie je skôr myšlienka, ako navrhnúť reaktívneho agenta, ktorý je schopný plniť komplexné úlohy, než aby to bol podrobný návod na implementáciu. V konkrétnej implementácii potom nutne dostanú všetky otázniky nejakú podobu.

Na niektoré veci sa však teraz predsa len trochu pozrieme.

i. Má podmienka pre spustenie akcie platiť počas celého priebehu akcie, alebo stačí, keď je splnená na začiatku? Zmysel môžu mať obe možnosti. Preferovaná je varianta, kedy podmienka pre spustenie stačí skutočne pre len spustenie. Druhú možnosť potom dokážeme jednoducho nasimulovať:

> Máme podmienku ϱ, ktorá spôsobí spustenie akcie α. Ak je α reaktívny plán, pridáme doňho krok s najvyššou prioritou, podmienkou $\neg\varrho$ a akciou na ukončenie. Ak prestane platiť ϱ, začne platiť $\neg\varrho$ a reaktívny plán bude ukončený. Ak je α sekvencia, nemala by byť dlhá a jej dokončenie by nemalo spôsobiť problémy. Každopádne, je možné ju zrejmým spôsobom zmeniť na reaktívny plán s dvoma krokmi tak, aby po prerušení platnosti spúšťacej podmienky bol ukončený.

Na druhej strane, keby sme sa priklonili k druhej variante, tá prvá by sa simulovala veľmi nešikovne.

Navyše, kroky reaktívneho plánu budú obyčajne podobné tomuto:

(„*máš hlad*“, 17, „*najedz sa*“).

Ľudia aj zvieratá svoju porciu väčšinou zjedia celú. Pokiaľ by neplatnosť podmienky mala spôsobiť ukončenie akcie, zostala by nedojedená porcia,

ak by bytosť počas počas jej jedenia prestala mať hlad. (Čo je pravdepodobné, lebo pocit nasýtenia obyčajne nezískame presne po zjedení posledného tak sústa.)

Keď teda podmienku spusteného podstromu už nechceme kontrolovať skôr, ako bude ukončený, musíme si navyše pamätať cestu, ktorú sme vybrali v predchádzajúcom kroku. Po tejto ceste sa potom vrátiť o jednu úroveň vyššie a tam vybrať akciu znovu. Ak žiadna podmienka na tejto úrovni nie je splnená, alebo je splnená podmienka cieľa, vrátime sa o ďalšiu úroveň vyššie ...

Dodajme ešte, že napr. jazyk *A Behavior Language* (ABL, [5]) umožňuje zadať podmienky dve: jednu, ktorá musí byť splnená pred spustením akcie (*precondition*) a druhú, ktorá musí platiť počas celého priebehu akcie (*context condition*).

ii. Vráťme sa znovu k príkladu zbierania jahôd z 3.2. Predpokladajme, že postavička vzala košík a pokúša sa prísť do záhrady. Z nejakého dôvodu sa jej nepodarí prísť do záhrady. Mala by sa o to pokúsiť znovu? Koľkokrát? Môže sa tým dostať do nekonečného cyklu, v ktorom sa bude pokúšať dostať sa do záhrady. Tento cyklus by mohol byť prerušený iba rozhodnutím na vyššej úrovni, že je teraz niečo dôležitejšie ako zber jahôd. Avšak po splnení dôležitejšej úlohy by sa postavička opäť vrátila k zberu jahôd a cyklus by pokračoval.

Väčšinou preto rozlišujeme medzi úspešným a neúspešným ukončením akcie. Úspech sekvencie primitívnych akcií záleží od toho, či sa celú sekvenciu podarí previesť. Reaktívny plán končí úspechom po splnení podmienky pre (úspešné) ukončenie. Ak žiadna podmienka nie je splnená, končí neúspechom. Aby sa zabránilo nekonečným cyklom, neúspech jedného kroku tiež spôsobí neúspešné ukončenie celého reaktívneho plánu. To spôsobí propagovanie neúspechu až do koreňa hierarchie a spustenie mechanizmu výberu akcie od začiatku. To ale bude vo väčšine prípadov viesť k rovnakej akcii a dostaneme sa na to isté miesto a sme znovu zacyklení, len v trochu väčšom cykle.

Rozšírenie POSH (viď kap. 7.1) toto rieši tým, že umožňuje pre krok reaktívneho plánu špecifikovať počet pokusov, koľkokrát sa má skúšať ho splniť a pre top-level správanie frekvenciu, ako často môže byť navštevované.

V jazyku E (kap. 6.1) sa reaktívne plány dajú zapísať pomerne priamočiaro. Obsahuje ale omnoho viac prostriedkov a aby k popísanej nežiadúcej situácii nedochádzalo, správania sa väčšinou neprogramujú metódou čistého HRP.

Uveďme však možnú sémantiku reaktívneho plánu bez ďalších rozšírení, v ktorej by k takýmto cyklom nedochádzalo.

Pri zlyhaní kroku reaktívneho plánu nezlyhá celý reaktívny plán, ale zlyhavší krok bude z plánu (dočasne) odstránený. Po zlyhaní kroku „*príď do záhrady*“ z plánu zbierania jahôd by ale potom bola stále splnená podmienka kroku „*vezmi košík*“ a tento krok by okamžite uspel, lebo postavička košík už drží. Dostali by sme teda cyklus spúšťania a uspenia kroku „*vezmi*

košík". Pri zbieraní jahôd vlastne chceme, aby celý reaktívny plán zlyhal po neúspešnom pokuse dostať sa do záhrady. Dovolíme teda, aby každý krok s nižšou prioritou ako nejaký už odstránený uspel najviac raz. Potom ho tiež odstránime. V našom príklade teda po neúspešnom pokuse dostať sa do záhrady bude tento krok odstránený. Potom uspejú a budú odstránené kroky „*vezmi košík*" a „*hľadaj prázdny košík*". Iné kroky už nebudú mať splnenú prioritu a plán skončí neúspešne.

Vlastnosť, že nezlyhá celý plán využijeme v inom prípade. Uvažujme, že súrodencom uzlu oberania jahôd v hierarchii je plán „*upratuj*", ale má nižšiu prioritu. V pôvodnej sémantike by po zlyhaní oberania jahôd zlyhal celý ich rodič a na upratovanie by sa vôbec nedostalo (kým by bol záujem oberať jahody). Takto sa po zlyhaní oberania na upratovanie dostane. Avšak iba raz. Keby bolo povolené viackrát, už by sa zase nemuselo dostať na zbieranie jahôd, aj keď už bude cesta do záhrady priechodná, keby upratovnie chcelo bežať vždy znova. Takto, ak nechce bežať žiadny krok s vyššou prioritou ako oberanie jahôd, po odstránení všetkých krokov s nižšou prioritou, ktorých priorita je splnená, rodičovský uzol skončí neúspešne a pri opätovnej návšteve uzlu už budú všetky odstránené kroky znovu prítomné a môže sa opäť skúsiť oberanie jahôd.

Niekedy môže byť i jednorázové vykonanie krokov s nižšou prioritou nežiadúce, ak po nich nemôže byť vykonaný odstránený krok. Predstavme si, že v našom zbieraní jahôd zlyhá krok „*odnes košík do kuchyne*". Tento krok bude odstránený a potom sa postavička vráti s košíkom do záhrady, skúsi zobrať ďalšiu jahodu a až potom (bez ohľadu na to, či sa do plného košíka ešte podarí pridať jednu jahodu) zlyhá. Toto správanie už nepôsobí vierohodne. Aby sme to „opravili", už nemusíme meniť sémantiku reaktívneho plánu. Stačí do oberania jahôd pridať krok („*máš plný košík*", 4.5, „*zlyhaj*"). Na tento krok sa dostane len po odstránení kroku „*odnes košík do kuchyne*".

Táto sémantika nám zároveň umožňuje jednoducho uviesť alternatívy pre splnenie nejakého cieľa. Napríklad, polievať sa dá hadicou alebo krhlou. Pre oba druhy polievania máme navrhnutý reaktívny plán. Združíme ich do jedného plánu a stanovíme podmienku pre úspešné ukončenie polievania. Plán polievania potom môže vyzerať takto:

podmienka	**priorita**	**akcia**
„*záhrada poliata*"	3	„*úloha splnená*"
$\emptyset$	2	„*polievaj hadicou*"
$\emptyset$	1	„*polievaj krhlou*"

Tab. 3.4 Reaktívny plán s alternatívami

Teraz, keď bude treba polievať, skúsi sa polievať hadicou. Ak sa to z akéhokoľvek dôvodu nepodarí (napríklad hadica je deravá), bude sa polievať krhlou. Jediné, čo sa nám nemusí páčiť, je, že o poradí skúšania alternatív je rozhodnuté vopred—pri návrhu. Vždy sa teda najsôr skúsi polievať hadicou a potom krhlou, nie opačne.

iii. Ďalšou otázkou je, či by sa malo po prerušení vykonávania nejakého (pod)stromu a opätovnému návratu k nemu vrátiť na to isté miesto do

stromu, kde bola jeho činnosť prerušená, alebo spustiť výber akcie v strome znovu. Vo väčšine prípadov nebude mať návrat na to isté miesto zmysel a spôsobil by zlyhanie tohoto stromu. Predstavme si, že postavička zbiera jahody a opakovane vykonáva krok „*zober najbližšiu jahodu*“. V tom je prerušená napríklad jedením. Až sa naje, zobrať ďalšiu jahodu nemá zmysel a v kuchyni sa to ani nepodarí. Keď ale prerušujúca činnosť nenaruší kontext prerušenej, návrat na pôvodné miesto zmysel má. Napríklad pri dialógu s kamarátom. Mohli by sme si tým teda ušetriť nejaké testovanie podmienok a lezenie po stromoch. Zároveň to ale vyžaduje udržovanie viacerých ciest od koreňa k listu. Pri prerušení by sme si poslednú cestu zapamätali a vybrali novú. Keď by sa nová „odrolovala“ na nejakú spoločnú začiatočnú časť so starou (kľudne prázdnu, ak sú v rôznych top-level stromoch), stará by bola automaticky považovaná za aktívnu (tj. že má splnené podmienky).

3.6 Nedostatky HRP

Niektoré veci sa v HRP vyjadrujú obtiažne, alebo sa nedajú vyjadriť vôbec. V článku [3] sa C. Brom pokúsil o zhrnutie týchto nedostatkov. Nejedná sa o problémy prežitia a plnenia úloh. V tom si HRP počína pomerne dobre. Sú to hlavne problémy s nedostatočnou biologickou vierohodnosťou. My uvedieme dva z nich, ktoré sa v nasledujúcich dvoch kapitolách pokúsime vyriešiť. Pre úplnejší zoznam čitateľa odkazujeme na spomínaný článok.

Niekedy je pri prechode medzi dvoma top-level správaniami vhodné vykonať krátke prechodné správanie, aby prepnutie prebehlo „hladko“. Ak si naša postavička počas oberania jahôd potrebuje odskočiť na záchod, nemala by na záchod odísť s košíkom v ruke. Toto sa však v klasickom HRP stane. Prirodzenejšie je košík odložiť, kľudne rovno v záhrade. V kapitole 4 rozšírime HRP o prechodné správania.

Navonok podobným problémom je dočasné odloženie činnosti v prospech práve bežiacej. Keď si postavička počas oberania potrebuje odskočiť, ale oberanie je už pred koncom (napr. zostáva obrať posledných 5 jahôd), mohla by s odskočením si počkať a najskôr dokončiť oberanie. V HRP však bude oberanie nekompromisne prerušené. V kapitole 5 HRP ďalej rozšírime tak, aby mohla byť jednoducho uskutočnená dohoda o odložení správania.

Obe rozšírenia pritom budú spätne kompatibilné s doteraz uvažovaným HRP, takže ak niektoré správania máme napísané v HRP a rozšírenia pre ne nebudeme chcieť využiť, budú bez zásahov fungovať tak ako doteraz.

4. Prechodné správania

Prechodné správanie je krátka činnosť, ktorej úlohou je hladký prechod medzi dvoma hlavnými správaniami. Význam prechodných správaní je jednak v praktickosti a jednak v biologickej vierohodnosti.

Pri prepnutí zo správania A na správanie B budeme o správaní A hovoriť ako o prerušovanom a o B ako o prerušujúcom. Problém HRP je, že prerušenie správania A nastane náhle, bez varovania a možnosti dať veci do nejakého konzistentného stavu.

Prechodné správanie je potrebné iba v prípade, že prvé správanie nebolo riadne ukončené, ale prerušené iným. Dobre navrhnuté správanie by totiž malo po sebe „upratať“ samo.

Praktický význam prechodných správaní je v tom, že uvedú prerušované správanie do bezpečného stavu, aby nedošlo k nejakej „havárii“, alebo do stavu, z ktorého možno bezproblémovo a čo najrýchlejšie pokračovať.

Význam biologickej vierohodnosti je zrejmý—živé bytosti (najmä ľudia) proste prechodné správania vykonávajú, preto ich očakávame aj od bytostí virtuálnych. Kým tie živé si to väčšinou vôbec neuvedomujú, u virtuálnych je potrebné všetko vyjadriť explicitne.

Ukážme si potrebu prechodných správaní na niekoľkých príkladoch zo života skutočných bytostí.

Uvažujme človeka, ktorý varí. V tom zazvoní telefón, tak ho ide zdvihnúť. Pretože sa mieni k vareniu po skončení hovoru vrátiť, neukončí ho, ale len stlmí sporák, aby mu medzitým jedlo nevykypelo. Stlmenie sporáku je prechodné správanie. Význam je jasný—predídenie havárii. Prechodné správanie bolo výsledkom ľudského uvažovania.

Uvažujme inú situáciu. Postavička pozerá film z DVD a potrebuje si odskočiť na záchod. Film si preto pozastaví a keď sa vráti, nechá ho pokračovať. Prechodným správaním bolo pozastavenie filmu. Postavička ho pozastavila preto, aby mohla pokračovať v sledovaní od miesta, kde bol film prerušený a nič jej neušlo. Opäť výsledok uvažovania.

Predstavme si človeka, ktorý na záhrade okopáva záhony. Rozhodne sa, že si urobí prestávku na jedlo. Dookopáva začatý záhon, položí motyku a ide sa najesť do kuchyne. Keď sa naje, vráti sa do záhrady, vezme motyku a pokračuje v okopávaní. Prechodným správaním bolo dokončenie aktuálneho záhonu a odloženie motyky. Človek má tendenciu zanechať veci v nejakom celistvom stave. Preto okopávanie začatého záhonu dokončí. O odložení motyky asi veľmi nepremýšľal, urobil to akosi automaticky. Pre človeka je totiž jednoduchšie motyku položiť, než ísť s ňou do kuchyne. Umelá bytosť ale nemá schopnosť vycítiť, že je niečo jednoduchšie, ak jej to explicitne nenaprogramujeme.

Uvážme podobnú situáciu, človeka, ktorý okopáva. Tentokrát ho však preruší zvonenie mobilu. Na to potrebuje zareagovať hneď, preto len položí motyku a zdvihne telefón. Prechodným správaním je odloženie motyky. Vidíme, že prechod z jedného správania môže byť rôzny v závislosti na pre-

rušujúcom správaní. Tentokrát totiž záhon nedookopával.

Uveďme aj jeden príklad zo zvieracieho sveta. Majme opicu, ktorá sa momentálne zaoberá plastovou fľašou a „rozmýšľa“, na čo by jej mohla byť dobrá. V tom zbadá pár krokov od seba loptu, čo sa jej zdá na prvý pohľad predmet zaujímavejší, tak sa rozhodne preskúmať, čo sa s ňou dá robiť. Fľašu teda odhodí a vydá sa za loptou. Odhodenie fľaše bolo prechodným správaním medzi „*skúmaním fľaše*“ a „*skúmaním lopty*“. Hoci sa opica po preskúmaní lopty asi neplánuje vrátiť k skúmaniu fľaše, skúmanie fľaše nebolo riadne ukončené a prechodné správanie je teda namieste.

Prechodné správania sú často výsledkom uvažovania (hoci aj podvedomého). Preto sa s nimi stretávame hlavne u ľudí. No v predchádzajúcom odstavci sme si uviedli príklad, že v nejakej forme ich možno nájsť aj u zvierat.

4.1 Pokus o implementáciu čisto v HRP

Ako môžeme prechodné správania vyjadriť v HRP? Veľmi šikovne nie.

Správania by sa mohli o všetko starať samé. Pri „násilnom“ prerušení správania A a spustení B by sa správanie B postaralo o vykonanie prechodného správania. V skuto4nosti by to síce nebolo samostatné správanie stojace medzi správaniami A a B, lebo by bolo súčasťou správania B, ale na tom veľmi nezáleží. Z hľadiska pozorovateľa nie je dôležité, ako je to vnútri urobené, ale že to navonok pôsobí rozumne.

Takže správanie „*telefonovanie*“ sa na začiatku pozrie, či náhodou nie je zapnutý sporák a prípadne ho stlmí. Ale telefonovanie môže prerušiť aj inú činnosť ako varenie, napr. pozeranie filmu. Preto ešte skontroluje, či náhodou nebeží film a prípadne ho stopne. Telefonovaním môže byť prerušené aj umývanie riadu, preto si skontroluje, či nemá náhodou mokré ruky a prípadne si ich osuší. Varenie, pozeranie filmu a umývanie riadu môžu byť prerušené aj inou činnosťou ako telefonovaním, napr. odskočením si na záchod. Preto si aj toto správanie musí kontrolovať všetky možné nežiadúce východiskové situácie.

Týmto spôsobom by sa popis správaní rýchlo stal neprehľadný a neudržiavateľný. Nehovoriac o tom, že rôzne správania môžu byť navrhnuté rôznymi autormi, ktorí v čase návrhu nevedia, aké ostatné správania budú u bytosti prítomné. Každému je asi jasné, že tadiaľto cesta nepovedie.

4.2 Rozšírenie HRP o prechodné správania

V predchádzajúcej časti sme ukázali, že v HRP sa prechodné správania dajú vyjadriť len veľmi ťažko. Model HRP preto rozšírime tak, aby sa prechodné správania zapisovali jednoducho. Rozšírenie bude jednoduché na použitie aj implementáciu.

Celé rozšírenie spočíva v tom, že k hierarchii reaktívnych plánov pridáme štvorcovú maticu T prechodných správaní. Riadky aj stĺpce matice sú označené reaktívnymi plánmi. Nech A, B sú reaktívne plány. Prvok T_{AB} matice prechodných správaní bude akcia reprezentujúca prechodné správanie z A na B.

Od nástroja, ktorý nám umožňuje zapisovať reaktívne plány, budeme odteraz požadovať, aby nám umožnil špecifikovať prvky matice prechodných správaní. Implicitne bude každý prvok matice považovaný za prázdny.

Naša matica je len formálna, implementačne to matica vôbec nemusí byť. Môže to byť napríklad zoznam usporiadaných dvojíc, zotriedený, aby sa v ňom rýchlejšie vyhľadávalo. Tým ušetríme nejaké miesto, ak sa predpokladá riedke zaplnenie matice (tj. málo prechodných správaní).

Potom, keď je nejaké správanie prerušované iným, vyhľadá sa odpovedajúci prvok matice a prípadná nájdená akcia sa vykoná. Čo ak je však prerušené prechodné správanie? Úlohou prechodného správania je „upratať“ po prerušovanom správaní. Ak je prechodné správanie z A na B prerušené správaním C, znamená to, že po správaní A ešte nebolo úplne upratané. Vykoná sa teda prechodné správanie z A na C, ktoré upratovanie po A dokončí s ohľadom na C.

Všimnime si, že tento model nás neobmedzuje na prechodné správania len medzi top-level správaniami, ale umožňuje zadať prechodné správanie medzi ľubovoľnými správaniami. Správanie, ktoré je prerušované, je potom taký predok práve vykonávaného listu hierarchie, ktorý je súrodencom prerušujúceho správania.

Väčšinou budú zaujímavé iba prechody medzi top-level správaniami. Ak sú v našom modelovacom nástroji top-level správania nejako odlíšené od ostatných, môžeme prechodné správania obmedziť len pre top-level správania. Ale napr. Expressivator (viď kap. 7.2) používa prechody aj medzi správaniami na nižšej úrovni.

5. Odloženie správania

Uvažujme človeka, ktorý zalieva záhony. Dostane hlad a tak sa potrebuje najesť. Ak má zalevania ešte veľa, urobí si prestávku a pôjde sa najesť. Medzi zalievaním a jedením vykoná prechodné správanie, ktorým môže byť vypolievanie vody, ktorú má práve v krhle a odloženie krhly. Takéto správanie už vieme namodelovať pomocou HRP rozšíreného o prechodné správania.

Uvažujme trochu odlišnú situáciu. Človek zalieva záhony a dostane hlad. Vidí, že zalievanie už má skoro hotové, a tak s jedením počká, kým nezaleje zvyšných niekoľko záhonov. Potom uprace krhlu a pôjde sa najesť. V tomto prípade sa nejednalo o žiadne prechodné správanie. Zalievanie bolo riadne dokončené, lebo jedenie mohlo počkať. Hoci z dlhodobého hľadiska je pre človeka jedenie dôležitejšie ako zalievanie, zalievanie vyhralo. Človeku stúpa motivácia dokončiť započatú prácu, keď vidí, že sa už blíži k cieľu.

HRP ale nemá prostriedok na vyjadrenie meniacej sa motivácie. O dôležitosti každej úlohy rozhodne priorita, ktorá je úlohe pevne daná pri jej začlenení do hierarchie, buď ako konštanta alebo funkcia času. Neberie však do úvahy momentálnu motiváciu, ktorú živý vzor modelovanej bytosti pociťuje.

5.1 Prečo nie dynamická zmena priority

Riešením by mohlo byť, dať správaniam možnosť upraviť si prioritu. Ako sme naznačili v kapitole 3.1, priorita by závisela nielen od času, ale aj od aktuálneho stavu správania. Napr. jazyk E Projektu ENTi upravovanie priorít umožňuje. Autor by potom pri návrhu správania explicitne upravoval motiváciu pre toto správanie upravovaním priority tak, aby odpovedala momentálnemu záujmu o aktuálnu činnosť s ohľadom na ostatné správania. Uvedieme však dôvody, prečo to nie je dobrým riešením.

Explicitné upravovanie priority je akýmsi neštandardným prostriedkom. Správanie si počas svojho návrhu vôbec nie je vedomé svojej priority. Keď bude spustené, malo by fungovať dobre pri ľubovoľnej priorite. Ak si ju ale má akokoľvek upravovať, hoci aj v relatívnych pojmoch (tj. „*zvýš prioritu o 12%*“, nie „*zvýš prioritu o 12 (jednotiek)*“) nutne si jej musí byť vedomé. Čo viac, aby mohlo svoju prioritu upraviť adekvátne, musí poznať aj priority ostatných správaní a vzťah svojej priority k nim. Tým zároveň porušuje zásadu tzv. *na správaniach založeného návrhu* (*behaviour-based design*), pri ktorom celkové konanie umelej bytosti rozkladáme na *navzájom nezávislé* správania. Je porušením práve vzájomnej nezávislosti správaní.

Ďalším závažným dôvodom proti dynamickej zmene priority je fakt, že dosiahnutie dobrých výsledkov týmto spôsobom je značne zložité. To vyplýva už aj z toho, že autor pri návrhu musí vedieť o ostatných správaniach a ich prioritách. Okrem toho, dostatočne dobré nastavenie priorít, aby sa bytosť správala vierohodne aspoň vo väčšine štandardných situácií, by vyžadovalo množstvo ladenia a experimentovania.

5.2 Rozšírenie HRP o odloženie správania

Budeme o odložení správania v prospech práve prebiehajúceho uvažovať v pojmoch času. Ľuďom ako autorom správania sa v pojmoch času uvažuje ľahšie ako v nejakých abstraktných prioritách. Prirodzenejšie je nám špecifikovať, že na jeden záhon postavička potrebuje 3 minúty, než že s každým ďalším zaliatym záhonom má priorita zalievania stúpnuť o 12(%). Podobne jednoducho dokážeme určiť, že keď má postavička hlad, ale zatiaľ ešte malý, môže s jedením kľudne hodinu počkať, ak má stredný hlad, môže počkať pol hodiny a ak má obrovský hlad, mala by sa najesť čo najskôr.

Rozšírenie bude preto spočívať práve v možnosti uviesť takéto časové údaje. Správaniu, ktoré chce začať, dáme možnosť špecifikovať, ako dlho ešte môže počkať (tzv. *tolerancia*) a bežiacemu správaniu možnosť odhadnúť čas potrebný na svoje dokončenie.

Vráťme sa k príkladu z úvodu tejto kapitoly. Postavička zalieva a dostane hlad. Hlad ale ešte nie je veľký a jedenie povie, že ešte môže pol hodinu počkať. Zalievanie vidí, že zostáva zaliať 5 záhonov a vie že na jeden záhon potrebuje zhruba 3 minúty a na upratanie krhly ďalšie 3 minúty. Odhadne teda, že skončí za 18 minút. Keďže jedenie môže počkať dostatočne dlho, bude odložené.

Keby zalievanie uviedlo odhad 45 minút, jedenie by odložené nebolo. Odloženie o 45 minút by totiž znamenalo, že sa jedenie nedostane k slovu do doby, kedy už nutne musí. Odloženie na kratší čas, napr. 30 minút, ktoré môže počkať, by nemalo zmysel, lebo dovtedy by zalievanie nestihlo skončiť a aj tak by bolo prerušené. Preto bude zalievanie prerušené ihneď. Tento výsledok je rovnaký ako bez rozšírenia o toleranciu a odhad. Rovnako teda bude spustené prechodné správanie medzi zalievaním a jedením.

Zaujímavejší je prípad, keď sa zalievaniu nepodarí skončiť v stanovenom limite. Počkať, čo je vlastne stanovený limit? Mal by byť limitom odhad, ktorý zalievanie uviedlo, alebo tolerancia jedenia (ktorá môže byť vyššia)? Pokiaľ zalievanie odhad podcenilo a potrebuje viac času, ale jedenie môže stále ešte čakať, nie je dôvod, prečo by zalievanie nemohlo pokračovať. Jedenie bude teda odložené na čas svojej tolerancie. Zalievanie by malo byť prerušené až v momente, kedy už jedenie naozaj nemôže čakať. Tento moment, prekvapivo, nemusí byť zhodný s okamihom vypršania tolerancie. Jedenie mohlo predtým svoju akútnosť preceniť a uviedlo pesimistický odhad. Teraz by mohlo povedať, že ešte môže počkať ďalších 10 minút. Na druhej strane, zalievanie teraz odhaduje, že skončí za 6 minút a tak dostane ďalšiu šancu. Iným prípadom je, keď odložené správanie po vypršaní doby odloženia už nechce bežať vôbec. Toto sa s jedením asi nestane, lebo hlad nezmizne len tak. Ak ale odloženým správaním bolo kŕmenie psa a medzitým ho nakŕmil niekto iný, zalievanie môže pokračovať bez ďalších obmedzení ďalej. Keby však po vypršaní tolerancie jedenie už nemohlo čakať (aspoň nie toľko, koľko zalievanie potrebuje), bude zalievanie prerušené. Otázkou zostáva, či by po nevyužitej šanci na dokončenie zalievania ešte malo byť spustené prechodné správanie. Odpoveď je áno, pretože potreba položiť krhlu pred odchodom do kuchyne je tu stále. Je ponechané na autorovi prechodného správania,

aby netrvalo príliš dlho a nespôsobilo tým nejaké škody (napr. žalúdočné problémy, lebo žalúdok už od hladu začal tráviť sám seba).

Predpokladajme teraz, že beží zalievanie. Chce bežať jedenie, ale môže počkať a je odložené. Zalievanie teda pokračuje. O chvíľu si postavička potrebuje odskočiť na záchod, čo má najvyššiu prioritu z týchto troch správaní. Rozlíšime 2 možnosti:

i. odskočenie si tiež môže počkať. V tom prípade zalievanie pokračuje ďalej. Jedenie aj odskočenie si obe čakajú na dokončenie zalievania.

 α) Ak zalievanie skončí pred vypršaním oboch tolerancií, postavička pôjde na záchod, pretože to má vyššiu prioritu a nie je dôvod nijako favorizovať jedenie, pretože jeho činnosť zatiaľ vôbec nezačala.

 β) Ak ako prvá vyprší tolerancia záchodu, odskočenie si prehodnotí svoj záujem a nutnosť a môže dať ďalší čas zalievaniu. Ak mu dá ďalší čas, vedie to opäť na prípad i. Ak mu čas nedá, je jasné, že zalievanie bude prerušené. Pretože žiadna iná činnosť nie je rozrobená, postavička si po prípadnom prechodnom správaní odskočí na záchod. Keď sa vráti, je síce rozrobené zalievanie, ale už bolo prerušené a spustí sa jedenie. Jedenie nemá dôvod čakať, pretože význam jeho odloženia bol v tom, aby zalievanie prerušené nebolo. Zalievanie už ale prerušené bolo, preto nemá zmysel ďalej čakať.

 γ) Ak ako prvá vyprší tolerancia jedenia, jedenie prehodnotí svoj záujem a nutnosť. Ak môže čakať ďalej, je to prípad i. Ak nie, je už na tomto mieste jasné, že zalievanie bude prerušené. Určite teda bude vykonané prechodné správanie *zo* zalievania. Žiadne iné správanie nie je rozrobené a najvyššiu prioritu zo správaní, ktoré chcú bežať, má odskočenie si. Nič teda nepokazíme, ak teraz spustíme odskočenie si. Rovnako, ako v predošlom odstavci, význam odloženia záchodu bol v tom, aby nebolo prerušené zalievanie, čo teraz nutne bude.

 Jedenie je ale to, ktoré už nemohlo počkať na dokončenie zalievania a ono sa postaralo o prerušenie zalievania. Preto, ak záchod ešte stále môže počkať toľko, koľko jedenie potrebuje, mohlo by sa dostať k slovu jedenie. Bola by to akási odmena za to, že práve ono prerušilo zalievanie. Inak by totiž odskočenie si aj tak stále čakalo.

 Ak odskočenie nemôže čakať na dokončenie jedenia, je jasné, že treba ísť na záchod, lebo má vyššiu prioritu. V prípade, keď potreba na záchod môže dosť dlho počkať, nepovažujeme za zásadné, ktoré správanie sa spustí prvé. V implementácii v Projekte ENTi (popísanej v kap. 6) je toto vyriešené tak, že jedenie by šancu dostalo, ale najviac jednu. To sa uplatní v prípade, keby na ukončenie zalievania čakalo ešte ďalšie správanie (napr. pitie) s prioritou medzi jedením a odskočením si. Po tom, čo by jedenie dostalo „za odmenu" šancu bežať prvé, dostalo by sa k slovu pitie, lebo ono by malo potom najvyššiu prioritu zo správaní, ktoré chcú bežať. Ak by pitie nebolo ochotné čakať na jedenie, spustí sa pitie a po ňom už jedenie ďalšiu odmenu od odskočenia si nedostane a pôjde sa na záchod. Nakoniec

sa vráti k zalievaniu.

ii. odskočenie si už nemôže dosť dlho čakať. Tento prípad je rovnaký ako i.β), keď počas čakania oboch vyprší tolerancia odskočenia si a zalievanie nedostane ďalšiu šancu.

Ďalšia možná situácia je, keď beží zalievanie a začne sa hlásiť o slovo potreba na záchod. Tá ešte môže počkať a je odložená. Počas tohto čakania postavička dostane hlad a chce sa najesť, čo má strednú prioritu z týchto troch. Táto situácia je však rovnaká, ako prípad i.γ) z predchádzajúceho odstavca, keď obe správania čakajú a prvá vyprší tolerancia jedenia.

5.3 Formálne rozšírenie reaktívneho plánu

V tejto časti zavedieme odhad a toleranciu do formálnej definície reaktívneho plánu.

Podmienka, ktorá spúšťa jedenie, obyčajne potrebuje kontrolovať hlad. Podmienka, ktorá spúšťa polievanie, asi kontroluje stav záhonov. Pre určenie, ako dlho jedenie môže počkať, potrebujeme vedieť úroveň hladu. Pre určenie, ako dlho môže počkať zalievanie, potrebujeme poznať stav záhonov. Spusteniu správania musí predchádzať vyhodnotenie podmienky. Toleranciu správania potrebujeme poznať len po splnení podmienky. Vidíme, že vyhodnotenie podmienky a určenie tolerancie majú spoločné rysy.

Podmienka bola doteraz booleovský výraz, ktorý je vyhodnotený na `true` alebo `false`. Odteraz podmienkou bude aritmetický výraz, ktorý sa vyhodnotí na číslo (celé, racionálne, reálne). Nezáporná hodnota bude znamenať, že podmienka je splnená a udávať hodnotu tolerancie (vo zvolených jednotkách času). Záporná hodnota znamená, že podmienka nie je splnená.

Podmienka ϱ kroku (ϱ, π, α) reaktívneho plánu sa vyhodnocuje pred spustením akcie α. Obdobu, ktorá by sa vyhodnocovala po prerušení alebo ukončení akcie nemáme. Odhad preto zavedieme ako samostatnú funkciu, ktorá má zhodnotiť situáciu a odhadnúť čas potrebný na dokončenie. Pritom stačí, aby funkcia dávala rozumné odhady iba v prípade, že prísušná akcia chce bežať. (Túto skutočnost však asi pri konštrukcii tejto funkcie nevyužijeme.)

Krok rozšíreného reaktívneho plánu je teda štvorica $(\varrho, \pi, \alpha, \epsilon)$, kde ϱ je podmienka vracajúca v prípade splnenia toleranciu, π a α sú priorita a akcia ako predtým a ϵ je nezáporná funkcia vracajúca odhad času potrebného pre dokončenie α. ϵ bude voliteľný parameter, ktorý netreba zadávať. V tom prípade bude odhad považovaný za ∞. To je v súlade s predchádzajúcou sémantikou reaktívneho plánu. Pre spätnú kompatibilitu ešte umožníme, aby podmienka mohla byť zadaná ako booleovský výraz, ktorý v prípade splnenia bude zároveň znamenať nulovú toleranciu.

6. Implementácia v Projekte ENTi

Projekt ENTi je nástroj na modelovanie ľudských bytostí, hlavne ich správania. Modelovaná bytosť je v ňom označovaná *ent*.

Podrobnou dokumentáciou projektu je [2]. Stručný úvod a porovnanie s inými projektami možno nájsť v článku [1].

Nástroj pozostáva z troch nezávislých častí: server prostredia, simulátor (jedného) enta a užívateľské grafické rozhranie.

Server prostredia simuluje svet a telá entov v tomto svete. Svet je rozdelený do miestností (napr. i záhrada je považovaná za miestnosť). Vo svete sa nachádzajú objekty, s ktorými enti môžu manipulovať.

Simulátor enta riadi entovo správanie. Pre nás je teda najzaujímvejšia práve táto časť projektu. Dá sa považovať za entovu myseľ.

Čas vo svete entov beží v diskrétnych kolách (krokoch). Pritom sa striedajú dni a čas sa zobrazuje v ľudskej podobe (jedna minúta simulačného času sú 3 kolá). V každom kole simulátor enta pošle *atomickú inštrukciu* serveru prostredia. Ten sa ju s telom enta pokúsi vykonať a výsledok (úspech/neúspech) pošle späť simulátoru enta. Atomická inštrukcia sa nemusí podariť (napr. otvoriť zamknuté dvere, krok na miesto, kde niekto stojí . . .). Zároveň server pošle informáciu o zmene stavu sveta. Telo enta je celé v réžii serveru prostredia, simulátor enta len posiela inštrukcie a dostáva odpovede. Vlastnosti ako hlad alebo únava tiež riadi server.

K serveru môže byť pripojených viac simulátorov enta a viac grafických rozhraní.

Cez *grafické rozhranie* má užívateľ možnosť život entov nielen sledovať, ale doňho aj zasahovať. Každému grafickému rozhraniu odpovedá jedno telo vo svete, ktoré však nie je riadené simulátorom enta, ale užívateľom. Rozhranie umožňuje „ľudskému entovi“ posielať rovnaké atomické inštrukcie ako simulovaným entom.

6.1 Jazyk E

Správanie entov sa programuje v *jazyku E*. Jeho interpret je súčasťou simulátoru enta. E je programovací jazyk navrhnutý s ohľadom na programovanie (modelovanie) virtuálnych bytostí. Bol vytvorený špeciálne pre Projekt ENTi.

Jazyk E je popísaný v *Príručke autora sveta a skriptov*, ktorá je súčasťou dokumentácie projektu ([2]).

Základnou jednotkou správania v jazyku E je tzv. *b-skript* (*behavioural script*). Je to akási obdoba procedúry/funkcie/metódy z bežných programovacích jazykov. V nasledujúcom texte bude slovom *skript* myslený b-skript.

Skript nemá návratovú hodnotu. Jeho výsledkom je úspech alebo zlyhanie. Možno si teda predstaviť, akoby každý skript vracal `true` alebo `false`. Rovnako, ako funkcie v bežných jazykoch, skript môže mať parametre. I keď skript nemá návratovú hodnotu, je možné hodnotu vypočítanú v skripte dostať von cez parameter.

Významným prostriedkom jazyka E s ohľadom na návrh správania sú tzv. *interrupty.* Interrupty nám umožňujú povedať interpretu, aby kontroloval nejakú podmienku za nás a keď bude splnená, spustil nejakú akciu. O tomto požiadaní o kontrolu podmienky hovoríme ako o navesení interruptu. Interrupt môže byť lokálny alebo globálny. Podmienka lokálneho interruptu sa kontroluje len počas behu skriptu, ktorý ho navesil. Podmienka globálneho interruptu sa kontroluje „stále“. Každý interrupt má prioritu, ktorá je rozhodujúca v prípade, keď je súčasne splnená podmienka viacerých navesených interruptov. V tom prípade je spustená akcia interruptu s vyššou prioritou.

Čitateľ už asi vidí podobnosť s reaktívnym plánom. Interrupt je vlastne obdoba kroku reaktívneho plánu. Navesenie globálneho interruptu je zasadenie nového stromu do lesa top-level správaní. Interrupty sa v jazyku E navesujú nasledovne:

```
localHook(podmienka, priorita, akcia, id)
globHook(podmienka, priorita, akcia, id)
```

Prvý riadok je navesenie lokálneho interruptu, druhý globálneho. `podmienka` a `akcia` sú b-skripty, `priorita` je aritmetický výraz, `id` je výstupný parameter, do ktorého je dosadený jednoznačný identifikátor interruptu. Skript podmienky nemôže volať atomické inštrukcie. Ak uspeje, je spustený skript akcie.

Oberanie jahôd z kapitoly 3.2 v E zapíšeme nasledovne.

```
oberajJahody
$-
    return 0.
:-
    localHook(vZahradeNieSuJahody, 6, COMMIT, id6),
    localHook(masPlnyKosik, 5, odnesDoKuchyne, id5),
    localHook({masKosik AND vZahrade}, 4, zoberJahodu, id4),
    localHook(masKosik, 3, pridDoZahrady, id3),
    localHook(vidisPrazdnyKosik, 2, vezmiKosik, id2),
    localHook({ true }, 1, hladajPrazdnyKosik, id1),
    /* tu ešte nejaký krátky začiatok */
    FAIL.
```

Text uzavretý v `/* text */` je komentár. Za sekvenciou znakov `$-` nasleduje tzv. *úžitková funkcia.* Slúži na výber medzi viacerými *variantami* skriptu. To nás teraz nemusí zaujímať. Za sekvenciou znakov `:-` nasleduje telo skriptu, ukončené je znakom `.` (bodkou). `COMMIT` je príkaz na okamžité úspešné ukončenie skriptu, `FAIL` na okamžité neúspešné ukončenie. Keď sa interpret dostane na koniec skriptu (znak `.`), skript je úspešne ukončený. Ak práve nie je splnená podmienka žiadneho naveseného interruptu, pokračuje sa vo vykonávaní skriptu. Na príkaz `FAIL` sa teda dostane len v prípade, že žiadny interrupt nemá splnenú podmienku. V takom prípade chceme, v súlade s našou predstavou reaktívneho plánu, aby oberanie jahôd zlyhalo. Presne o to sa príkaz `FAIL` postará.

Tento skript by nám, bohužiaľ, nefungoval. Slúži skôr pre predstavu, ako asi reaktívny plán v jazyku E zapísať. Interpret podmienky interruptov vyhodnocuje až po atomickej inštrukcii alebo ukončení nejakého podstromu (keď je potrebné vybrať ďalší podstrom na vykonávanie). Tento skript by po spustení navesil interrupty, nevykonal žiadnu atomickú inštrukciu a zlyhal. Na testovanie podmienok interruptov by sa vôbec nedostalo. Potrebujeme po navesení interruptov ešte „niečo málo“ vykonať, aby sme poslali atomickú inštrukciu a podmienky interruptov začali byť vyhodnocované. Aby sme zachovali sémantiku reaktívneho plánu, mali by sme pre každý navesený interrupt

```
localHook(podmienka, priorita, akcia, id)
```

pridať príkaz

```
if podmienka then akcia fi.
```

Vlastne, ešte stále to nestačí. Podmienka interruptu bude kontrolovaná aj v prípade, že akcia interruptu už beží a prípadne akciu spustí znova. Strom skriptov by sa nám teda rozvetvoval o rovnaké podstromy, kým by bola podmienka splnená. Našťastie jazyk E poskytuje aj prostriedok, ako interrupt dočasne blokovať a znova ho povoliť. Práve na to využijeme identifikátor interruptu. Tesne pred spustením akcie by sme mali interrupt zablokovať a po dokončení ho opäť povoliť. Správne by sme teda mali písať

```
localHook(
    podmienka,
    priorita,
    { block(id), akcia, enable(id) },
    id
).
```

6.2 Rozvrhovač

Už sme naznačili, že top-level správanie v jazyku E pridáme navesením globálneho interruptu. Napriek tomu sa tento spôsob v užívateľských skriptoch nepoužíva. Používa sa akási nadstavba, knižnica *rozvrhovača*.

V každom top-level správaní pridanom pomocou globálneho interruptu by sa opakoval nejaký kód, ktorý by bolo treba vykonať pre všetky top-level správania. Preto sa vykoná v rozvrhovači. Príkladom takéhoto kódu je nám už známe zablokovanie interruptu na začiatku a jeho uvoľnenie na konci akcie. Iným príkladom opakovaného kódu je spustenie správania od začiatku, ak bolo prerušené. To sme diskutovali v 3.5.iii. Interpret jazyka E reštart správania po prerušení nevykoná, poskytuje ale pre tento účel príkaz `RERUN`, ktorým to v prípade potreby môžeme urobiť sami.

Na úrovni rozvrhovača sa ďalej správania rozlišujú na *denné úlohy* a *životné funkcie*. Intuitívne asi rozlíšime, že pre správania ako okopávanie použijeme dennú úlohu a pre jedenie životnú funkciu. Hlavný rozdiel medzi dennou úlohou a životnou funkciou je, že denná úloha je spustená iba raz za deň. Keď je raz dokončená alebo zlyhá, je znovu spustená až ďalší deň. Životná funkcia môže byť spustená ľubovoľnekrát za deň, pokiaľ je splnená jej podmienka.

Pridanie dennej úlohy alebo životnej funkcie je rovnako jednoduché ako navesenie interruptu. Napríklad, životnú funkciu pre jedenie pridáme nasledujúcim volaním:

```
rozvrhovacRegistrujZivotniFunkci("najedzSa","masHlad",20,id).
```

masHlad je skript podmienky, po jeho úspechu sa spustí skript najedzSa. Tretím argumentom je priorita a posledným je premenná, do ktorej bude uložený identifikátor zaregistrovanej životnej funkcie.

Keď už užívateľ (autor b-skriptov) používa na pridávanie správaní rozvrhovač, mohli by sme prechodné správania a odloženie správania implementovať v ňom. Užívateľovi sa rozhranie príliš nezmení a bez prídavnej námahy môže modelovať vierohodnejších agentov.

Presne to sme urobili a výsledkom je nová verzia rozvrhovača, plne spätne kompatibilná s pôvodnou. Užívateľ teda môže používať rozvrhovač tak, ako doteraz, a v prípade záujmu má k dispozícii nové rozhranie pre prácu s pridanými rozšíreniami. Pridanie životnej funkcie, ktorá bude špecifikovať toleranciu a udávať odhad času potrebného na dokončenie, uskutočníme volaním

```
rozvrhovacRegistrujZivotniFunkci(
    "najedzSa",
    "masHlad",
    20,
    "odhadJedenia",
    idJedenia
).
```

Podmienka masHlad by mal byť jednoparametrický skript, ktorý v prípade úspechu do parametru dosadí svoju toleranciu. Ak však použijeme skript bez parametrov, bude to fungovať tiež a tolerancia bude považovaná za nulovú. odhadJedenia by mal byť tiež jednoparametrický skript, ktorý pri zavolaní vypočíta a do parametru zapíše odhad času potrebného na dokončenie akcie najedzSa.

Keď chceme, aby ent medzi zalievaním a jedením vykonal prechodné správanie, ktoré sa postará o vypolievanie vody, ktorú má ent práve v krhle, stačí po registrácii zalievania a jedenia zavolať

```
rozvrhovacRegistrujTransition(
   idZalievania, idJedenia, "vyprazdniKrhlu").
```

Celý mechanizmus spúšťania správania, ako je diskutovaný v časti 5.2, je implementovaný v tele globálnych interruptov, ktoré sú pre registrované správania navesené. Pre každú registrovanú úlohu alebo životnú funkciu je navesený globálny interrupt, ktorého podmienka je takmer totožná s tou zadanou pri registrácii. Na začiatku akcie interuptu potom prebehne rozhodovanie, či nemá byť správanie odložené a potom, keď naozaj chce bežať, vykoná sa prípadné prechodné správanie (ak bolo pre danú dvojicu registrované). Pre toto rozhodovanie sú potrebné informácie o tom, ktoré správanie práve beží, aká je jeho odhadová funkcia (skript), ktoré správanie chce bežať, aká je jeho tolerancia apod. Ukladanie takýchto informácií a ich

predávanie medzi rôznymi správaniami je realizované cez entovu pamäť a globálne premenné. Vzhľadom na to, že podmienka globálneho interruptu, ktorá sa testuje, je (takmer) nezmenená, réžia spojená s rozhodovaním o odložení a o vykonaní prechodného správania je až po splnení podmienky a len na začiatku správania. Počas plynulého behu správania žiadna ďalšia réžia nie je. Na samotnej simulácii spomalenie nie je pozorovateľné.

Pre odloženie správania jazyk E poskytuje prostriedok na uspanie skriptu a jeho prebudenie v daný čas a/alebo na platnosť nejakej podmienky. Bohužiaľ, prebudenie skriptu na platnosť podmienky nefungovalo správne. Bolo to teda treba do interpretu doprogramovať. Vyskytli sa aj ďalšie komplikácie s interpretom, ktoré bolo treba riešiť. Popri tom sa podarilo urobiť nejaké optimalizácie dátových štruktúr, čo viedlo k znateľnému zrýchleniu simulácie. Implementácia rozšírení mala teda na Projekt ENTi aj tento nepriamy pozitívny dopad.

Kompletný popis rozhrania pôvodnej aj novej verzie rozvrhovača, detaily jeho implementácie a zmienka o problémoch s interpretom sú v prílohe A.

6.3 Ukážka správania

V tejto časti opíšeme správanie enta, ktorý využíva rozšírenia rozvrhovača o prechodné správania a odloženie správania. Budeme sledovať nasledujúce správania, uvedené vo vzostupnom poradí podľa priority: *zalievanie*, *jedenie*, *odskočenie si na záchod* a *pitie*.

Ráno ent začne zalievať záhony. O chvíľu potrebuje ísť na záchod. Pretože zalievanie bude trvať ešte dlho, preruší ho. Vykoná prechodné správanie, ktorým je v tomto prípade vypolievanie vody z krhly a odloženie krhly. (Ent bez použitia rozšírení by kľudne odišiel na záchod s plnou krhlou.) Odskočí si na záchod a keď sa vráti, pokračuje v zalievaní. Časom dostane hlad, ale myslí si, že ešte vydrží, kým dokončí zalievanie. Pokračuje teda v zalievaní bez prerušenia. (Pôvodný ent by sa teraz odišiel najesť. Samozrejme, s krhlou.) Ako čas plynie, potrebuje si znovu odskočiť. Nie ale tak akútne, aby zalievanie nemohol dokončiť. Povie si, že ešte vydrží. O chvíľu sa ozve smäd, ale len malý, a tak ent stále pokračuje v zalievaní. („Starý“ ent by už zalievanie ďalšie dva razy prerušil.) Časom hlad stúpa a ent si povie, že si predsa len urobí prestávku. Prestávku si síce urobil kvôli hladu, to ale ešte neznamená, že sa pôjde najskôr najesť. Má už tri činnosti, ktoré chce urobiť počas prestávky. Najdôležitejšie z nich je pitie. Ale smäd ešte nie je veľký, preto uprednostní jedenie pred pitím (tzv. „odmena“ z odstavca 4.2.i.γ)). Lenže ent tiež potrebuje ísť na záchod a už nevydrží, kým sa naje. S pitím môže počkať, kým si odskočí. Pôjde teda najskôr na záchod. Predtým ešte vykoná prechodné správanie zo zalievania na odskočenie si. Potom sa pôjde napiť. (Jedenie tu už druhýkrát odmenu nedostane. Pozor, oproti príkladu z odstavca 4.2.i.γ), tu sú priority pitia a odskočenia si obrátene.) Na záver prestávky sa naje a vráti sa dokončiť zalievanie.

Dlhšiu a podrobnejšie popísanú ukážku možno nájsť v prílohe A.

7. Príbuzné práce

7.1 POSH reaktívne plány

POSH (Parallel-rooted, Ordered, Slip-stack Hierarchical) reaktívne plány sú metódou výberu akcie, ktorú navrhla Joana J. Bryson pre svoju BOD (Behavior-Oriented Design) architektúru na tvorbu komplexných agentov ([4]).

Našim reaktívnym plánom budeme v tejto kapitole pre odlíšenie hovoriť *bežné reaktívne plány (BRP).*

POSH reaktívne plány vychádzajú z bežných reaktívnych plánov, prinášajú však niekoľko rozšírení.

V HRP sú jednotkami správania bežný reaktívny plán a sekvencia akcií. Nasledujúca tabuľka zachycuje korešpondenciu medzi prvkami HRP a POSH.

HRP	**POSH**
sekvencia atomických akcií	action pattern
bežný reaktívny plán	competence drive collection

Tab. 7.1 Korešpondencia medzi prvkami HRP a POSH

Action pattern môže oproti jednoduchej sekvencii atomických akcií obsahovať paralelné alebo neusporiadané akcie. Avšak sama autorka uviedla ([4]), že zatiaľ nemala dôvod tieto nové vlastnosti využiť.

Competence (pre lepšie skloňovanie budeme používať tiež slovo *kompetencia*) je rozšírenie bežného reaktívneho plánu o maximálny počet pokusov, koľkokrát sa môže skúsiť vykonať krok plánu pri jeho zlyhávaní. Kroky kompetencie sú teda štvorice $(\pi, \varrho, \alpha, \eta)$, kde π, ϱ, α sú v poradí priorita, podmienka, akcia, ako v bežnom reaktívnom pláne. η je voliteľný maximálny počet pokusov na vykonanie kroku. Záporné η znamená neobmedzený počet.

Drive collection je tiež rozšírenie bežného reaktívneho plánu, mienené ako koreň hierarchie. Prvky drive collection nazývame *drive elementy.* Drive element je pätica $(\pi, \varrho, \alpha, A, \nu)$. π a ϱ sú priorita a podmienka ako v BRP. A je koreň stromu kompetencií. α je na začiatku rovné A. Keď je splnená podmienka ϱ, spustí sa α. Ak je α kompetencia a spustí nejaký svoj krok, do α je dosadená akcia spusteného kroku. Ak α zlyhá, dosadí sa do α znovu A. ν je maximálna frekvencia, pri ktorej môže byť daný drive element navštevovaný.

V každom cykle výberu akcie sa vyberá jeden drive element z drive collection. V ňom sa potom nezačína výber akcie od koreňa (A), ale od α. Tým sa ušetrí výpočtový čas, lebo pre výber akcie je potrebné testovať menej podmienok. Testovanie menej podmienok ale môže viesť k zníženiu reaktívnosti, ak nastala zmena v platnosti podmienok niekde nad α. Ušetrením výpočtového času je to však trochu kompenzované. Navyše, zásadné veci by mali byť na najvyššej úrovni (v drive collection), ktorá je vyhodnocovaná v každom cykle.

Ďalšou vlastnosťou drive collection je, že umožňuje, aby v hierarchii kompetencií boli cykly. To vďaka tomu, že pre každú hierarciu si vždy pamätáme iba jeden aktívny uzol (totiž α) a nie cestu, ktorej nekonečné predlžovanie by u bežných reaktívnych plánov viedlo k pretečeniu zásobníka.

V α máme uložený stav prerušeného správania. U bežných reaktívnych plánov sme si pre každé prerušené správanie potrebovali pamätať cestu, tu stačí jeden uzol.

Nijaké prostriedky, ktoré by uľahčovali vyjadrenie prechodných správaní alebo odloženie správania, POSH neprináša. POSH reaktívne plány bývajú používané napríklad na návrh správania robotov alebo zvieracích agentov (opíc), teda v doménach, kde potreba prechodných správaní nie je až taká výrazná ako pri simulovaní ľudí.

7.2 Expressivator

Expressivator je architektúra na modelovanie agentov, s dôrazom na ich správne pochopenie pozorovateľom. Možno ho zaradiť medzi „behaviour-based“ architektúry, kde je správanie postupne rozkladané na jednoduchšie, navzájom nezávislé činnosti. Klasické behaviour-based prístupy sú však orientované na riešenie problémov. Prípadná porozumiteľnosť agentovho správania býva riešená len ako dodatok do hotového systému a väčšinou končí neuspokojivými výsledkami.

P. Sengers, autorka Expressivatoru, argumentuje ([6]), že mnohé AI systémy, bez ohľadu na to, aké sú dobré v riešení problémov, môžu byť nepoužiteľné, ak správanie agentov je pre užívateľa nezrozumiteľné. U zrozumiteľného agenta by mal užívateľ získať jasnú predstavu o tom, *čo* agent robí a *prečo* to robí. V Expressivatore je preto správanie navrhované tak, aby bolo zrozumiteľné hneď od začiatku. Správania sú rozkladané na podsprávania podľa toho, ako majú byť pochopené pozorovateľom, nie podľa toho, čo agent v skutočnosti (vnútorne) robí.

Základné prvky tejto architektúry zhruba odpovedajú prvkom v HRP.

HRP	Expressivator
sekvencia atomických akcií	sign
bežný reaktívny plán	low-level signifier high-level signifier

Tab. 7.2 Korešpondencia medzi prvkami HRP a Expressivatoru

Sign je sekvencia fyzických akcií, ktorá má byť užívateľom interpretovaná určitým spôsobom.

Low-level signifier odpovedá nejakému jednoduchému správaniu. Je zložené zo signs, fyzických akcií a mentálnych akcií. Jeho úlohou je správne a zrozumiteľne vyjadrovať, čo agent momentálne robí (akú krátkodobú činnosť). Keď totiž nedokážeme rozoznať, čo robí, ťažko môžeme uvažovať o tom, prečo to robí.

High-level signifier je vyššie správanie poskladané z predchádzajúcich prvkov. Má za úlohu vyjadrovať agentove vyššie, dlhodobejšie aktivity.

Systém si navyše vedie záznam o tom, čo bolo vyjadrené a čo si teda pozorovateľ asi myslí. Na to slúži tzv. *sign management.* Na základe aktuálnej užívateľovej interpretácie potom môže závisieť ďalšie správanie.

Veľký význam P. Sengers prikladá prechodným správaniam. Tie používa hlavne na to, aby jasne vyjadrili, že dochádza k zmene hlavných správaní a zároveň vysvetlili dôvod pre túto zmenu. Kým predchádzajúce elementy sa starajú hlavne o jasné vyjadrenie aktivít, prechodné správania vyjadrujú agentovo uvažovanie. Sú hlavným „rozprávačom“ príbehu.

Prechodné správania sú implementované v dvoch častiach: *spúšťač* a *démon.* Obe sú plnohodnotnými správaniami. Spúšťač monitoruje bežiace správania a zmeny vo svete (cez senzorické vstupy). Zisťovanie informácii o bežiacich správaniach umožňujú tzv. *meta-level controls.* Keď spúšťač usúdi, že je dôvod zmeniť správanie, zapíše do pamäte príznak, že je požadovaná zmena správania, správanie, na ktoré sa má prejsť a dôvod zmeny. Démon čaká na objavenie takejto informácie v pamäti. Potom vyberie vhodné vyjadrenie dôvodu, ktoré vykoná a spustí nové správanie. Správne vyjadrenie môže vyžadovať zmeny v nasledujúcom správaní. Meniť a spúšťať iné správania je démonovi umožnené tiež pomocou meta-level controls. Meta-level controls sú nástrojom akejsi introspekcie. Umožňujú správaniam získavať informácie o iných správaniach a ovplyvniť ich. Sú teda porušením behaviour-based princípu, pretože správania nie sú úplne nezávislé.

Povšimnime si rozdiel medzi našimi prechodnými správaniami a tými, o ktorých hovorí Sengers.

My prechodné správania používame na hladký prechod medzi správaniami. O zmene správania je rozhodnuté niekde inde. Hladký prechod má zvýšiť agentovu vierohodnosť. Prechodné správania v Expressivatore sú prostriedkom, ako vyjadriť agentovo uvažovanie a pocity, a tým aj dôvody na zmenu správania. Je pre ne implementovaný zvláštny mechanizmus v podložnom systéme, totiž meta-level controls. Prechodné správania bežia na pozadí a samé monitorujú potrebu prechodu a potom ho vykonajú. Naše prechodné správania nebežia stále, len keď sú spustené našim rozšíreným mechanizmom HRP. V Expressivatore majú prechodné správania významnejšie postavenie a väčšie „právomoci“. Môžu ovplyvňovať iné správania. V našom modele si samotné prechodné správanie nie je vedomé iných správaní. Má si len vykonať svoju prácu, keď bude zavolané. Do iných správaní vôbec nezasahuje.

8. Záver

Ukázali sme, že pre vierohodnosť umelej bytosti je potrebné, aby medzi hlavnými správaniami prebiehali správania prechodné a aby správania, ktoré nebudú trvať dlho, boli prevedené v celku, bez prerušenia.

Nekompromisné prepínanie medzi správaniami v hierarchickom reaktívnom plánovaní takéto možnosti neposkytovalo. Model HRP sme preto rozšírili. Naše rozšírenie je myšlienkovo jednoduché a, ako sme ukázali v kap. 6, pomerne efektívne implementovateľné. Vidíme, že vysporiadať sa s rozoberanými problémami biologickej vierohodnosti ide bez citeľnej straty výkonu.

Brom poukazuje ([3]) aj na ďalšie nedostatky HRP, ktoré by v záujme vierohodnosti mali byť vyriešené. Naše rozšírenie model HRP nijako dramaticky nenabúralo a mohol by byť preto braný ako východiskový model pre ďalšie rozširovanie.

A Prechodné správania a odloženie správania v Projekte ENTi

A.1 Kto sú Enti?

Enti sú virtuálne postavičky žijúce vo virtuálnom svete podobnom tomu nášmu, ktoré sa snažia napodobniť ľudské správanie.

Projekt Enti vznikol ako študentský softwarový projekt na Matematicko-fyzikálnej fakulte Univerzity Karlovej v Prahe.

Pretože v klasických programovacích jazykoch sa správanie umelých bytostí programuje obtiažne, bol špeciálne pre tento projekt navrhnutý jazyk E, v ktorom sa správanie Entov programuje.

A.2 Prechodné správania (transition behaviors)

Virtuálna bytosť má väčšinou niekoľko hlavných (*top-level*) cieľov, ktoré sa snaží dosiahnuť. K dispozícii má sadu jednoduchých (atomických) akcií, ktoré môže vykonávať, napr. „urob krok“, „otoč sa“, „uchop predmet“. Zásadná otázka, na ktorú potrebuje nájsť odpoveď, znie: *„Čo mám práve teraz robiť?“*

Dať odpoveď rovno na úrovni atomických akcií je príliš zložité. Obecne je cieľ možné splniť viacerými spôsobmi. Činnosti, ktorá vedie k splneniu nejakého cieľa, hovoríme správanie. K splneniu cieľa teda môže viesť viac správaní. Je vhodné si správanie rozdeliť na splnenie dielčich podcieľov. Pre úspešné ukončenie správania potom môže byť potrebné splnenie všetkých podcieľov alebo len niektorých podcieľov, pritom môže i nemusí záležať na poradí plnenia týchto podcieľov.

Príkladom cieľa môže byť „nezomrieť od hladu“. Správania, ktoré dokážu splniť tento cieľ, môžu byť „najedz sa v kuchyni“, „najedz sa v reštaurácii“. Prvé správanie zjemníme na splnenie podcieľov „prísť do kuchyne“, „nájsť jedlo v chladničke“, „zjesť nájdené jedlo“. Tu stačí, ak sa podarí splniť posledný podcieľ. (Ak sme práve v kuchyni, nemusíme do nej chodiť.) K tomu ale možu byť potrebné tie predchádzajúce, a to ešte v určitom poradí.

Postupne sa dostaneme až na úroveň, kde dané správanie vieme popísať atomickými akciami, ktoré bytosť dokáže vykonať rovno.

Teraz už odpoveď na otázku vieme dať tak, že si najskôr vyberieme top-level cieľ, ktorý chceme napĺňať práve teraz. V rámci neho potom postupne vyberáme podciele a im odpovedajúce podsprávania až atomické akcie.

Otázkou zostáva, ako vybrať top-level cieľ a ako vybrať podcieľ v rámci správania. To ale pre nás zatiaľ nie je dôležité.

Dôležité je, že v jednom okamihu sa bytosť môže zaoberať iba jedným top-level cieľom. (To je trochu obmedzujúce. Jendiné konanie by niekedy mohlo napomôcť splneniu rôznych cieľov. Ale ani v skutočnom svete to takto príliš často nebýva. Väčšinou sa v jednom momente zaoberáme len jedným cieľom.)

Takže prechodné správania…

Predstavme si, že naša postavička je na záhrade a okopáva záhony. Súčasťou okopávania je, že keď skončí, uprace motyku a umyje si ruky. Ale preto, že je okopávanie činnosť zdĺhavá a ťažká, postavička vyhladne skôr, ako ju stihne dokončiť. Usúdi, že teraz je dôležitejšie najesť sa. Rozhodne sa, že sa naje v kuchyni. Čo asi teraz urobí? Predsa pôjde do kuchyne. S motykou v ruke. To, zjavne, nie je úplne správne. Čo s tým? Môžeme k správniu „najedz sa v kuchyni" pridať podsprávanie, ktoré zariadi, že ak postavička drží motyku, tak ju najskôr položí a umyje si ruky. A čo ak sa rozhodne sa najesť, práve keď vysáva, sprchuje sa, telefonuje… Máme pre každé toto správanie pridať do „najedz sa v kuchyni" nejakú časť, ktorá to dá najskôr doporiadku? A čo do ostatných správaní? Veď, keď chce ísť do kina, mala by najskôr dotelefonovať, ak práve telefonuje. Takto by sa popis správaní rýchlo stal neprehľadný, až neudržiavateľný. Navyše, správanie „zájdi do kina" mohol navrhnúť niekto iný a my mu do toho nechceme alebo nemôžeme zasahovať.

Hodila by sa nám možnosť, ako dať veci doporiadku, keď sa prechádza z jedného správania na druhé a prvé ešte nebolo úplne dokončené. Teda možnosť vykonať nejaké *prechodné správanie.*

A.3 Odložené spustenie správania

V prípade, že postavička okopáva a ešte má toho veľa, je vhodné okopávanie prerušiť, ísť sa najesť a potom sa k okopávaniu znova vrátiť. Medzitým sa vykoná krátke prechodné správanie, odloženie motyky. Keď sa ale sprchuje a dostane hlad, nie je praktické, aby sprchovanie prerušila a po najedení k nemu opäť vrátila. Hlad pravdepodobne nie je taký akútny, aby zomrela od hladu, ak sa nepôjde najesť okamžite. Človek tiež nedostane hlad odrazu, ale postupne sa zvyšuje. Ak je hlad veľký už pred sprchovaním, najskôr sa naje. Ak nie, naje sa až potom. Takisto, ak už stačí okopať len jeden záhon, tak je rozumné najskôr dokončiť okopávanie a až potom sa ísť najesť. Lenže v počítači sa to, ako presne je akútne ísť sa najesť, zapisuje ťažko.

My dáme správaniam, ktoré majú záujem bežať, možnosť, aby špecifikovali, koľko ešte môžu počkať. A správaniam, ktoré práve bežia, dáme možnosť odhadnúť, koľko času ešte potrebujú na dokončenie. Potom, keď prerušujúce správanie môže počkať toľko, koľko prerušované odhaduje, že potrebuje, odložíme spustenie prerušujúceho do doby, kým prerušované neskončí. Ak ale neskončí ani dovtedy, keď už prerušujúce nemôže počkať, tak bude aj tak prerušené. Problém, ako presne určiť nutnosť behu jednotlivých správaní v konkrétnom okamihu, sme si týmto nezjednodušili. Previedli sme ho totiž na problém, ako správne odhadnúť čas potrebný na dokončenie a čas, ktorý je správanie ochotné počkať. Ľuďom ako autorom správaní umelých bytostí sa však v pojmoch času uvažuje pomerne dobre. Ľahšie je nám špecifikovať, že na každý neokopaný záhon potrebujeme 15 minút, ako napr. to, že s každým ďalším okopaným záhonom stúpne nutnosť dokončiť okopávanie o 8.

A.4 Niečo málo o jazyku E

V prvom rade, podrobný popis jazyka E je možné nájsť v *Príručke autora sveta a skriptov*, ktorá je súčasťou dokumentácie Projektu Enti a dostupná zo stránky projektu (`http://ufal.mff.cuni.cz/ bojar/enti/`).

E je jazyk navrhnutý na programovanie správania umelých bytostí. Bol vytvorený špeciálne pre Projekt Enti. Je to interpretovaný jazyk, syntaxou podobný Prologu (a to nie je náhoda).

Obdobou procedúr (funkcií, metód) z klasických jazykov alebo klauzúl z Prologu je tzv. *b-script (behavioral script)*. Z názvu vyplýva, že je určený na popis správania. b-script má 2 možné výsledky: úspech a neúspech (zlyhanie). Možno si ho teda predstaviť ako booleovskú funkciu, ktorá vráti `true` alebo `false`. V tele skriptu je možné volať iné skripty, pracovať s pamäťou enta, s premennými, volať atomické inštrukcie. Volanie skriptu aj volanie atomickej inštrukcie má tvar predikátu (ako v Prologu). Atomická inštrukcia, rovnako ako b-script, buď uspeje, alebo zlyhá (urobiť krok sa buď podarí, alebo nepodarí). Podobne ako v bežných jazykoch, E obsahuje podmienky a cykly.

Obdobou funkcie `main()` z C alebo Javy je b-scipt, ktorý je deklarovaný ako top-level. Tento skript začne interpret pri spustení simulácie vykonávať a jeho ukončením končí simulácia daného enta.

S doteraz uvedenými prostriedkami by programovanie správania prebiehalo podobne ako v klasických jazykoch a zvláštny jazyk by nebol vôbec potrebný. Beh enta by predstavoval beh top-level skriptu, ktorý by prípadne v cykloch a podmienkach volal iné skripty.

Vierohodné umelé bytosti potrebujú reagovať na zmeny prostredia a na svoje vnútorné zmeny (napr. potreba ísť na záchod). A to, samozrejme, vtedy, keď tieto zmeny nastanú, teda asynchronne. Klasickým prístupom to znamená testovať množinu podmienok a keď nejaká platí, vykonať patričnú akciu (som hladný(á) $\Rightarrow$ idem sa najesť). Tieto podmienky sa však musia testovať takpovediac stále. Nie, napríklad, len na začiatku nejakého cyklu. Pokúsiť sa to zachytiť nejakými `if-then` príkazmi by bolo neprehľadné, ak nie úplne nerealizovateľné.

Interrupty

Kľúčovým prvkom jazyka E oproti iným jazykom sú tzv. *interrupty*. Interrupt je prostriedok, ako interpretu povedať, nech podmienku testuje za nás a keď bude splnená, spustiť nejakú akciu - skript. Pre priblíženie si to môžeme predstaviť ako ošetrenie signálu poslaného procesu v unixových operačných systémoch. Potom splnenie danej podmienky odpovedá príchodu signálu. Po splnení podmienky sa teda začne vykonávať ošetrujúci skript.

Podobne, ako je možné blokovať príchod signálu, je možné blokovať spustenie ošetrujúceho skriptu. To je obzvlášť užitočné, ak už jedna inštancia ošetrujúceho skriptu beží a spúšťajúca podmienka stále platí.

Zaregistrovaniu interruptu (tj. oznámeniu interpretu, že chceme, aby nám kontroloval nejakú podmienku) hovoríme navesenie interruptu.

Niektoré podmienky potrebujeme kontrolovať skutočne stále (napr. potrebu ísť na záchod), niektoré len keď beží určité správanie (kontrolovať, či nám jedlo na sporáku nekypí, stačí počas varenia; ošetrenie by bolo stlmiť sporák).

Rozlišuje sa preto medzi interruptmi globálnymi a lokálnymi. Podmienka globálneho interruptu je kontrolovaná stále, bez ohľadu na to, ktorým skriptom bol navesený a či ten skript práve beží. Podmienka lokálneho interruptu je kontrolovaná len počas behu navesujúceho skriptu. (Toto nie je úplne presné, v skutočnosti sa interrupty môžu „dediť", ale pre nás to takto stačí.)

Ošetrujúci skript globálneho interruptu má po spustení podobné postavenie ako top-level skript, s tým rozdielom, že jeho ukončenie nespôsobí koniec behu enta. Budeme ho teda označovať tiež ako top-level.

Top-level skripty nemajú vlastníka. Pre ostatné je vlastníkom skript, ktorý ich zavolal. (V čase písania skriptu hovoriť o jeho vlastníkovi nemá zmysel. Zmysel to má až za behu, keď ho nejaký iný skript zavolá alebo je spustený ako top-level.)

Rovnako, globálne interrupty nemajú vlastníka, vlastníkom lokálneho interruptu je skript, ktorý ho navesil. Po spustení ošetrujúceho skriptu interruptu tento skript zdedí vlastníka od interruptu.

Zvláštnym typom interruptu je tzv. sleep interrupt. Sleep interrupt nemá ošetrujúcu akciu. Namiesto toho po splnení podmienky zobudí skript, ktorý ho navesil (v prípade, že sa skript po navesení sleep interruptu uspal). Sleep interrupt má postavenie ako globálny interrupt, má však vlastníka.

Priority

Podmienok interruptov (globálnych i lokálnych) môže platiť viac naraz, a teda je viac možností, ktorá bude ošetrená ako prvá. Na to, aby sme mohli určiť dôležitosť ošetrenia tej ktorej podmienky, sú zavedené priority. Každý interrupt a každý aktívny skript má prioritu. (Aktívny skript je taký, ktorý „má záujem" bežať.) Prioritu interruptu zadávame pri jeho navesovaní a určuje prioritu, ktorú dostane jeho ošetrujúci skript po spustení. Podľa priority aktívnych skriptov sa rozhoduje, ktorý z nich práve pobeží. A to tak, že sa najskôr vyberie top-level skript s najvyššou prioritou. Pokiaľ nebol navesený žiadny globálny interrupt, alebo podmienka žiadneho nebola splnená, existuje len jeden top-level skript — ten deklarovaný ako top-level. V rámci zvoleného top-level skriptu sa potom podľa priority vyberie ten aktívny potomok, ktorý pobeží. Opäť, pokiaľ nebola splnená podmienka lokálneho interruptu, máme len jedného aktívneho potomka (pokiaľ s každým aktívnym skriptom nerátame za aktívnych aj jeho predkov).

Top-level skript má na začiatku prioritu 0. Volaný skript zdedí prioritu od volajúceho.

Vidíme, že bez interruptov by priority nemali žiadny zmysel.

Priorita interruptu (zadávaná pri navesovaní) je číslo z aritmetiky jazyka E. Priorita skriptu má tiež v každom okamihu konkrétnu hodnotu — číslo,

môže sa však meniť v čase. Skript si môže zmeniť prioritu volaním príkazov `setGlobalPriority` a `setLocalPriority`. Prvý z nich nastaví prioritu top-level skriptu, ktorého je volajúci skript potomkom. Má to teda vplyv na výber top-level skriptu, čiže na to, ktorý strom skriptov pobeží. Druhý príkaz nastaví prioritu volajúceho skriptu v rámci jedného stromu.

Na top-level úrovni je dokonca možné nastaviť špeciálne lichobežníkové priority.

Viac o prioritách možno nájsť v príručke.

A.5 Výber akcií u Entov

Teraz si už vieme predstaviť, ako správanie enta v Éčku naprogramovať: každému cieľu priradíme podmienku, kedy môže byť napĺňaný, a prioritu, ktorá určuje jeho dôležitosť. Pre každý cieľ potom navesíme jeden globálny intyerrupt s danou podmienkou a prioritou a ako ošetrujúcu akciu uvedieme správanie (b-script), ktoré vedie k splneniu tohto cieľa.

V každom správaní sa pritom budú niektoré veci opakovať.

Podmienkou pre spustenie správania „najedz sa“ bude „som hladný“. Navesíme teda globálny interrupt s podmienkou „som hladný“ a ošetrujúcim správaním „najedz sa“. Po splnení podmienky sa spustí skript pre jedenie. Ent sa teda vydá do kuchyne. Podmienka ale stále platí (stále je hladný) a tak sa ošetrujúci skript spustí znova. (Keby sme teraz vedeli, ako svet entov a interpret Éčka fungujú, povedali by sme, že nová inštancia ošetrujúceho skriptu sa spustí po vykonaní prvej atomickej inštrukcie toho prvého, tj. v ďalšom kole.) Budú potom dva aktívne skripty najedz sa s rovnakou prioritou, a tak sa medzi nimi vyberie náhodne. Ale hneď nato sa spustí ďalší, a ďalší … Až kým ent prestane byť hladný. Aby sme tomuto predišli, budeme na začiatku každého ošetrujúceho skriptu jeho spúšťací interrupt blokovať a na konci ho znova povoliť.

Ďalšou vecou, ktorá sa bude často opakovať, je nastavovanie priority. Interrupt môžeme navesiť iba s konštantnou prioritou, top-level skripty však môžu mať aj prioritu lichobežníkovú (ktorú sme tu nerozoberali). Skript teda zdedí od interruptu konštantnú prioritu, ak ale chceme prioritu lichobežníkovú, musíme si ju nastaviť sami volaním setGlobalPriority.

Vďaka interruptom a prioritám sa jednotlivé činnosti môžu navzájom prerušovať. Keď dôjde k prerušeniu nejakej činnosti inou, po ukončení prerušujúcej sa ent opäť vráti k prerušenej. A to na miesto, kde predtým prestal. To však nemusí mať dobrý zmysel. Predstavme si, že ent polieva a práve chce zaliať záhon. Vtom je však činnosť zalievanie prerušená jedením. Až sa naje, vráti sa opäť k zalievaniu a to tam, kde prestal. Teda zavolá atomickú inštrukciu na zaliatie záhonu. To ale teraz nemá zmysel, lebo sa práve nachádza v kuchyni a nedrží krhlu. Jednoduchý spôsob, ako sa tomuto vyhnúť, je, že si správanie bude kontrolovať, či nebolo náhodou prerušené, a ak bolo, tak sa spustí od začiatku. V tomto prípade, ak ent nemá krhlu, tak si znova nejakú nájde, ak nie je v záhrade, tak do nej príde…

Aby sa tieto veci nemuseli opakovane písať v každom správaní, existuje knižnica rozvrhovača.

A.6 Rozvrhovač

Ako už bolo napísané, rozvrhovač sa stará o blokovanie interruptu počas behu jeho ošetrujúcej akcie, nastavenie priority (zvlášť lichobežníkovej), reštartovanie činnosti po prerušení. V našich skriptoch potom globálne interrupty nenavesujeme sami, ale použijeme skripty rozvrhovača, ktorý sa navyše postará o zmienené veci.

Na úrovni rozvrhovača rozlišujeme správania na tzv. denné úlohy (napr. okopávanie) a životné funkcie (napr. jedenie). Hlavný rozdiel medzi dennou úlohou a životnou funkciou je ten, že úloha sa po ukončení (úspešnom či neúspešnom) spustí znova až na ďalší deň. (Vo svete entov plynú dni.) Životná funkcia sa spustí ľubovoľnekrát za deň. Denné úlohy sa ďalej rozlišujú na jednorázové (spustia sa len v jeden deň, potom už nie) a pravidelné (sú spúšťané každý deň).

API rozvrhovača

Používame konvenciu z Prologu: vstupný parameter označujeme `+param` a výstupný `-param`.

```
/*(1)*/ rozvrhovacRegistrujUlohu(+uloha, +param,
                      +priorita, -UID)
/*(2)*/ rozvrhovacRegistrujUlohu(+uloha, +param,
                      +lStart, +lVyska, +lZlom, +lKonec, -UID)
/*(3)*/ rozvrhovacSpustUlohu(+uloha, +param,
                      +priorita, -UID)
/*(4)*/ rozvrhovacSpustUlohu(+uloha, +param,
                      +lStart, +lVyska, +lZlom, +lKonec, -UID)
/*(5)*/ rozvrhovacRegistrujZivotniFunkci(+akce, +podminka,
                      +priorita, -UID)
/*(6)*/ rozvrhovacRegistrujZivotniFunkci(+akce, +podminka,
                      +lStart, +lVyska, +lZlom, +lKonec, -UID)
/*(7)*/ rozvrhovacOdeberUlohu(+UID)
/*(8)*/ rozvrhovacAktualniUloha(-UID)
/*(9)*/ rozvrhovacNazevUlohy(+UID, -uloha)

/*(10)*/ rozvrhovacInit
/*(11)*/ rozvrhovacStart
```

(1) zaregistruje pravidelnú dennú úlohu s danou prioritou. Úlohe predá parameter `param`. V `UID` vráti jednoznačný identifikátor úlohy (použiteľný v iných volaniach rozvrhovača).

(2) je ako (1), ale s lichobežníkovou prioritou.

(3) a (4) sú ako (1) a (2), ale dané úlohy budú jednorázové.

(5) zaregistruje životnú funkciu s danou prioritou. Po splnení podmienky sa spustí akcia. Podmienka je skript bez parametrov. V `UID` vráti jednoznačný identifikátor.

(6) je obdobné (5), ale s lichobežníkovou prioritou.

(7) odoberie úlohu s daným identifikátorom.

(8) vráti identifikátor práve bežiacej úlohy.

(9) vráti názov úlohy s daným identifikátorom.

(10) inicializuje rozvrhovač. Nutné volať pred začiatkom práce s rozvrhovačom.

(11) spustí rozvrhovač. Registrované činnosti sa začnú spúšťať až po tomto volaní. Jednorázové úlohy sa spustia hneď. Ďalšie úlohy je možné registrovať aj potom, ale prejaví sa to až po ukončení všetkých predošlých (typicky až na ďalší deň).

A.7 Rozšírenie o prechodné správania a odložené spustenie

Nové prvky zavedieme do knižnice rozvrhovača, aby si ich autor skriptov nemusel programovať sám. Nad pôvodnou verziou rozvrhovača by sa to písalo obtiažne, možno až tak obtiažne, že by bolo ľahšie vzdať sa rozvrhovača a starať sa o všetko sám.

Hovorili sme, že správanie, ktoré má byť spustené, bude mať možnosť uviesť čas, ktorý ešte môže so začatím počkať. Tento čas budeme nazývať *toleranciou*. Tolerancia má niečo spoločné s podmienkou na spustenie. Ak napr. podmienka životnej funkcie „najedz sa" kontroluje úroveň hladu, čas ktorý ešte táto životná funkcia môže počkať, bude pravdepodobne tiež záležať na úrovni hladu. Navyše zisťovať toleranciu má zmysel len pri platnosti podmienky. Podmienkou životnej funkcie bol doteraz bezparametrický skript. My to zmeníme na skript s jedným výstupným parametrom, v ktorom v prípade úspechu vráti toleranciu. Denné úlohy však žiadnu podmienku nemali. Predpokladalo sa, že chcú bežať vždy, keď im to ich priorita dovolí. V novej verzii rozvrhovača budú mať podmienku aj denné úlohy. Tou môže byť kľudne skript, ktorý vždy uspeje. Ide nám hlavne o to, aby v parametri vrátil svoju toleranciu.

Ďalej sme chceli dať bežiacej úlohe možnosť udať odhad času, ktorý potrebuje na svoje dokončenie. Pre tento účel budeme pri registrovaní úloh uvádzať ďalší parameter — odhadový skript. Tým by mal byť skript, ktorý do svojho jediného parametru uloží odhad potrebného času.

Nové API rozvrhovača

S novým rozvrhovačom môžeme využiť nasledujúce nové volania. Pôvodné pritom zostali zachované a je možné použitie starých a nových kombinovať.

```
/*(12)*/ rozvrhovacRegistrujUlohu(+uloha, +podminka, +param,
          +priorita, +odhad, -UID)
/*(13)*/ rozvrhovacRegistrujUlohu(+uloha, +podminka, +param,
          +lStart, +lVyska, +lZlom, +lKonec, +odhad, -UID)
/*(14)*/ rozvrhovacSpustUlohu(+uloha, +podminka, +param,
          +priorita, +odhad, -UID)
/*(15)*/ rozvrhovacSpustUlohu(+uloha, +podminka, +param,
          +lStart, +lVyska, +lZlom, +lKonec, +odhad, -UID)
/*(16)*/ rozvrhovacRegistrujZivotniFunkci(+akce, +podminka,
```

```
          +priorita, +odhad, -UID)
/*(17)*/ rozvrhovacRegistrujZivotniFunkci(+akce, +podminka,
          +lStart, +lVyska, +lZlom, +lKonec, +odhad, -UID)
/*(18)*/ rozvrhovacRegistrujTransition(
          +fromUID, +toUID, +transition)
/*(19)*/ rozvrhovacOdeberTransition(+fromUID, +toUID)
```

Skriptom na registrovanie a spustenie úlohy (12-15) pribudli dva parametre.

Prvým z nich je podmienka. Ako už bolo povedané, mal by to byť jednoparametrický skript, ktorý svojím úspechom či neúspechom rozhodne, či má daná úloha bežať. V prípade úspechu v parametri vráti toleranciu. Avšak bude fungovať, aj keď to bude bezparametrický skript. V tomto prípade je to ekvivalentné tomu, ako keby vrátil toleranciu rovnú nule (čiže žiadna tolerancia).

Druhým novým parametrom je odhad. Tým by mal byť jednoparametrický skript, ktorý vo svojom parametri vráti odhad času potrebného na dokončenie. Pokiaľ neuspeje, je to akoby vrátil ∞ (teda akoby sme so žiadnym odhadom nepracovali). Vedľajším efektom tejto vlastnosti je, že ak nechceme programovať odhadový skript, ale z nového API chceme využiť to, že úloha môže mať podmienku (s toleranciou), stačí uviesť ako odhadový skript čokoľvek, čo zlyhá, napríklad voľnú premennú.

Skriptom pre registráciu životnej funkcie (16, 17) pribudol parameter odhad. Jeho význam je rovnaký, ako pri skriptoch pre úlohy. Rovnako, je uprednostňovaná podmienka s jedným parametrom vracajúca toleranciu, ale je možné použiť aj bezparametrickú podmienku.

Úplne nová je dvojica skriptov na registráciu a odobratie prechodného správania.

(18) pre dané 2 úlohy/životné funkcie zaregistruje skript `transition`, ktorý bude prechodným správaním medzi danými dvoma správaniami. Prechodné správanie je možné registrovať aj pre jednorázovú úlohu. Ak už pre danú (usporiadanú) dvojicu bolo prechodné správanie registrované, je nahradené novým. Ak bol zadaný nesprávny identifikátor činnosti, nič sa nevykoná a skript aj tak uspeje.

(19) zruší prípadné prechodné správanie predtým registrované pre danú dvojicu správaní.

A.8 Podrobnosti implementácie rozvrhovača

Rozvrhovač pre každú registrovanú alebo jednorázovo spustenú úlohu alebo životnú funkciu navesí globálny interrupt. Od navesenia interruptu budeme o činnosti hovoriť, že je naplánovaná. Informáciu o tom, ktoré činnosti sú momentálne naplánované, si rozvrhovač udržuje v zozname naplánovaných úloh.

Jednorázové úlohy sú naplánované hneď po volaní `rozvrhovacSpustUlohu`. U pravidelných činností je to zložitejšie. Prvýkrát sú naplánované po spustení rozvrhovača, tj. po zavolaní `rozvrhovacStart`. Naplánovanie

činností registrovaných po tomto volaní, ako aj opätovné naplánovanie činností, ktoré boli zo zoznamu naplánovaných úloh odstránené (viď nižšie), prebehne po najbližšom vyprázdnení zoznamu naplánovaných úloh.

Pre každú (pravidelnú i jednorázovú) činnosť rozvrhovač uloží záznam do entovej pamäte. Po ukončení jednorázovej úlohy je jej záznam zmazaný. Záznamy ostatných činností sú v pamäti až do ich odobratia volaním `rozvrhovacOdeberUlohu`.

Činnosť je ukončená, pokiaľ:

a. bola spustená a skončila (uspela alebo zlyhala)
b. vyčerpala svoju lichobežníkovú prioritu
c. bol na ňu volaný `rozvrhovacOdeberUlohu`
d. nastal nový deň

Interrupt dennej úlohy (pravidelnej i jednorázovej) je zrušený po ukončení tejto úlohy (podľa bodov a.–d.). Zároveň je úloha odobratá zo zoznamu naplánovaných úloha.

Interrupt životnej funkcie je zrušený a ž.f. je odstránená zo zoznamu naplánovaných úloh, ak je ukončená nedobrovoľne, tj. podľa bodov b.–d.

Aby činnosť zistila, že má byť ukončená podľa bodov b.–d., má každá činnosť navesený lokálny interrupt, ktorý to kontroluje.

Každá činnosť má ešte ďalší lokálny interrupt, ktorým si kontroluje, či nebola prerušená. Ak bola, je spustená od začiatku (`RERUN`ovaná).

Pre plánovanie činností má rozvrhovač navesený vlastný globálny interrupt, ktorý sa spustí, keď je zoznam naplánovaných úloh prázdny. Vtedy vyzdvihne z pamäti záznamy o denných úlohách a životných funkciách a naplánuje ich.

Naplánovanie činnosti znamená navesenie interruptu, ktorý sa spustí na platnosť podmienky činnosti, alebo keď nastane nový deň (aby činnosť dostala možnosť ukončiť sa). Po navesení je činnosť uložená do zoznamu naplánovaných úloh.

V tele interruptu spúšťajúceho činnosť sa potom spustí samotná činnosť. Tomu však ešte predchádza niekoľko akcií. Postup spustenia činnosti je nasledovný:

- zablokuje sa ďalšie spustenie interruptu
- nastaví sa skriptu priorita
- ak beží iná činnosť, porovná sa jej odhad s toleranciou spúšťanej činnosti. Ak je odhad menší ako tolerancia, je práve bežiacej činnosti daná možnosť na dokončenie a to tak, že spúšťaná činnosť sa uspí na čas svojej tolerancie, avšak tak, aby sa zobudila hneď, ako bude činnosť ukončená, alebo prerušená inou.
- po prípadnom uspaní a prebudení je vhodné znova skontrolovať platnosť podmienky. (Činnosť už totiž nemusí chcieť bežať; napr., ak už záhon zalial niekto za nás.) Tiež, ak bola činnosť zobudená na prerušenie bežiacej činnosti inou, mala by sa tolerancia (pravdepodobne už zmenená)

porovnať s odhadom prerušujúcej činnosti a prípadne sa znovu uspať a dať priestor na dokončenie tejto úlohe. Oboje zariadime jednoducho tak, že po prebudení svoj interrupt povolíme a ukončíme sa `FAIL`. To spôsobí nový výber činnosti, teda nové testovanie podmienky, a keď sa znova dostane na nás, porovnanie tolerancie a odhadu prebehne pre novú bežiacu činnosť.

- Ak ešte stále beží iná úloha, znamená to, že buď nedostala šancu na dokončenie, alebo ju nevyužila a bude aj tak prerušená. Avšak úloha, ktorá je prerušovaná a pre ktorú sa má vykonať prechodné správanie, môže byť rôzna od tej práve bežiacej. To sa stane v nasledujúcich prípadoch:
  - beží činnosť A, je prerušená činnosťou B. Nato sa zobudí činnosť C. Tá vidí, že beží B, avšak činnosť, ktorá je prerušovaná, je A
  - beží A, je prerušená činnosťou B. Začne sa vykonávať prechodné správanie z A na B, to je však prerušené činnosťou C. Tá opäť vidí ako bežiacu činnosť B, avšak prerušovaná je A

  Preto si budeme pamätať, ktorá činnosť je prerušovaná. Na tomto mieste potom, ak prerušovaná činnosť ešte nie je nastavená, nastavíme ako prerušovanú činnosť tú bežiacu. Inak prerušovanú činnosť ponecháme bez zmeny.
- na tomto mieste už je jasné, ktorá činnosť bude prerušená (ak vôbec nejaká) v prípade, že pobežíme ďalej. Teraz nastavíme seba ako bežiacu činnosť. Odteraz si tiež budeme kontrolovať, či sme neboli prerušení. V prípade, že áno, spustíme sa od začiatku (`RERUN`). To, že sme sa stali bežiacou činnosťou, môže zobudiť nejaké spiace činnosti, ktoré, ak majú vyššiu prioritu a nedajú nám šancu na dokončenie, nás prerušia
- ak je nastavená nejaká prerušovaná úloha, spustí sa transition z prerušovanej úlohy na nás. Po ukončení sa nastaví, že nie je prerušovaná žiadna úloha
- pred spustením samotnej činnosti ešte navesíme lokálny interrupt, ktorý kontroluje, či nenastal nový deň, činnosti nevypršala priorita, alebo nebol na ňu volaný `rozvrhovacOdeberUlohu`. Ošetrujúcou akciou je:
  - odstránenie činnosti zo zoznamu naplánovaných úloh
  - zrušenie spúšťacieho interruptu činnosti (ktorý je v tomto momente len blokovaný)
  - činnosť prestane byť bežiaca
  - činnosť je ukončená
- konečne môže byť spustená samotná činnosť
- po dobehnutí je odstránena zo zoznamu naplánovaných úloh, prestane byť bežiacou. V prípade životnej funkcie je jej interrupt znova povolený, v prípade dennej úlohy je zrušený. Ak sa naviac jednalo o jednorázovú úlohu, je záznam o nej odstránený z pamäte enta a sú zrušené všetky prechodné správania registrované pre danú úlohu

Podrobnosti ďalších skriptov rozvrhovača sú zrejmé zo zdrojového kódu,

ktorý je na priloženom CD, a jeho komentárov.

A.9 Ukážka správania

Implementoval som malú ukážku nových prechodných správaní a odloženia správania. Postup, ako si ju spustiť je v časti o inštalácii a spustení (príloha B). Skripty sú napísané tak, aby boli nové prvky vidieť čo najskôr. Preto enti budú často hladní, smädní a potrebovať ísť na záchod. Svet bude rovnaký ako ten ukážkový z balíka projektu, len vlastnosti niektorých objektov budú zmenené. Vo svete budú dvaja simulovaní enti, záhradník a hudobník. Títo tiež pochádzajú z ukážkového sveta. Avšak záhradník bude využívať prechodné správania a odloženie správania. Skripty hudobníka sú nezmenené a hudobník bude teda dôkazom, že i pôvodné skripty fungujú s novou knižnicou rozvrhovača.

U záhradníka budeme sledovať úlohy *zalievanie* a *okopávanie* a životné funkcie *jedenie*, *odskočenie si na záchod* a *pitie*. V tomto poradí sú zároveň aj ich priority (od najnižšej). Zalievanie pritom chce bežať len v čase od cca 5:06 do 8:00. Okopávanie od 7:47 do 10:20.

Popíšeme priebeh správania záhradníka. Pretože užívateľ má možnosť zasahovať do sveta, záleží správanie entov aj od jeho konania. Opisovaný priebeh nastane v prípade, že užívateľ entom nebude pomáhať ani prekážať, len ich z diaľky pozorovať.

Krátko po 5:00 sa záhradník rozhodne, že bude zalievať. Vyberie sa do komory po krhlu, napustí do nej vodu a príde do záhrady.

5:30 potrebuje ísť na záchod. Zalievanie bude ešte trvať dlho, preto je prerušené. Ent vykoná prechodné správanie — ak má v krhle nejakú vodu, tak ju vypolieva a krhlu položí. Potom si odskočí na záchod a vráti sa do záhrady.

5:45 začne mať hlad, ale ešte vydrží a tak pokračuje v zalievaní. Jedenie je teda odložené a čaká na dokončenie zalievania.

6:00 sa znovu ozve potreba na záchod. Tentokrát už zalievanie predpokladá, že skončí skôr a tak je odskočenie si na záchod odložené a čaká na ukončenie zalievania.

6:30 ent začne byť smädný. Môže s tým ale ešte počkať a tak je pitie odložené. Teraz správania jedenie, odskočenie a pitie čakajú na dokončenie zalievania.

6:38 vyprší tolerancia jedla a už nemôže počkať toľko, koľko zalievanie teraz odhaduje. Zalievanie bude teda prerušené. Lenže na prerušenie zalievania čakajú aj pitie a odskočenie si. Pitie má najvyššiu prioritu a tak dostane možnosť byť vykonané ako prvé. Pitie ale ešte môže počkať toľko, koľko jedenie potrebuje a tak je pitie znovu odložené a čaká na dokončenie jedenia. Odskočenie si čakalo na prerušenie zalievania a momentálne má najvyššiu prioritu zo správaní, ktoré chcú bežať (odskočenie si a jedenie). Spýta sa teda jedenia, koľko bude trvať. Odskočenie si síce ešte mohlo chvíľu čakať na zalievanie, ale už nemôže čakať toľko, koľko potrebuje jedenie. Nedá teda jedeniu možnosť bežať. Na to sa znovu ozve pitie. To totiž čaká na

dokončenie jedenia, ktoré ale bežať nebude. Pitie mohlo čakať na jedenie, ale čo ak by odskočenie si trvalo dlhšie? Pitie sa teda spýta odskočenia si, ako dlho bude trvať. Pitie usúdi, že toľko počkať môže a tak je odložené a čaká, kým si ent neodskočí. Ent teda pôjde na záchod, ešte predtým ale vykoná prechodné správanie (zo zalievania na odskočenie si). Keď si vybaví potrebu, najvyššiu prioritu má pitie a žiadne správanie nie je rozrobené (zalievanie bolo riadne prerušené, jedenie ani nebolo začaté). Ent sa teda ide napiť, potom najesť a potom sa vráti k zalievaniu.

7:15 ent opäť potrebuje ísť na záchod. Zalievanie už ale nebude trvať dlho, a tak odskočenie počká. 7:30 sa opakuje rovnaká situácia s jedením. Odskočenie a jedenie teda obe čakajú na ukončenie zalievania.

7:47 chce začať okopávanie, ktoré tvrdí, že musí začať hneď (tj. nemá žiadnu toleranciu). Zalievanie bude teda prerušené. Na to čakajú odskočenie si a jedenie. V tomto poradí však obe dajú šancu okopávaniu, ktoré (mylne) tvrdí, že skončí rýchlo. Vykoná sa teda prechodné správanie zo zalievania na okopávanie a spustí sa okopávanie.

8:17 odskočeniu vyprší tolerancia, ale prehodnotí svoju potrebu a ešte môže počkať 10 minút. O 10 minút sa situácia opakuje a odskočenie môže počkať ďalších 10. Krátko na to sa ozve pitie, ktoré ale tiež môže počkať. Okopávanie totiž vytrvalo tvrdí, že prebehne rýchlo.

8:36 odskočeniu dôjde trpezlivosť a už dlhšie čakať nemôže. Okopávanie bude prerušené. Z čakajúcich má najvyššiu prioritu pitie. Pitie však ešte môže čakať a tak si ent odskočí (po prechodnom správaní, ktorým je teraz odloženie nástrojov, ak nejaké drží). Potom sa pôjde napiť. Jedenie tiež čakalo a teraz sa dostane k slovu. Pretože hlad už bol väčší, chvíľu trvá, než sa naje. Ešte počas jedenia ent znova potrebuje ísť na záchod :-) (o 9:00), ale nie nutne a tak počká. Keď jedenie skončí, ent pôjde na záchod a potom sa vráti k okopávaniu.

9:30 je ent už zase hladný, ale nie veľmi. Okopávanie teda pokračuje, jedenie čaká.

9:45 by ent chcel opäť na záchod, ale okopávanie už bude končiť, a tak vydrží. Hneď nato okopávanie skončí a ent si odbehne na záchod.

Tu naša ukážka končí. Najbližších niekoľko hodín ent nemá nič na práci, a tak sa bude potulovať po dome a skoro stále jesť, piť a chodiť na záchod :-).

Top-level skript pre takéto správanie vyzerá asi takto:

```
Toplevel:  Zi;

Zi
$-
  return 0.
:-
  rozvrhovacInit,

  rozvrhovacRegistrujUlohu("zalejZahony", _, _,
      20, 30, 500, 540, "zalejZahonyOdhad", zalievanieUID),
```

```
rozvrhovacRegistrujUlohu("okopajZahony", _, _,
     500, 35, 920, 960, "okopajZahonyOdhad", okopavanieUID),
rozvrhovacRegistrujUlohu("bludenie", _, 5, _),

rozvrhovacRegistrujZivotniFunkci("odskocSi", "odskocSiTest",
     70, "odskocSiOdhad", odskocSiUID),
rozvrhovacRegistrujZivotniFunkci("najedzSa", "najedzSaTest",
     40, "najedzSaOdhad", _),
rozvrhovacRegistrujZivotniFunkci("zazenSmad",
     "zazenSmadTest", 80, _),

rozvrhovacRegistrujTransition(
     zalievanieUID, odskocSiUID, "vyprazdniKrhlu"),
rozvrhovacRegistrujTransition(
     zalievanieUID, okopavanieUID, "vyprazdniKrhlu"),
rozvrhovacRegistrujTransition(
     okopavanieUID, odskocSiUID, "uvolniRuky"),

rozvrhovacStart.
```

A.10 Zásahy do interpretu jazyka E

Počas implementácie nových prvkov do rozvrhovača sa vyskytli problémy, s ktorými som sa musel vysporiadať.

Prvý problém sa týkal sleep interruptu. Menším problémom bola jeho syntax. V príručke je uvedený ako `sleepHook`. Tento lexikálny element však interpret nepoznal. V zdrojovom kóde pre lexikálnu analýzu som dohľadal, že sleep interrupt skutočne existuje a zapisuje sa ako `hookSleepUntil`. Väčším problémom bolo, že nefungoval. Jeho podmienka sa po uspaní skriptu, ktorý ho navesil, ani vôbec nekontrolovala. Ukázalo sa, že interpret medzi globálnym a lokálnym interruptom rozlišoval podľa toho, že lokálny má vlastníka, kým globálny nie. Sleep interrupt síce vlastníka má, ale jeho podmienka má byť kontrolovaná akoby to bol globálny interrupt. S informáciou o vlastníkovi teda nevystačíme. Bolo treba pridať do triedy reprezentujúcej interrupt jeden element, ktorý indikuje, či je interrupt lokálny alebo globálny, a postarať sa o jeho správne nastavenie pri vytváraní interruptu. Zároveň bolo treba upraviť všetky časti kódu, ktoré zisťujú, či je interrupt lokálny alebo globálny.

Ďalší problém sa týkal príkazu `reconsider`. Ten interpret síce rozpoznal, no nebol nijako rozumne implementovaný a po jeho zavolaní interpret spadol na segmentation fault. Tento príkaz som dopísal tak, že po jeho zavolaní sa znovu začne s výberom akcie. Narozdiel napríklad od príkazov `COMMIT` alebo `FAIL`, volajúci skript zostane aktívny a po navrátení sa k nemu výpočet pokračuje od miesta za reconsider.

Posledným zásahom do interpretu bolo prepísanie niektorých dátových štruktúr tak, aby sa s nimi pracovalo efektívnejšie. V podstate som tieto štruktúry nahradil triedami z STL a prepísal kód, ktorý ich používa. Beh interpretu, a tým aj celej simulácie, sa tým znateľne zrýchlil.

B Inštalácia a spustenie Projektu Enti

Projekt je dostupný len pre operačný systém Linux.

B.1 Čo budeme potrebovať

Zdrojové kódy Projektu ENTi

- oficiálne vydané zdrojové kódy projektu
- upravené časti interpretu
- novú knižnicu rozvrhovača

Všetky tieto časti sú na priloženom CD.

Softwarové nástroje

- Perl, interpret jazyka (`www.cpan.org`)
- knižnica Qt (`www.trolltech.com/products/qt/`)
- knižnica SDL (`www.libsdl.org`)
- Mercury, kompilátor jazyka (`www.cs.mu.oz.au/research/mercury/`)

  Vyskúšané s verziou 0.12.2 (current stable release k 8. augustu 2006). Táto verzia je zároveň priložená na CD pre prípad, že kompilátor Mercury nie je dostupný z vašej linuxovej distribúcie.
- FreePascal, kompilátor jazyka (`www.freepascal.org`)

  Pozor, s verziou 2.0.0+ to nefunguje. Potrebujeme nejakú staršiu. Odskúšané je to s verziou 1.0.6. Tá už ale nie je podporovaná vývojármi a stratila sa aj z ich webu. Preto je priložená na CD.
- GCC, kompilátor jazyka C/C++

  Odskúšané s verziou 3.3.6, verzia 3.4.* nefunguje.

B.2 Postup inštalácie

Na priloženom CD sa všetko týkajúce sa inštalácie nachádza v adresári `enti`. Cesty v tomto odstavci budú uvádzané relatívne k tomuto adresáru.

1. Nainštalujeme potrebné SW nástroje (tj. Perl, Qt, SDL, Mercury, FreePascal, GCC) z linuxovej distribúcie, prípadne z priloženého CD. Mercury a FreePascal sú priložené v adresári `tools`. Presvedčte sa, či sú v `PATH` dostupné správne verzie programov (obzvlášť GCC a FreePascal).
2. rozbalíme zdrojové kódy projektu

   ```
   tar xzf enti-2005-10-14.tgz
   ```
3. nahradíme upravované zdrojové kódy interpretu a knižnicu rozvrhovača

   ```
   cp -f patches/* enti-2005-10-14/src/ent/mikulas/simple/
   cp -f transitions/lib/rozvrhovac.e enti-2005-10-14/lib/
   ```
4. spustíme konfiguráciu

   ```
   cd enti-2005-10-14
   ./configure
   ```
5. preložíme a nainštalujeme

```
make && make install
```

Ak sa prekladač FreePascalu sťažuje, že nenašiel nejakú knižnicu, správne nastavíme cestu k nim (k správnej verzii!) v súbore `~/.ppc386.cfg` a skúsime znova.

6. nastavíme premenné prostredia podľa inštrukcií zobrazených na konci inštalačného skriptu

B.3 Spustenie

1. skopírujeme si niekam ukážkový balík sveta a inštanciujeme ho
```
cp -R $ENTIROOT/ukazkovy_svet ~/moj_svet
```
2. instanciujeme balík sveta
```
cd ~/moj_svet
make
```
3. spustíme
```
entiserver start &
ent &
ent &
entiprohlizec
```
Pre lepší prehľad výstupov jednotlivých programov je lepšie spustiť každý príkaz zo samostatného terminálu. Príkazy `entiserver` a `ent` je treba spúšťať z adresára balíku sveta.
4. aby sme videli nejakú ukážku novoimplementovaných prvkov, nahradíme niektoré skripty balíku sveta a knižnice upravenými
```
cp -Rf /cesta_k_CD/enti/transitions/lib $ENTIROOT
cp -Rf /cesta_k_CD/enti/transitions/svet/* ~/moj_svet
```
5. opakujeme kroky 2. a 3. a sledujeme výpisy záhradníka (jeden z príkazov `ent`)

Popis práce s grafickým rozhraním nájdete v *Užívateľskej príručke*, ktorá je súčasťou dokumentácie projektu ([2]) a možno ju tiež nájsť na CD v inštalačnom balíku Entov (`enti-2005-10-14.tgz`), po jeho rozbalení v súbore `enti-2005-10-14.tgz/doc/WWW/publications/uziv.pdf`.

Literatúra

[1] Bojar O., Brom C., Hladík M., Toman V. (2005): *The Project ENTs: Towards Modeling Human-like Artificial Agents.* In: SOFSEM 2005 Communications - student research forum, Slovak Republic.

[2] Bojar, O., Brom, C., Hladík, M., Toman, V., Vejlupek, M., Voňka, D. (2002): *Enti, Dokumentace studentského softwarového projektu.* Matematicko-fyzikální fakulta, Univerzita Karlova, Praha.

[3] Brom C. (2005): *Hierarchical Reactive Planning: Where is its limit?* In: Proceedings of MNAS – Modelling Natural Action Selection. Edinburgh, Scotland.

[4] Bryson J. J. (2001): *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents.* PhD thesis, Massachusetts Institute of Technology, 59–75.

[5] Mateas M., Stern A. (2004): *A Behavior Language: Joint Action and Behavioral Idioms.* Chapter in H. Prendinger and M. Ishizuka (Eds), Life-like Characters. Tools, Affective Functions and Applications, Springer

[6] Sengers P. (1999): *Designing Comprehensible Agents.* In: Proceedings of the 16th International Joint Conference on Artificial Intelligence. Stockholm, Sweden.