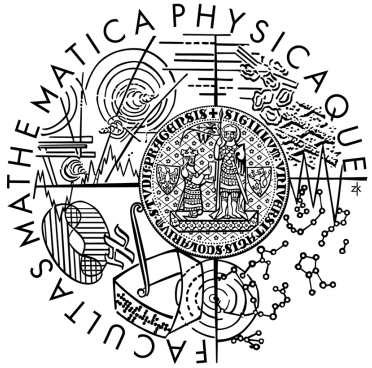
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Václav Brodec

Vývojové prostředí rozšiřující možnosti řízení dialogu v AIML

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: doc. RNDr. Vladislav Kuboň, Ph.D.

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2014

Děkuji především panu docentu Vladislavu Kuboňovi za trpělivé a podnětné vedení bakalářské práce, dále paní doktorce Silvii Cinkové za cennou konzultaci.

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

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

V Praze dne 4. 12. 2014 podpis

Název práce: Vývojové prostředí rozšiřující možnosti řízení dialogu v AIML

Autor: Václav Brodec

Katedra / Ústav: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: doc. RNDr. Vladislav Kuboň, Ph.D., Ústav formální a aplikované lingvistiky

Abstrakt: Jazyk AIML vznikl jako nástroj na tvorbu jednoduchých konverzačních agentů. Proto postrádá některé z vlastnosti pokročilých dialogových systémů. Jednou z nich je podpora pro řízení dialogu. Ta je přitom užitečná v mnoha aplikacích, v nichž byl jazyk díky své popularitě nasazen.
Práce řeší problém implementace řízení dialogu v čistém AIML. A to nasazením rozšířených přechodových sítí při návrhu a generování zdrojového kódu. Jejím výsledkem je vývojové prostředí, které podporuje zvolený postup, a podstatně tak usnadňuje návrh složitějších robotů, aniž by bylo nutné uchýlit se k rozšíření standardního interpretu.

Klíčová slova: AIML, vývojové prostředí, řízení dialogu

Title: Development Environment Extending the Dialog Management Options of AIML

Author: Václav Brodec

Department: Institute of Formal and Applied Linguistics

Supervisor: doc. RNDr. Vladislav Kuboň, Ph.D., Institute of Formal and Applied Linguistics

Abstract: The AIML language was created with a goal of authoring of simple chat bots. Therefore it lacks some of the features of advanced dialog systems. One of them is the support for dialog management, which is beneficial in many applications that the language has already spread into due to its popularity.
This thesis solves the problem of dialog management implementation in pure AIML by using the augmented transition networks in design and code generation. It results in a development environment that supports the chosen solution, thus facilitating the design of more complex bots, while maintaining compatibility with standard interprets.

Keywords: AIML, development environment, dialog management

Obsah

Úvod

Přestože je jazyk AIML (Artificial Intelligence Markup Language, „značkovací jazyk pro umělou inteligenci“) považován za velmi prostý a hrubý nástroj pro tvorbu konverzačních robotů (chatterbotů), stále patří k nejpopulárnějším přinejmenším mezi z pohledu lingvistiky amatérskou uživatelskou základnou.[1] O jeho doložitelných úspěších svědčí četná vítězství robotů postavených na této technologii v soutěži o Loebnerovu cenu, i v posledních dvou letech.[2]

Původním smyslem robotů napsaných v AIML byla povrchní simulace diskuze se záměrem ošálit lidský protějšek. Postupně se ale díky své přístupnosti etablovaly i v jiných oblastech. Dnes se v komerční sféře užívají v aplikacích jako je ovládání hlasem, služby pro zasílání zpráv, automatizace zákaznické podpory, interaktivní vyhledávání, reklama, asistované vyplňování formulářů a další.[3] AIML ovšem pro tyto účely postrádá některé zásadní, a přitom dosažitelné nástroje. I s přihlédnutím k jeho inherentním omezením klade na programátora poměrně velkou zátěž, zejména v oblasti řízení dialogu. Zatímco v případě simulace přirozené konverzace jsou některé aspekty chatterbotů, jako je právě nevyzpytatelný průběh rozhovoru, žádoucí, stávají se překážkou tehdy, kdy má mít diskuze jiný účel kromě diskuze samotné – typicky naplnit nějakou konkrétní potřebu lidského protějšku.

Práce si klade za cíl poskytnout programátorovi chybějící nástroj pro návrh struktury dialogu, a ulehčit mu tak tvorbu komplexních účelových robotů s využitím AIML. Opírá se přitom o analýzu slabin i silných stránek jazyka. Poskytuje odůvodnění toho, proč je řízení dialogu vhodnou oblastí pro rozšíření, a odpovídá na otázku, které z existujících řešení tohoto problém je přínosné i adaptovatelné v omezujících podmínkách jazyka. Jejím praktickým výsledkem je vývojové prostředí, které kromě podpory řízení dialogu poskytuje vše potřebné k vytváření robotů, aniž by kompromitovalo výhody zvolené platformy.

V její 1. kapitole je podán základní přehled o prostředcích jazyka AIML. V kapitole 2. jsou tyto zhodnoceny a srovnány s dialogovými systémy a navrhnuty perspektivní směry rozšíření. Na jeden z nich, řízení dialogu, se zaměřuje kapitola 3. V kapitole 4. jsou navrženy principy řízení dialogu v AIML na základě rozšířených

přechodových sítí, kapitola 6. dokumentuje jejich implementaci ve výsledném vývojovém prostředí. Mezitím kapitola 5. toto prostředí uvádí a popisuje z pohledu uživatele.

1. Přehled prostředků jazyka AIML

Jazyk AIML je dialektem XML (obecně rozšířený jazyk vyvinutý konsorciem W3C). Tedy každý validní zdrojový soubor jazyka AIML je zároveň validním XML dokumentem. Ve verzi 1.0.1 jej specifikuje dokument „A.L.I.C.E. AI Foundation Official AIML 1.0.1 standard“, jehož poslední verze je dostupná na webové adrese http://www.alicebot.org/TR/2011/ [4]. Jazyk vznikl koncem 90. let 20. století, za jeho otce se považuje americký výzkumník Dr. Richard S. Wallace. Teprve nedávno (2013) byla zveřejněna pracovní specifikace jeho nové, druhé verze, která jazyk mírně rozšiřuje.[5] Přestože od tohoto místa následující tvrzení většinou platí i pro ni, nadále je uvažován jazyk AIML ve verzi 1.0.1. Právě tuto široce rozšířenou verzi užívá i popisované vývojové prostředí Botníček.

1.1. Základní stavba jazyka

V své publikaci [6] Dr. Wallace představuje jazyk v následující podobě: Základní prvek jazyka je označován jako kategorie. Každá kategorie (*category*) obsahuje textový vzor (*pattern*), vůči kterému se porovnává vstup uživatele konverzujícího s robotem. Dále kategorie obsahuje výstupní kód, specifikující reakci robota, v podobě šablony (*template*). Při rozhodování, zda-li má být na daný vstup aplikována určitá kategorie, hraje kromě hlavního vzoru pattern roli ještě jeden, nepovinný vzor (*that*), který je srovnáván s předchozí promluvou robota (při absenci that takové srovnání vždy uspěje). Konečně kategorie mohou, ale nemusí, být sdruženy do společného tématu (*topic*), jež je popsáno stejným způsobem jako vzory. Jak jsou tyto struktury zachyceny ve zdrojovém kódu, ilustruje výpis v obrázku 1.

Většina dalších prvků jazyka se obvykle vyskytuje ve zmiňované výstupní šabloně template. Teprve tyto další značky dělají ze souboru kategorií více než pouhou tabulku překládající uživatelský vstup na odpověď robota. Jejich funkce ovšem nejlépe vynikne až po osvětlení procesu vedoucího od zadání uživatelského vstupu po finální odpověď robota.

```
...
    <topic name="VZOR PRO POPIS TÉMATU">
        <category>
            <pattern>VZOR PRO VSTUP</pattern>
            <that>VZOR PŘEDCHOZÍ REAKCE ROBOTA</that>
            <template>
                ...
            </template>
        </category>
        ...
    </topic>

    <category>
        <pattern>VZOR PRO VSTUP</pattern>
        <that>VZOR PŘEDCHOZÍ REAKCE ROBOTA</that>
        <template>
            Šablona s reakcí může obsahovat prostý text i další značky.
        </template>
    </category>
...
```

Obrázek 1 – kořenové prvky jazyka

1.2. Průběh předzpracování vstupu

Promluva uživatele k robotu je po předání interpretu v podobě textového řetězce nejdříve podrobena předzpracování. Předzpracování má tři fáze, ale specifikace vyžaduje pouze jednu z nich. Jejich pořadí je ovšem pevně určeno a každá následující přebírá jako vstup výstup z předchozí. První se nazývá *substituce*. V této nepovinné fázi se dle nastavení a implementace interpretu nahradí některé podřetězce vstupu. Obvykle jde o takové, jež obsahují informaci, která se musí uchránit před destruktivním vlivem dalších fází předzpracování.

Druhou fází je *dělení vět*. V té interpret rozdělí vstup na věty. Za věty jsou ve smyslu zpracování interpretem považovány sousledné úseky textu, na které robot nezávisle reaguje. Výstupní reakce na jednotlivé věty jsou pak ve výsledku spojeny v odpovídajícím pořadí do jednoho textového řetězce, který se vrací jako odezva na celou promluvu. Obvykle věty odpovídají větám v běžném pojetí slova, ovšem jak přesně bude text na ně rozdělen, záleží opět na příslušné implementaci interpretu. Běžně se (alespoň pro angličtinu) užívá dělení dle interpunkčních znamének. Právě proto, aby nedocházelo k dělení na špatných místech (např. kvůli zkratkám s tečkami), jsou rozpoznané podřetězce v první fázi substituovány výrazem, který interpunkci již neobsahuje. Např. zkratku „Mr.“ nahradí úplné „Mister“.

Konečně třetí, poslední a povinnou fází předzpracování je *převod textu do*

formátu, s nímž dokáže pracovat porovnávání vzorů. Porovnávání vzorů je užíváno při hledání kategorie, která odpovídá uživatelově vstupu, a pracuje s textem, který se nachází v tzv. *normálním tvaru*. Normální tvar je definován takto: tvoří jej pouze *normální slova* oddělená mezerami. A konečně normální slova se sestávají výhradně z *normální znaků*, tedy číslic a velkých písmen (neuvažujeme-li pro zjednodušení exotická písmena bez rozlišení velikosti). Tedy text, který vstupuje do závěrečná fáze je zpracován tak, že jsou v něm veškerá písmena převedena na velká, všechny ostatní znaky kromě číslic pak na mezery. Případné nadbytečné mezery mezi slovy a na koncích se odstraňují. Např. řetězec „Ahoj, já jsem C-3PO!“ by byl převeden na „AHOJ JA JSEM C 3PO“. V tomto kroku se mohou ze vstupu ztratit neslovní výrazy, např. emotikony (neboli „smajlíci“). V takové situaci by měl interpret předem zasáhnout a první fází postižené výrazy převést na normální slova, např. „:-/“ na „NEMÁM Z TOHO DOBRÝ POCIT“.

1.3. Porovnávání vzorů

Interpret hledá správnou odpověď tak, že vezme předzpracovaný uživatelský vstup a přidá k němu navíc předchozí robotovu odpověď a aktuálně probírané téma (oboje normalizované stejným způsobem jako vstup uživatele). Tyto tři části srovná s databází kategorií, které načetl pro instalovaného robota. Kategorie, jejíž vzor pattern nejlépe odpovídá normalizovanému uživatelskému vstupu a jejíž vzor that nejlépe odpovídá předchozí odpovědi robota a jejíž vzor tématu nejlépe odpovídá aktuálnímu, je pak vybrána a její šablona template předána jako odpověď pro další zpracování. Může se stát, že neodpovídá žádná kategorie, v takovém případě interpret ohlásí neúspěch.

Validní vzory mají stejný formát jako normalizovaný vstup s výjimkou toho, že kromě normálních slov mohou obsahovat i *zástupné znaky* * (hvězdička) a _ (podtržítko). Tyto znaky se mohou vyskytovat ve vzoru na stejných místech jako slova a každý musí být stejně jako slovo od zbytku vzoru oddělen mezerou. Takže zatímco „_ NEMÁM ŽÁDNÝ * *“ je platným vzorem, výraz „** JSEM TO CÍTILA“ nikoli. Zástupné znaky při porovnávání odpovídají jednomu a více slovům vstupu. Tedy vstup „CO JE TO“ lze úspěšně porovnat se vzorem „CO JE *“, nikoli

však „CO JE TO *“.

Komparaci vstupů se vzory (a určení odpovídající kategorie) lze modelovat takto (zanedbají-li se pro názornost vzory s podtržítkem): Na jedné straně se nachází uživatelský vstup, předchozí promluva robota a aktuální téma. Pro tento konglomerát se v terminologii AIML užívá označení *vstupní cesta* (to má svůj původ ve standardním způsobu implementace). Na straně druhé je pak kategorie se svými vzory pattern, that (obsahuje jen zástupný znak *, pokud kategorie nemá tento vzor) a vzorem tématu (zástupný znak *, pokud je kategorie definována mimo téma). Porovnávají se vždy jedna ze tří částí vstupní cesty s příslušným vzorem. Aby kategorie uspěla, musí srovnání uspět u všech jejích vzorů.

Každá část i vzor se zpracovávají zleva doprava, postupně po jednotlivých slovech a zástupných znacích. Pokud je zkoumaný prvek vzoru normálním slovem a shoduje-li se se slovem části cesty na dosažené pozici, přesune se kurzor části na další slovo a kurzor vzoru na další prvek. Pokud se neshoduje, vzor dané části neodpovídá, a tak ani kategorie celé vstupní cestě.

V případě, že se ve vzoru kurzor posune na pozici se zástupným znakem hvězdička, tak je slovo z části cesty na aktuální pozici vždy považováno za odpovídající. Zároveň se stává začátkem a prozatímním koncem myšleného úseku zachyceného oním zástupným znakem. Dále se postupuje tak, že se se zbytkem vzoru za hvězdičkou zkouší porovnat zbytek části cesty za zachyceným slovem. Pokud dojde na nějaké další pozici ke shodě, je zachytávání slov hvězdičkou ukončeno. Při jakékoli neshodě se slovo z části cesty přeskočí a považuje se za nový konec dosud zachyceného úseku. S ostatkem vzoru (kde se až do nalezení shody dále kurzor neposouvá) se porovná o toto slovo menší sufix části.

Porovnávání se vzorem probíhá do té doby, než se vyčerpají slova z příslušné části vstupní cesty. Považuje se za úspěšné jen tehdy, pokud se zrovna došlo na konec vzoru či probíhalo zachytávání do zástupného znaku. Např. vstupní cesta složená ze vstupu „ALE BEZE VŠEHO“, předchozí promluvy robota „MOHU SE VÁS ZEPTAT“ a s aktuálním tématem „KRIZE“ může aktivovat kategorii se vzorem pattern „ALE *“, vzorem that “* ZEPTAT“ v tématu „*“. Hvězdičkami budou zachyceny úseky „BEZE VŠEHO“, „MOHU SE VÁS“ a „DOTAZ“.

Praktickým důsledkem popsaného postupu je to, že záchytné znaky při shodě celé (neprázdné) části vstupní cesty zachytí vždy alespoň jedno slovo a zachytávání není hladové (ve smyslu, že u za sebou se opakujících zástupných znaků zachytí všechny až na poslední právě jedno slovo). Dále je díky němu garantováno, že když se v souboru kategorií robota vyskytuje taková, která má na místě všech tří vzorů hvězdičky, interpret je vždy schopen najít odpovídající kategorii.

Algoritmus ještě mírně komplikuje zástupný znak podtržítko. Kategorie jsou totiž pro porovnání částečně uspořádány postupně podle vzoru pattern, vzoru that a vzoru tématu. Uspořádání je definováno tak, že kategorie, které mají na dané pozici zástupný znak podtržítko jsou zkoušeny před kategoriemi, jež zde mají konkrétní slovo, a teprve po nich následují ty, jež tam mají hvězdičku. Tedy např. normalizovaný vstup „VELICE MĚ TĚŠÍ“ bude v případě přítomnosti obou kategorií zachycen spíše tou se vzorem „_ MĚ TĚŠÍ“ než doslovným „VELICE MĚ TĚŠÍ“.

1.4. Rekurze a zpětné odkazy v šabloně

Po nalezení kategorie, jež odpovídá vstupu, přichází na řadu zpracování její šablony s odpovědí. Zdrojový kód prvku template může kromě prostého textu obsahovat různé značky, které mohou zásadním způsobem proměnit to, jak bude vypadat odpověď, kterou interpret na konec vrátí. Je to především značka *srai* pro rekurzivní zpracování částí šablony, jež dodává jazyku AIML výpočetní sílu. Při rekurzivním zpracování se část šablony obklopená touto značkou předá interpretu obdobně jako běžný uživatelský vstup (na rozdíl od něj však není zaznamenávána pro pozdější odkazování). Obsah značky tedy opět prochází předzpracováním, je porovnáván se vzorem pattern a nalezená šablona odpovědi je po vlastním vyhodnocení dosazena jako prostý text na místo značky srai v původní šabloně.

Prvek srai dále doplňují značky, které odkazují na části vstupu právě zachycené zástupnými znaky. U značek *star*, *thatstar*, *topicstar*, které poskytují odkazy na zachycené části vstupu, resp. předchozí promluvy robota, resp. tématu, se určuje pořadí zástupného znaku, jehož obsah má být při zpracování vracen, pomocí číselného atributu index. Tento atribut lze vynechat, a pak se uvažuje první zástupný

znak v daném vzoru.

Kombinace rekurzivního zpracování a zpětných odkazů na zachycené částí má široké uplatnění, které je klasifikováno takto:

- Symbolické redukce (obrázek 2) – jinými slovy převedení složitých vstupů na jednodušší. Při psaní robota je žádoucí odpověď uložit pod co nejjednodušším vzorem a na ten odkazovat ze vzorů postihujících rozvitější alternativy:

```
...
<category>
    <pattern>NEVÍŠ NÁHODOU KDO TO BYL *</pattern>
    <template>KDO JE <srai><star/></srai></template>
</category>
...
```

Obrázek 2 – symbolická redukce

- Rozděl a panuj (obrázek 3) – vstup může být vhodné rozdělit na části a ty zpracovat samostatně a výsledek zpět složit do jedné odpovědi:

```
...
<category>
    <pattern>NE *</pattern>
    <template><srai>NE</srai> <srai><star/></srai></template>
</category>
...
```

Obrázek 3 – rozděl a panuj

- Převedení na ekvivalent (obrázek 4) – kategorie může obsahovat pouze jeden vzor pattern, proto se nabízí převést výrazy na kanonický tvar (či např. výraz s překlepem na správný):

```
...
<category>
    <pattern>SOUHLASÍM</pattern>
    <that>*</that>
    <template><srai>ANO</srai></template>
</category>
...
```

Obrázek 4 – převedení na ekvivalent

- Klíčová slova (obrázek 5) – autor robota má možnost aktivace šablony přítomností klíčového slova na vstupu. To lze provést takto (první kategorii spouští izolované slovo, na její šablonu je odkazováno ze zachycení

klíčového slova jako prefixu, sufixu a infixu vstupu):

```
...
<category>
    <pattern>OTEC</pattern>
    <template>Ne, to není pravda. To je nemožné.</template>
</category>
<category>
    <pattern>_ OTEC</pattern>
    <template><srai>OTEC</srai></template>
</category>
<category>
    <pattern>OTEC _</pattern>
    <template><srai>OTEC</srai></template>
</category>
<category>
    <pattern>_ OTEC * </pattern>
    <template><srai>OTEC</srai></template>
</category>
...
```

Obrázek 5 – zanesení klíčového slova

- Řízení výpočtu – rekurze a porovnávání vzoru dává autorovi robota k dispozici aparát, který ve spojení s další významným nástrojem jazyka, *predikáty* (které se navzdory svému označení a až na jednu poměrně důležitou vlastnost dají považovat za proměnné jako v běžných imperativních jazycích), dovoluje psát obecné programy.

  Základním prvkem imperativního programu je podmíněný příkaz. Takový, který jestliže je v predikátu *on* uloženo nějaké jméno, tak jej vypíše, a pokud je v něm uložena hodnota „NEZNÁMÉ“, tak zahlásí, že danou osobu nezná, je pomocí AIML implementován na obrázku 6 (nově představená značka *get* vrací hodnotu uloženou v predikátu specifikovaném v atributu name):

```
...
<category>
    <pattern>JAK SE JMENUJE</pattern>
    <template><srai>REKNIJMENO <get name="ON"/></srai></template>
</category>

<category>
    <pattern>REKNIJMENO *</pattern>
    <template>Jmenuje se <get name="ON"/>.</template>
</category>

<category>
    <pattern>REKNIJMENO NEZNÁMÉ</pattern>
    <template>Nevím co je zač.</template>
</category>
...
```

Obrázek 6 – simulace podmíněného příkazu

1.5. Další prvky šablony

Specifikace dělí prvky šablony do několik kategorií, které se označují podle generalizovaného efektu, které mají jejich značky při užití v kódu šablony. *Atomické prvky* nemají přímé potomky a dodávají místo sebe na místě, kde jsou aplikovány, nový obsah, a to z různých zdrojů. Kromě zmiňovaných zpětných referencí (star, thatstar, topicstar) jsou k dispozici odkazy na předchozí promluvu robota (*that*, tedy stejný název jako má značka v kategorii) a předchozí vstup konverzujícího uživatele (*input*). Mají také volitelný atribut s názvem index, který ovšem na rozdíl od svého jmenovce ve značkách s koncovkou -star dovoluje užití dvojice kladných celých čísel *(i, j)* oddělených čárkou. První číslo označuje i-tou předchozí větu (podle definice podané v části věnované předzpracování) a druhé j-té slovo v této větě. Vynechání druhého, resp. obou indexů interpret vyhodnotí jako užití indexu *(i, 1)*, resp. *(1, 1)*.

Další již zmíněnou značkou je get, která vrací hodnotu predikátu s názvem určeným jejím atributem name. Predikáty slouží jako pracovní registry robota, jejich hodnotu lze nastavit a přepsat značkou *set* (probraná dále). Pokud nebyl predikát s určeným názvem nastaven, vrací get prázdný řetězec. Podobnou funkci má značka *bot*, která vrací též předem nastavené hodnoty, ovšem s rozdílem, že ty její musí být dodané interpretu již před načítáním zdrojových kódů robota a nelze je za běhu měnit. Značka *bot* obvykle slouží autorům k uchování konstantních faktů týkajících se jejich robota. Je výjimečná tím, že se může vyskytovat nejen v šabloně, ale i ve vzoru *pattern* a *that*, kde je její obsah vyhodnocen už při načítání robota interpretem.

Přítomnost značky *bot* tak formálně rozděluje vzory na dva druhy: složené (mohou obsahovat značky *bot*, normální slova a zástupné znaky), které se ve zdrojovém kódu nacházejí pouze ve značkách *pattern* a *that* kategorií, a vzory jednoduché (bez značek *bot*), které se vyskytují ve všech ostatních instancích.

Méně významné jsou značky, jejichž obsah je dodáván interpretem bez ohledu na kód robota: *date* (místní datum a čas ve formátu dle vůle interpretu), *id* (nespecifikovaný identifikátor konverzujícího uživatele), *size* (počet nahraných kategorií) a *version* (číslo verze daného interpretu). Mezi atomické značky jsou počítány ještě *sr*, *person*, *person2* a *gender*. Zatímco značka sr je zjednodušením již z příkladů známé kombinace <srai><star/></srai>, zbytek představuje zkratku za obdobnou konstrukci <tag><star/></tag>, kde tag stojí za jednu z dosud neprobraných značek person, resp. person2, resp. gender v jejich neprázdné variantě.

Dalšími prvky jsou *prvky podmiňovací*. Patří sem vlastní prvek podmíněného příkazu *condition* (emulovaný ve výše uvedeném příkladu pro značku srai), jež se vyskytuje ve třech podobách (obrázek 7). V nejjednodušší formě obsahuje dva atributy, *name* a *value*. V případě, že text uložený v predikátu s názvem určeným name odpovídá jednoduchému vzoru popsaném ve value, je obsah prvku zpracován a výstup vrácen. V opačném případě je vrácen prázdný řetězec. V druhé formě obsahuje condition jen atribut name a atributy value jsou místo toho ve vnořených prvcích, položkách seznamu *li*. Jejich vyhodnocování probíhá v pořadí definice a ve výsledku je zpracován obsah prvního prvku li, jehož vzoru text z predikátu odpovídá. Volitelně může tato verze obsahovat jeden prvek li bez jakéhokoli atributu, a pak jeho obsah slouží jako výchozí možnost, jestliže všechny ostatní selžou. Konečně poslední forma condition neobsahuje žádné atributy a porovnávání predikátů a vzorů (se stejnými názvy atributů) je přesunuto do prvků li s jinak shodnými pravidly vyhodnocování jako u předchozí formy. Podmiňovací prvky jsou jen dva, tím druhým je *random*. Interpret při jeho vyhodnocení náhodně vybere z jeho synovských prvků li jeden (žádný jiný typ prvků přímo obsahovat nemůže) a jeho obsah je zpracován pro výstup.

```
...
<condition name="PREDIKAT" value="TESTUJICI VZOR">
    Text predikátu odpovídá testujícímu vzoru.
<condition/>
...
<condition name="PREDIKAT">
    <li value="TESTUJICI VZOR">Odpovídá testujícímu vzoru.</li>
    <li value="DALSI VZOR">Odpovídá až dalšímu vzoru.</li>
    <li>Ani jeden vzor neodpovídá.</li>
</condition>
...
<condition>
     <li name="PREDIKAT" value="TESTUJICI VZOR">Dvojice odpovídá.</li>
     <li name="DALSI PREDIKAT" value="DALSI VZOR">Až další odpovídá.</li>
     <li>Ani jeden pár predikát-vzor neodpovídá.</li>
</condition>
...
```

Obrázek 7 – tři formy podmíněného příkazu

Zejména při různých pomocných výpočtech a nastavování nachází uplatnění první z páru *neviditelných prvků*, a to prvek *think*. Obsah šablony obalený značkou think je sice normálně zpracován, nicméně výsledek zpracování není zobrazen a think vrací vždy prázdný řetězec. Druhým takovým prvkem je *learn*. Ten umožňuje načíst dodatečně do běžícího robota zdrojový soubor s dalšími kategoriemi. Zdroj specifikuje obsah prvku learn a musí být validní referencí URI[7].

Pro ukládání informací autorovi robota slouží *zachytávací prvky*. Protějškem get je prvek *set*, který uloží svůj zpracovaný obsah do predikátu určeného atributem name. Zvláštností prvku set je to, že standard připouští konfiguraci jeho výchozího chování. Běžně set po nastavení hodnoty vrací stejný řetězec na výstup. Interpret ale blíže nespecifikovaným prostředky může dovolit určit, pro které predikáty se má set chovat v duchu „vrať název, pokud jsi nastaven“. To dává příležitost umístit tento druh značek do zdrojového kódu v mnohem přirozenější podobě. Je patrné, že první alternativa je výrazně přehlednější než druhá (obrázek 8):

```
<template>
    Myslíš, že to udělal <set name="ON"><star/></set>? To bych od <get
name="ON"/> ale vůbec nečekal.
</template>

<template>
    Myslíš, že to udělal <think><set name="ON"><star/></set></think>on?
To bych od <get name="ON"/> ale vůbec nečekal.
</template>
```

Obrázek 8 – porovnání „vrať název, pokud jsi nastaven“ s běžným chováním

Pomocí prvku set lze navíc změnit aktuálně probírané téma, a to tak, že se uloží jeho popis do predikátu s názvem TOPIC. Tato technika dovoluje ovlivnit výběr kategorie nezávisle na uživatelském vstupu (pokud se nepočítá rekurzivní zpracování). V praxi méně využívaný zachytávací prvek se nazývá *gossip*. Vyhodnocený výraz v gossip je určen k uložení pro budoucí potřeby administrátora robota. Způsob a formát uložení je opět ponechán na vůli interpretu.

Transformační prvky *person*, *person2* a *gender* převádějí mezi mluvnickými kategoriemi. Značka person po vyhodnocení změní obsah v 2. osobě na text v 1. osobě a naopak. Značka person2 (navzdory svému názvu) prohazuje 3. a 1. osobu. Konečně gender operuje obdobným způsobem s mužským a ženským rodem. Implementace převodu je opět zcela na interpretu.

Formátovací značky *uppercase* a *lowercase* ve shodě se svým názvem převádějí text na velká, resp. malá písmena. Značka *formal* zvětší první písmeno každého slova, zatímco *sentence* provede to samé na začátku každé věty. Konce vět jsou u prvku sentence striktně definovány (oproti fázi rozdělování na věty) jako řetězce končící tečkou. Pokud text tečku neobsahuje, je celý považován za větu.

Přehled (při vynechání již probraného srai) uzavírají *prvky pro externí zpracování*. Jsou dva, prvních z nich se nazývá *javascript*, druhý *system*. Zatímco javascript je určen po předání obsahu stroji na interpretaci stejnojmenného skriptovacího jazyku, system je prostředkem pro volání programů z příkazové řádky běžícího systému. Pro obsah těchto prvků platí, že pokud jeho části dokáže interpret AIML rozpoznat jako validní obsah šablony nebo pokud tyto části nejsou uzavřeny ve standardní CDATA sekci jazyka XML[8], tak je před předáním externímu procesoru vyhodnotí jako běžný kód šablony. Prvky externího procesoru nejsou povinny vracet jiný výstup, než prázdný řetězec. Když interpret nemá vhodné procesory těchto značek k dispozici, může (ale nemusí) oznámit chybu ve zpracování.

1.6. Zdrojový kód

Témata a volné kategorie se dle standardu nacházejí výlučně v prvku *aiml*. Na druhou stranu připouští, že prvky aiml nemusí být nutně kořenovými prvky

vlastního XML dokumentu. Mohou být vloženy jako text do zdroje, který není XML dokumentem či se mohou vyskytovat v XML dokumentu jako jiný než kořenový prvek (i když specifikace přitom upozorňuje na scházející podporu pro rozlišení takto vložených objektů pomocí atributu id, jako je tomu např. u jazyka XSLT[9]). Pro souběžné užívání vlastních značek se značkami jiných jazyků AIML plně podporuje práci s jmennými prostory. Jmenný prostor AIML má URI http://alicebot.org/2001/AIML. Je zavedenou praxí vkládat do šablon např. fragmenty XHTML kódu za předpokladu, že jsou řádně označeny jako součást jiného jmenného prostoru. Také lze prvkům jazyka AIML přidávat nestandardní atributy, jestliže mají určený odlišný a neprázdný jmenný prostor.

2. Zhodnocení jazyka AIML

Specifikace ve svém úvodu předesílá následující cíle jazyka AIML:

- *Naučit se AIML má být lehké.*
- *AIML má kódovat minimální množinu konceptů k naplnění svého cíle. Tím je vytvoření reaktivního znalostního systému postaveném podle vzoru robota A.L.I.C.E. (to je první robot napsaný v AIML, resp. v jeho zárodečné podobě).*
- *AIML má být kompatibilní s XML.*
- *Má být snadné psát programy pro zpracování dokumentů AIML.*
- *Objekty jazyka by měly být čitelné pro člověka a v rozumné míře srozumitelné.*
- *Návrh jazyka má být formální a stručný.*
- *AIML nemá obsahovat části závislé na jiných jazycích.*

Jak dokládá druhý bod, jazyk AIML vznikl podle vzoru, jehož cílem bylo simulovat konverzaci s člověkem. Na tom, zda-li konverzace bude mít nějaký užitečný obsah, v případě A.L.I.C.E. příliš nezáleží, pokud je schopna smysluplně reagovat. Tento pohled je ale zcela odlišný od očekávání, které mají uživatelé a programátoři od dalších aplikací, ve kterých byl od svého vzniku AIML nasazen. V produktech, jako jsou interaktivní nápovědy či osobní asistenti, je sice také vhodné, aby měl uživatel pocit, že jedná s člověkem, ale hlavním měřítkem úspěchu je především úspěšné naplnění jeho potřeb. Právě zde se projevuje dědictví A.L.I.C.E, kdy AIML dává programátorovi zajímavé nástroje pro vytvoření robota, který bude za příznivých podmínek uživatele i autora překvapovat neočekávanými výstupy, ale již málo mu pomůže např. v zajištění toho, aby se robot uživatele zeptal na vše, co je třeba.

2.1. Srovnání s dialogovými systémy

Roboty napsané v jazyce AIML jsou typickými představiteli tzv. *chatterbotů*. Jejich ambicí je rozhovor pouze simulovat, aniž by potřebovaly porozumět obsahu. Mnohem pokročilejším druhem konverzačních agentů jsou *dialogové systémy*. Ty mají za cíl rozhovor přímo modelovat. A to včetně analýzy a

pochopení uživatelského vstupu s tím, že získané poznatky budou využity ke konstrukci přesnějších reakcí robota. Základní atributy, kterými se dialogové systémy vyznačují, jsou podle ([10] , s. 38) tyto:

1. *Díky dostupné programovatelné paměti si dovedou zapamatovat a aplikovat údaje získané během aktuální či některé z minulých konverzacích.*
2. *Nasazují prostředky logiky na modelování skutečného světa.*
3. *Dokážou více, než pouze napodobovat skutečný rozhovor, tím, že se snaží uživateli porozumět a vytvořit pomocí znalostní báze smysluplnou odpověď.*
4. *Vstup nezpracovávají pouze porovnáváním vzorů. Pro zlepšení jeho analýzy čerpají z rozmanitých konceptů, množin synonym, sémantických vztahů.*
5. *Odpovědi mohou být ovlivněny předchozími vstupy od uživatele.*
6. *Využívají správce dialogu k řízení konverzace podle minulých vstupů podle nějaké konkrétní strategie.*

Chatterboty, obecně jednodušší typ konverzačních agentů, naproti tomu mají následující vlastnosti, které většinou stojí v opozici vůči znakům dialogových systémů:

1. *Vstup zpracovávají výlučně pomocí porovnávání vzorů či hledání klíčových slov. Neužívají syntaktické, sémantické či pragmatické analýzy.*
2. *Obsahují předdefinované párů typu podnět – reakce. Mezi zpracováním vstupu a generováním odpovědi není žádný mezikrok.*
3. *Nejsou způsobilé k tomu, aby shromažďovaly a využívaly znalosti. Povědomí o věcech a inteligence je pouze simulována, a to přímo obsahem párů s podněty a reakcemi.*
4. *Jejich řízení dialogu je primitivní: Bez zohlednění kontextu očekávají vstup a poskytují na něj odpověď.*

Některé, zejména sofistikovanější chatterboty mají znaky dialogových systémů. I pokud se jako zástupci chatterbotů vezmou právě ty postavené na AIML (archetypální A.L.I.C.E. určitě naplňuje intuitivní představu chatterbota), je mimo jiné i z přehledu v 1. kapitole patrné, že jim nedělá obtíže si během rozhovoru ukládat a využívat získané údaje či dokážou pozměnit svou odpověď vlivem předešlých konverzačních výměn.

2.2. Výhody

Dialogové systémy jsou bezesporu mnohem mocnějšími a ve výsledku působivějšími aplikacemi. Z praktického hlediska však má AIML oproti těmto systémům výhody, které jej v důsledku mohou dle okolností dělat přijatelným, preferovaným či dokonce nadřazeným řešením:

Jak slibuje jeden z cílů ve specifikaci, programování v AIML je jednoduché. Vytvoření robota nevyžaduje expertní lingvistické znalosti. Jestliže na nějaký neočekávaný vstup dává neplatné odpovědi, je snadné jej triviálně rozšířit párem s doslovnou kopií vstupu a správnou reakcí s poměrně malým rizikem, že bude narušen chod robota v jiné situaci.

Vytváření robotů v AIML není náročné ani z pohledu softwarové podpory. Začátečník nepotřebuje žádné sofistikované databáze či podpůrný software. Stačí mu textový editor a interpret. Ideově nenáročná interpretace zdrojového kódu přitom způsobila, že vznikly implementace pro běžné platformy v mnoha programovacích jazycích[11]. Užití XML pak přináší obrovskou výhodu v podobě již existujícího, prověřeného a bohatého ekosystému nástrojů, které je možné ihned užít pro zpracování dokumentů v AIML.

Lze snadno nahlédnout, že navzdory své notaci je AIML Turingovsky úplným jazykem. Dokonce i bez ohledu na paměťové nároky, neboť specifikace neřeší paměťový model interpretu a neomezuje množství témat, kategorií, predikátů, délku identifikátorů či jiných prvků jazyka. (Každý Turingův stroj lze odsimulovat programem v AIML poměrně přímočaře tak, že se ztotožní stavy automatu s tématy a instrukce stroje se promění v kategorie, které místo pásky modifikují před rekurzivním zanořením vzor, případně nastavují nový stav změnou tématu. Počáteční stav pásky je určen textem uživatelského vstupu a výchozí stav předdefinovanou hodnotou predikátu TOPIC. Zakódování polohy hlavy na pásce do vzoru a vstupu lze provést pomocí znaků mimo užívané abecedy. Ty je vhodné užít i k označení kraje vzoru a zároveň popsané oblasti pásky.)

2.3. Možnosti rozšíření

Při hledání příhodných směrů, kterými by šlo usnadnit tvorbu a rozšířit

možnosti robotů v AIML, byly vzaty v úvahu jeho schopnosti, ilustrované v kapitole 1. Jako přiměřený rámec pro rozšíření byla adaptována myšlenka přiblížení se vlastnostem dialogových systémům. Na volbu měla vliv tato kritéria: míra zachování kladných vlastností platformy po změně, náročnost provedení a faktický přínos pro uživatele. Uvedené v pořadí, v jakém byly definovány znaky dialogových systémů, vyplynuly z analýzy následující závěry:

1. Jak již bylo zmíněno, AIML díky predikátům (které mají jistá omezení, více v kapitole 4.) dokáže uchovat a později přistupovat k informacím během konverzace. Bez možnosti náhrady ale postrádá prostředky pro trvalé uložení a užití získaných dat mezi sezeními. Prvek gossip (též představený v kapitole 1.) sice dovoluje ukládat data pro pozdější potřebu, ale již nespecifikuje jejich formát ani způsob načtení robotem. A tak se jedná o nástroj platný nanejvýše pro diagnostické potřeby administrátora robota. Přestože by zavedení permanentní paměti bylo zajímavým rozšířením, vyžadovalo by bohužel podporu na straně interpretu a odchýlení od standardu.
2. Aplikace logiky v AIML je obtížný, i když ne nutně neřešitelný problém. Kromě toho, že AIML v základu operuje přímo nad vstupními slovy, aniž by prováděl jakoukoli sémantickou analýzu, tak i samotné vzory pro zpracování vstupu jsou velmi omezené, pokud by jich bylo třeba k provádění logického odvozování. Jak vyplývá z přehledu v kapitole 1., v AIML jsou podporovány pouze vzory, které testují přítomnost všech uvedených slov v daném pořadí. Nic na tom nemění ani přítomnost zástupných znaků. Pokud má autor potřebu napsat vzor, který zachytává vstupy obsahující některé z uvedených slov (logické nebo) či slova v libovolném pořadí, lze toto pracně obejít pomocí vypsání všech možností do více kategorií, které se budou odkazovat na stejnou šablonu. Pro implementaci logické negace (vstup neobsahuje dané slovo) jen s pomocí porovnávání vzorů nedává AIML nad rigidním procesem porovnávání programátorovi dostatečný vliv (na rozdíl od řešení analogického problému v logickém programování pomocí tzv. řezu, např. v jazyku Prolog[12]). Další obtíží je slabá schopnost generování, která ztěžuje konstrukci odpovědi dle výsledku odvození.

3. Odhlédne-li se od poměrně nejasné definice toho, co znamená porozumět, tak AIML především ve svém kódu nerozlišuje mezi instrukcemi pro robota a daty. Tím se smazává hranice mezi odvozovacími pravidly a znalostní bází. Programování, rozšiřování a údržba robota je díky tomu náročnější, nemluvě o omezené znovupoužitelnosti kódu. Přestože zkušený programátor je schopen kód robota vhodně strukturovat, podpora od vývojových nástrojů jistě může být užitečným, byť nepříliš objevným rozšířením.
4. AIML není ani zdaleka navržen ke složitější analýze vstupu, přinejmenším bez vydatné pomoci interpretu, a i pak je veškerá informace, se kterou autor robota pracuje, omezena na celá normalizovaná slova. Není bohužel možné, aby docházelo v předzpracování k převodu na základní, jednodušší tvary, a tak se zredukovalo množství vzorů, které musí autor robota vytvořit. Důvodem je, že robot má přístup pouze k výsledku předzpracování, nikoli původnímu vstupu. A tak musí být v průběhu výpočtu do značné míry zachován původní tvar, neboť schopnosti generování jsou u značek v AIML (person, person2, gender) omezeny na prosté záměny jednoho tvaru za jiný. V této souvislosti je nutné připomenut, že AIML je i s náležitou podporou interpretu z obdobných důvodu obtížně aplikovatelný na syntetické jazyky (např. češtinu).
5. Proměnlivost odpovědí v závislosti na předchozích vstupech je vlastní mnoha běžným chatterbotům, a tedy i AIML[10]. Na rozmanitosti odpovědí AIML robota (po odmyšlení způsobů výběru odpovědi, probraných blíže v bodu 6.) se podílí změna hodnot predikátů, aplikace náhodného výběru a užití externích procesorů. Tuto nabídku lze považovat v tomto směru za dostatečnou.
6. Jak již bylo předesláno v úvodu kapitoly, možnosti řízení dialogu v AIML jsou silně omezené. Od zcela reaktivního chování se odchyluje užitím predikátů, řídících značek a především nastavováním témat konverzace. Tyto prostředky samotné ale nejsou s to implementovat komplikovanější strategii v řízení dialogu. Na druhou stranu představují společně se standardním porovnáváním vzorů dostatečně silné základní prvky na vytvoření správce

dialogu v rámci robota, zcela bez závislosti na předzpracování či speciálním chování interpretu, a dělají tak z řízení dialogu vhodného kandidáta na netriviální rozšíření.

3. Dostupné metody řízení dialogu

Přehledová publikace [13] v úvodu kapitoly o řízení dialogu (s. 23) popisuje správce dialogu jako *ústřední součást dialogového systému, jež obstarává interakci s uživatelem a externími zdroji znalostí*. Kromě toho ještě připomíná, že je na něj často nahlíženo jako souhrn dvou části:

- *vlastní řízení dialogu – určuje akce, které se mají provést po obdržení a interpretování vstupu od uživatele,*
- *modelování kontextu dialogu – zaznamenává, co bylo během konverzace vyřčeno, a ve světle toho interpretuje uživatelský vstup.*

Klasické přístupy k řízení dialogu jsou dva: užití *grafů* a užití *rámců*. Kromě toho se vyskytují hybridní řešení[14], řešení založená na plánování a jiných metodách AI či na statistickém modelování. Ta jsou ale až příliš vzdálená podmínkám AIML.

3.1. Grafy

Vlastní řízení dialogu založené na grafech zjednodušuje celý rozhovor na výčet jeho možných stavů a přechodů mezi nimi. Díky tomu je možné zredukovat chování agenta do elementárních kroků spojených s uzly a hranami grafu. Například pro robota, který má za úkol provést uživatele dotazníkem, mohou uzly představovat robotovy otázky, a hrany uživatelovy odpovědi. Obecně lze ale přiřadit prvkům grafu jakékoli smysluplné akce (např. dodatečné výpočty, komunikaci s ostatními komponentami dialogového systému, přístup do databáze). Přechody mezi stavy jsou pak determinovány obvykle uživatelským vstupem, proměnnými prostředí či jinými faktory.

Modelování kontextu dialogu je u grafových řešení z velké části implicitně poskytováno již samotnou strukturou grafu. Kromě toho mají správci k dispozici prostředky pro ukládání a získávání informací bokem. Jejich testování pak může být součástí podmíněného přechodu ze stavu do stavu.

Grafová řešení vynikají v situacích, kdy lze směřování konverzace dobře předpovědět. Díky pevnému pořadí se při generování odpovědí snadno využije toho,

co uživatel naposled prohlásil, dosazením do předdefinované šablony. Na druhou stranu v případě, že existuje variabilita v pořadí, ve kterém mohou etapy konverzace proběhnout, může bez dalších vylepšení značně narůst počet uzlů až za únosnou mez pro ruční návrh. Grafy přirozeně podporují dialog, ve kterém má iniciativu nikoli lidský protějšek, ale robot. Uživatel jen těžko může změnit téma či se vrátit zpět k dřívějšímu, aniž by na to bylo vysloveně pamatováno. Proto se zpravidla nehodí v aplikacích, kdy je součástí rozhovoru i jednání o jeho průběhu.

3.2. Rámce

Rámce jsou odpovědí na strnulou povahu grafů v tom, že středem zájmu není cesta k splnění všech cílů během rozhovoru, ale přímo cíle samotné. Řízení dialogu skrze ně lze připodobnit k vyplňování formuláře. Systém potřebuje od uživatele vyplnit všechny jeho kolonky (*sloty*), a tak pokud je nějaká prázdná, vyvolá to reakci v podobě zjišťovací otázky.

Pořadí, v jakém budou sloty zaplňovány, není specifikováno. Systém se může snažit zaplnit sloty otázkami v libovolném pořadí, např. v jakém je má definované. Podmínkou je, že jakmile se nějaký slot zaplní, neměl by systém uživatele dále obtěžovat opětovnými dotazy s cílem jej vyplnit. Kontext dialogu modelují samotné rámce. Reprezentace slotů přitom má obvykle podobu atribut – vyplněná či prázdná hodnota.

Jednou z hlavních výhod rámců oproti grafům je jejich schopnost pohotově reagovat na tzv. „over-answering“.[13] Např. dostane-li systém na otázku „Čím cestujete?“ odpověď „Lodí, z doku 94.“, neměl by se později již dotázat „Z jakého přístavního doku vyrážíte?“, přestože v danou chvíli nadbytečnou informaci nepožadoval. Něčeho takového není v grafech možné snadno dosáhnout, neboť jako odpověď na otázku „Čím cestujete?“ mají mezi alternativami pouze dopravní prostředky a nadbytečný výstup ignorují či je může dokonce zmást. Rámcový systém si s tím poradí a pouze vyplní dva sloty najednou.

Toto chování je na druhou stranu vykoupeno nutností složitějšího zpracování uživatelova vstupu, aby byly v jeho promluvě rozpoznány v různých permutacích požadované informace, často i s využitím sémantické analýzy.

Flexibilita rámců s sebou nese také komplikovanější výběr následující otázky, pokud má přirozeně reflektovat dosud nevyplněné sloty.

3.3. Z pohledu AIML

Na první pohled má technika rámců k AIML technicky i principiálně blíže. S pomocí predikátů jako záznamů o vyplnění slotů a za využití vzorů pro rozpoznávání klíčových termínu by se jí šlo poměrně snadno přiblížit natolik, nakolik to jen v omezujících podmínkách jazyka jde.

Naproti tomu řízení dialogu pomocí diagramu je diametrálně odlišné od reaktivní povahy AIML a jeho potenciální uvedení do světa AIML tak paradoxně představuje mnohem větší přínos. Zvolená implementace grafového přístupu, představená v následující kapitole 4. navíc některé jeho nedostatky (explozi stavů a omezené přepínání kontextů) úspěšně mírní.

4. Rozšířené přechodové sítě pro řízení dialogu v AIML

Ve své publikaci [15] přestavuje Paul Graham v kapitole 23. rozšířené přechodové sítě RPS, původně uvedené v [16], takto: *„Přechodová síť je množinou uzlů pospojovaných navzájem orientovanými hranami – v podstatě vývojový diagram. Jeden uzel je určen jako výchozí a některé další pak jako koncové. Hrany mají přiřazené podmínky, které musí být splněny před tím, než po nich může výpočet pokračovat. Vstupem do sítě je věta a ukazatel na slova věty. Postup po některých hranách posouvá ukazatel směrem ke konci věty. Syntaktická analýza věty pomocí přechodové sítě pak odpovídá nalezení cesty z výchozího uzlu do některého z koncových, po níž lze všechny podmínky uložené hranami cesty splnit.*

RPS mají oproti tomuto modelu navíc tyto vlastnosti:

1. *RPS mají registry, paměť pro odkládání informací v průběhu analýzy. Hrany mohou kromě testů mohou obsah těchto registrů modifikovat.*
2. *RPS jsou rekurzivní. Hrany si mohou klást jako podmínku přechodu po nich to, že výpočet úspěšně projde nějakou podsítí.“*

RPS díky možnosti rekurze dovolují konstruovat mnohem úspornější analyzátory. Využívají pravidelností jazyka a opakované postupy izolují do vlastních podsítí. Také umožňují odložit některá rozhodování při analýzu až na dobu, kdy bude o struktuře věty známo více. Své uplatnění nalezly i v jiných oborech umělé inteligence. Jednou z takových aplikací je právě řízení dialogu.

4.1. Adaptace pro řízení dialogu

Jako inspirace pro nasazení RPS k řízení dialogu v AIML robotech posloužily tzv. Dialogue Action Forms (DAF), užité k reprezentaci doménových znalostí v části projektu Companions[17]. Vedení konverzace je modelováno průchodem rozšířenou přechodovou sítí, jejíž uzly představují okamžiky výpočtu reakce konverzačního agenta, ve kterých má na výběr z více alternativ. Tyto alternativy jsou představovány odchozími hranami. S každou z nich je spojen test, který je vyhodnocen před tím, než se provede přidružená akce a zpracování se přesune do koncového uzlu hrany. Na rozdíl od obecných RPS má smysl rozlišovat

různé druhy uzlů, jež budou mít dopad na průběh dialogu. Kromě *běžných uzlů*, po jejichž dosažení je ihned vybrána jedna z odchozích hran a výpočet pokračuje dál, je nezbytné mít k dispozici *uzly blokující*, v nichž je výpočet přerušen, aby mohl být přijat vstup od protějšku v diskurzu.

RPS je vlastní nedeterministický průběh výpočtu, který jim pomáhá překonávat nejisté kroky při parsování věty tak, že uhodne správný přechod. Obvykle je nedeterminismus simulován backtrackingem.[15] V intencích řízení dialogu je ale simulace nedeterminismu složitější, neboť průběh výpočtu mohou doprovázet viditelné postranní následky v podobě vstupu a výstupu. Ty nelze napravit v případě, že se výpočet nedostane do koncového uzlu sítě a je nutné jej částečně vrátit zpět. Cestou, jak vyřešit tento rozpor, je postranní efekty zanedbat a nenahlížet na řízení dialogu jako na výpočet správných odpovědí robota, ale spíše jako na proces hledání cesty vedoucí k naplnění aktuálního cíle konverzace, koncového uzlu.

V ideálním světě by systém řízení dialogu uhodl nejlepší cestu, kterou se má konverzace ubíhat, bez ohledu na uživatelský vstup. Pokud se to ale nepovede, dochází k backtrackingu do té doby, než je cíl naplněn a nebo byly vyčerpány všechny možnosti. Pak je legitimní pro řízení dialogu ohlásit neúspěch (i když je vhodnější na takové eventuality pamatovat, a připravit si náhradní řešení v podobě únikové cesty). Pro potřeby modelování dialogu tedy hraje roli, v jakém pořadí budou zkoušeny odchozí hrany uzlu. Proto existuje motivace specifikovat v jejich počátečním uzlu strategii postupného výběru. Rozumným požadavkem navíc je, aby byl dostupný náhodný výběr, pro přirozenější reakce ve scénářích, kdy se průběh dialogu dostane do smyčky.

Stejně jako v aplikacích využívajících DAF[18], je i v navrhovaném systému nakládáno s rekurzivním charakterem RPS tak, že jsou vnořené sítě přidávány na zásobník, který lze na začátku výpočtu předvyplnit sítěmi, kterými se má dle záměru programátora projít. Jakmile dojde výpočet v síti na vrcholu zásobníku do koncového stavu, je tato síť odstraněna a výpočet pokračuje sítí umístěnou pod ní, dokud nedojde k vyčerpání zásobníku. To signalizuje úspěšné ukončení konverzace.

4.2. Cesta k implementaci

Tento proces byl výzvou díky následujícím faktorům:

- Pro využití stávajících interpretů bylo nutné všechny provozní části napsat v čistém kódu jazyka AIML.
- Jazyk postrádá podporu aritmetiky.
- Predikáty nelze porovnávat vůči sobě, ale vždy pouze oproti konstantě.
- Implementace by měla spoléhat především na základní konstrukty a procesy jazyka (u nichž lze očekávat lepší výkon v interpretech), než na simulaci imperativního programu v těle šablon.
- Z praktického hlediska je dobré se vyhnout extrémnímu rekurzivnímu zanořování přes srai.
- Návrh by měl být dostatečně robustní, aby odolal efektům uživatelského kódu v hranách sítě.

Pro vývoj měl zásadní význam fakt, že nastavení témat je prostředkem, jak ovlivnit porovnání vzorů, který nezávisí na aktuálním uživatelském vstupu. To z něj činí přirozeného kandidáta na životaschopnou implementaci uzlů i hran. AIML navíc poskytuje v kombinaci

- zástupného znaku hvězdička,
- značky set, která nastavuje hodnotu predikátu TOPIC (a tím i téma) a
- značky topicstar, která dovoluje přistoupit k části zachycené vzorem tématu,

vše potřebné pro zkonstruování zásobníku se základními operacemi. Testování hodnoty na vrcholu zásobníku s nimi probíhá následovně (obrázek 9):

```
<topic name="HLAVA *">
    <category>
        <pattern>*</pattern>
        <template>Na vrcholu zásobníku je HLAVA.</template>
    </category>
</topic>
```

Obrázek 9 – testování vrcholu zásobníku

Přidání prvku na vrchol zásobníku lze provést způsobem z obrázku 10:

```
<topic name="*">
    <category>
        <pattern>*</pattern>
        <template>
            <set name="TOPIC">HEAD <topicstar/></set>
        </template>
    </category>
</topic>
```

Obrázek 10 – přidání vrcholu na zásobník

A konečně odstranění libovolného vrcholu je možné díky tomu, jak jsou zachycována slova sousledností zástupnými znaky (blíže v kapitole 1.), vyřešit takto (obrázek 11; lze kombinovat s podmíněným odstraněním, jestliže je na vrcholu určitý stav, podle příkladu výše):

```
<topic name="* *">
    <category>
        <pattern>*</pattern>
        <template>
            <set name="TOPIC">
                <topicstar index="2"/>
            </set>
        </template>
    </category>
</topic>
```

Obrázek 11 – odstranění vrcholu ze zásobníku

Uvedená řešení předpokládají, že je na dně zásobníku vždy umístěn pomocný prvek, který nebude nikdy odstraněn, jinak by selhalo porovnávání. Hvězdička musí totiž, pokud se nevyskytuje ve vzoru samotná, zachytit vždy alespoň jedno slovo. Pohyb po síti pak probíhá dle návrhu následovně:

1. Po zadání vstupu uživatelem dojde k jeho odeslání interpreteru jazyka AIML.
2. Ten při hledání vhodné kategorie s odpovědí srovnává (mimo jiné) aktuální téma oproti vzoru jejich tématu. Tedy volbou tématu lze přímo ovlivnit, jaká kategorie bude vybrána.
3. Porovnání vrátí takovou kategorii, jejíž vzor tématu (vždy tvořený názvem stavu, který kategorie reprezentuje, a hvězdičkou) odpovídá situaci na zásobníku. Tedy např. kategorie se vzorem tématu „STAV *“ je aktivována, jestliže obsah predikátu TOPIC začíná normálním slovem „STAV“ neboli „STAV“ je vrcholem zásobníku. Na konci šablony každé kategorie se stavem musí být bez ohledu na to, zda-li jde o uzel či hranu, přítomen kód, který

v důsledku odebere navštívený stav ze zásobníku tím, že normální slovo „STAV“ nepřidá k hvězdičkou zachycenému zbytku zásobníku při nastavování nového obsahu predikátu TOPIC.

4. Pokud je aktuální stav uzlem, jsou na zásobník šablonou jeho kategorie postupně přidány stavy, které reprezentují odchozí hrany. Pokud je aktuální stav hranou, je po splnění jí předepsaného testu přidán na zásobník koncový uzel hrany. Pokud test není splněn, nepřidá se na zásobník nic a dochází k backtrackingu přes další stavy na zásobníku.

Takto načrtnutá implementace nemá speciální zásobník pro sítě, neboť ten je bohužel k dispozici díky speciálnímu chování predikátu TOPIC jen jeden. Síť se podle ní na zásobníku nachází tehdy, jestliže se na něm nachází některý z jejích výchozích uzlů a opouští jej (u vícenásobného vložení její později navštívená instance) tehdy, pokud se výpočet dostane do některého z jejích koncových stavů nebo jsou ze zásobníku při backtrackingu odstraněny všechny její uzly a hrany.

Kvůli explicitnímu generování následných stavů, které je nezbytné pro elegantní zapojení do procesu porovnávání vzorů s uživatelským vstupem, vyžaduje úspěšný průchod sítí od výchozího do cílového uzlu, aby byly uklizeny stavy na zásobníku, které nepřišly na řadu. To se provede tak, že při dosažení cílového stavu je na zásobník přidána speciální značka, která přepne výpočet do režimu úklidu. Při něm jsou postupně odstraňovány stavy pod značkou do té doby, až výpočet dorazí k jejímu protějšku. Pak jsou obě pomocné značky odstraněny a pokračuje se zpracováním následujícího stavu. Práce s pomocnými značkami využívá stejné techniky jako základní operace se zásobníkem.

4.3. Uspořádání odchozích hran

Botníček podporuje dva druhy uspořádání odchozích hran uzlu. Prvním způsobem je prosté uspořádání podle klesající nezáporné celočíselné *priority*, což je jedna z dodatečných vlastností, kterou lze pro každou hranu specifikovat. Při průchodu sítí do hloubky tedy budou hrany zkoušeny v pořadí dle klesajících priorit, které určí autor robota. To lze zařídit snadno již ve fázi generování tak, že se hrany seřadí, a jejich názvy se pak oddělené mezerami vypíší jako obsah, který má být

přidán na zásobník v jejich počátečním uzlu. Druhým způsobem, jak hrany vybrat, a který během opětovného průchodu může podávat přirozenější výsledky, je náhodné uspořádání. Botníček tehdy zohledňuje priority též, a to tak, že při zkoušení je pravděpodobnost vybrání hrany rovna podílu priority hrany a součtu priorit všech dosud nevyzkoušených hran. Z podstaty věci nelze náhodný výběr generovat dopředu, a je nutné jej naprogramovat přímo v AIML.

Při tom se nelze obejít bez značky random. Ta dovoluje vybrat náhodně jeden ze svých potomků. Přestože to specifikace jasně nevymezuje, lze oprávněně předpokládat, že tento náhodný výběr má rovnoměrné rozdělení v kontextu opakovaného průchodu identickým prvkem. Pro zohlednění priorit v AIML byl vyvinut následující algoritmus, který rozdělení transformuje, byť za cenu časové složitosti úměrné *nm*, kde *n* je počet odchozích hran a *m* je maximální povolená priorita:

1. Při potřebě rozvržení hran se výpočet přepne do zvláštního stavu pro náhodný výběr a k rekurzivnímu zpracování vstupu (značka srai) se předá seznam názvů všech hran oddělených mezerami, kde je každý název za sebou uveden v tolika kopiích, jak je velká priorita dané hrany. Jestliže je priorita rovna nule, není název uveden ani jednou, a tak je hrana ignorována.
2. Seznam názvů je porovnán vůči vzoru, který každé jednotlivé slovo seznamu zachytí pomocí zvláštního zástupného znaku hvězdička. Ještě předtím je ale obalen z obou stran pomocnými slovy, který zajistí, že se seznam nachází ve správné fázi zpracování (RANDOMSTART, RANDOMEND). Např. seznam hran „RANDOMSTART JEDNA JEDNA TRI OSM RANDOMEND“ je zachycen pomocí vzoru „RANDOMSTART * * * * RANDOMEND“.
3. Pokud se další krok ilustruje výše uvedeným příkladem, na zachycená slova lze odkazovat značkou star s indexy 1 až 4. Uvnitř prvku random pak lze provést náhodný výběr s rovnoměrným rozdělením jedné z těchto čtyř referencí, která bude vrácena na výstup.
4. Zvolená a zbylé tři reference jsou předány proceduře na smazání duplikátů vybraného názvu hrany tak, že jsou z obou stran doplněny pomocnými slovy REMOVESTART, resp. REMOVEEND a předány značce srai. Tato

procedura je schopna odstranit ze zbytku dodaného vstupu opakování prvního zadaného slova jakoby šlo o operaci nad spojovým seznamem. Nepříjemným důsledkem toho, že AIML nedovoluje porovnat dvě nekonstantní hodnoty, je nutnost vygenerovat pro každou takovou hranu kopii této procedury, jež bude mít odstraňovaný název napevno zadaný ve vzoru. To bohužel přispívá k celkové velikosti zdrojového kódu při exportu.

5. Po odstranění duplicit je ze zbylých hran rekurzivně vybírána následující do té doby, než se dojde na konec. Výstupem hlavní procedury je tak seznam hran přesně odpovídající definici. To, že byl pro potřeby přepnutí do této procedury smazán zásobník, nehraje roli, neboť jeho kopie je v místě volání stále dostupná pod značkou topicstar, pomocí které je navrácen do původní podoby jako v běžných stavech.

4.4. Realizace testovacích hran

Botníček podporuje několik druhů testů u hran sítě. Zvláštní postavení mezi nimi má typ testu, který je AIML vlastní, tj. porovnání vzoru. Ten je přímo součástí definice tohoto typu hrany hrany. V tématu, které takovou hranu reprezentuje, se nachází obvykle dvě kategorie. První z nich obsahuje programátorem specifikované testovací vzory pattern a that a v šabloně kód, který se má provést v případě úspěchu. Druhá z kategorií je návratová. Její vzory pattern a that jsou tvořeny hvězdičkami, které díky pořadí vyhodnocování přijdou na řadu až tehdy, když neprojde vstup vzorem, který specifikoval autor robota. V šabloně návratové kategorie se neděje nic jiného, než že se odstraní název hrany ze zásobníku. Jedinou nesnází je, že uživatel může specifikovat testovací vzor stejně obecně jako návratovou kategorii, jen se samými hvězdičkami. Pak je nutné kód návratové kategorie vynechat, aby se při načítání do interpreta navzájem nepřepsaly (a tím by se ztratil kód prováděný v případě úspěchu) nebo dokonce nezpůsobovaly chybová hlášení kvůli shodné vstupní cestě.

Dalším na implementaci náročnějším druhem hrany je ta, jež vychází z definice RPS: rekurzivní zanoření do sítě, které uspěje tehdy, když v podsíti výpočet dojde do cílového stavu. V navrženém řešení ke své funkci potřebuje

dokonce celá dvě témata, každé má po jedné kategorii. V prvním z nich, jehož vzor je stejný jako u jakékoli jiného stavu, se na zásobník nejprve přidá název hrany a nad něj pomocná značka pro návrat, aby se mohl po úspěšném projití podsítě výpočet vrátit zpět. Teprve po nich přijde na řadu vybraný výchozí vrchol cílové sítě. Druhé téma má oproti ostatním stavům nikoli dvouslovný, ale tříslovný vzor, sestávající se ze speciální značky úspěšného dosažení cíle, opět názvu této rekurzivní hrany a obvyklé hvězdičky na konec.

Vynořování pak probíhá takto:

1. Po dosažení cíle v podsíti se na zásobníku za sebou nachází úklidová značka po dosažení cílového uzlu podsítě, její protějšek pro vstupní uzel podsítě a značka pro návrat.
2. V této konfiguraci jsou tyto tři nahrazeny značkou úspěšného projití.
3. Značka úspěšného projití a pod ní nacházející se název sítě jsou zachyceny a provede se kód v druhém tématu rekurzivní hrany. Na jeho konci je na zásobník vložen koncový vrchol rekurzivní hrany.
4. Neúspěšné projití podsítě je detekováno tak, že se na vrchu zásobníku sejde jen úklidová značka výchozího uzlu podsítě a návratová značka. Tehdy je níže umístěný název rekurzivní hrany ze zásobníku odstraněn a pokračuje se stavem pod ním, tedy jde o běžný backtracking.

Ostatní navrhované typy hran, které vychází z možností AIML, jsou již realizované především na straně kódu šablony, v rámci jednoho tématu – názvu hrany s hvězdičkou. Zatímco testování výstupu AIML kódu se musí spolehnout na prozatímní uložení do pomocného predikátu, jehož hodnota je teprve porovnávána se vzorem, tak přímé testování hodnoty predikátu či hrana, jejíž test vždy projde, mají implementaci odrážející přímo definici.

4.5. Nakládání s uživatelským vstupem

Zvláštní pozornost zaslouží vysvětlení toho, co se stane s řetězcem, který je zadán jako vstup po přerušení v blokujícím uzlu. Díky zachytávání zástupnými znaky, odkazování pomocí -star značek a předávání skrze srai vstup probublává sítí do té doby, dokud výpočet probíhá v rámci nepřerušujících (procesních) uzlů. Díky

tomu je dostupný nejen pro jakoukoli hranu testující vzor, která se na cestě nachází, ale po uložení reference do predikátu je možné se podle něj rozhodovat i v jiných typech hran. Jakmile se proces řízení dialogu octne v blokujícím uzlu, je zachycen a předáván vstup nově zadaný. Předchozí vstupy jsou i nadále přístupné aplikací značek input, ale není radno jich při programování užívat, neboť vnášejí do sítě nadbytečnou závislost na přesném průběhu výpočtu, kterou je lepší popsat přímo strukturou sítě.

4.6. Změna kontextu

Pro DAF je typická jedna vlastnost, kterou RPS postrádají, a která je při řízení dialogu vhodná k neplánované změně probíraného tématu (myšleno téma konverzace, a nikoli jako klíčové slovo AIML): síť může být aktivována nejenom tím, že se nachází na vrcholu zásobníku, ale také rozpoznáním nějakého indexovaného termínu, který byl pro ni definován. Tato funkcionalita není v aktuální verzi Botníčku bohužel ještě přímo podporována, mimo jiné proto, že by podstatně stěžovala odlaďování. Pokud je ale uživatel obezřetný, může tento nedostatek snadno obejít přidáním vlastního souboru k těm, jež mu prostředí vyexportuje. Ten bude obsahovat požadované indexy podle vzoru na obrázku 12:

```
<topic name="_">
    ...
    <category>
        <pattern>_ TERMIN *</pattern>
        <template>
            <set name="TOPIC">SIT <topicstar/></set>
            <srai>
                <star index="1"/>VYNECHAN CI JINA SLOVA <star index="2"/>
            </srai>
        </template>
    </category>
    ...
</topic>
```

Obrázek 12 – přepínání kontextu mimo pořadí

Všechny indexy budou umístěné v tématu, jež má prioritní vzor „_“. To zajistí, že v případě hledání kategorie bude interpret přinucen začít jimi místo toho, aby se pokoušel přejít do kategorie s dalším stavem aktuální sítě. Do vzoru pattern se zadá klíčové slovo, pro které se má aktivovat cílová síť. První řádek pak nastaví na

vrchol zásobníku (a tedy jako další navštívený stav) výchozí uzel cílové sítě, specifikovaný uvnitř značky set. Následující rekurzivní prvek srai pak do něj předá jako vstup původní vstup. Z toho je ovšem před tím třeba odstranit či nahradit klíčový termín, jinak by došlo k opětovné aktivaci indexující kategorie.

5. Uživatelská dokumentace vývojového prostředí

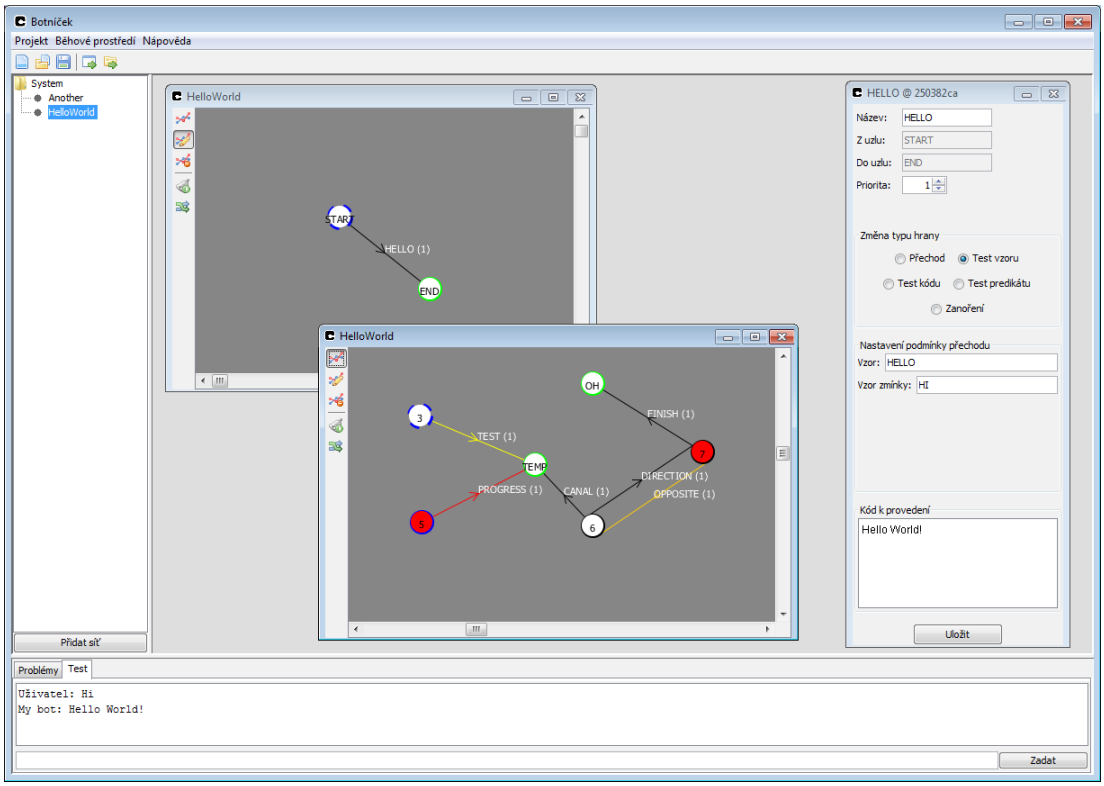
Aplikace ke svému běhu vyžaduje nainstalované běhové prostředí platformy Java (JRE) alespoň ve verzi 1.7 (obchodně označované jako Java 7), které je v oficiální distribuci dostupné pro běžné operační systémy z webových stránek https://www.java.com/download/. Program by měl ovšem pracovat i v alternativních strojích cílených na verzi 7.

5.1. Instalace

Uživatel má k dispozici multiplatformní instalátor s názvem botnicek-installer.jar, jež vyžaduje stejně jako samotný program dostupné JRE v odpovídající verzi. Alternativně lze program bez instalace spouštět přímo ze spustitelného archivu botnicek.jar, neboť pro svůj běh nevyžaduje žádné dodatečné kroky.

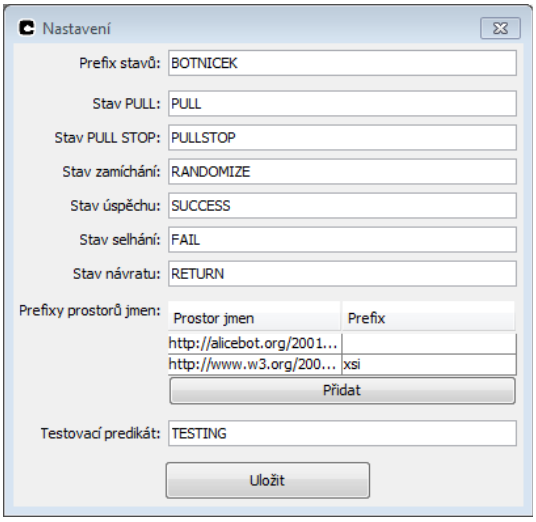
5.2. Vytvoření nového projektu

Po otevření aplikace (obrázek 13) lze nový projekt vytvořit pomocí menu a položky Projekt → Nový. (Kromě myši Botníček plně podporuje, až na samotný grafický návrh systému, i ovládání skrze klávesnici včetně klávesových zkratek. Ty jsou uvedeny u funkcí v menu. Např. pro založení nového projektu stačí tedy stisknout Ctrl+N.) Program pak vyzve k zadání názvu projektu. Názvy projektu a vytvářených sítí Botníček omezuje (z důvodu kompatibility napříč systémy) na množinu znaků tvořenou písmeny anglické abecedy, arabskými ciframi a podtržítkem. V novém projektu se již nachází jedna vzorová síť s názvem HelloWorld, kterou může uživatel eventuálně snadno smazat jejím vybráním a stiskem klávesy Delete.



Obrázek 13 – prostředí aplikace

Po vytvoření je na místě upravit výchozí nastavení projektu přes Projekt → Nastavení (obrázek 14). Nejdůležitější zde se nacházející položka se nazývá „Prefix stavů“. Obsahuje neprázdný řetězec, který bude přidán na začátek všech provozních stavů. Toto opatření dovoluje vyhnout se konfliktům s názvy provozních stavů v situaci, kdy je toto prostředí využíváno jako pomůcka pro vytvoření pouze části databáze robota. Pouze musí autor robota zajistit, aby byl prefix natolik unikátní, že nebude docházet ke kolizím s názvy témat v cizích souborech. Následující textová pole jsou právě názvy pracovních stavů (bez prefixu), která si Botníček vyhrazuje pro funkci jím vygenerovaných robotů (jejich význam je probrán v kapitole 4.). Konflikty cizích a uživatelských stavů (názvů uzlů a hran) vyloučeny nejsou, neboť ty nejsou opatřeny prefixem, aby bylo možné jejich názvy snadno odečíst ke chtěnému napojení z kódu vytvořeného jinde, mimo podporu řízení dialogu.

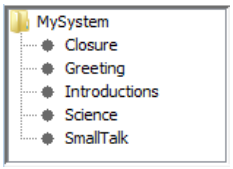


Obrázek 14– dialog nastavení projektu

Dále se ve formuláři vyskytuje tabulka registrovaných prefixů XML značek s definicemi jmenných prostorů. Povinnými a již přítomnými položkami jsou jmenný prostor značek jazyka AIML a určení XML schématu. Vlastní prefixy a prefix AIML pak lze užít u značek ve všech polích pro editaci kódu šablony, a tak rozlišit vložené jazyky pro potřeby validace i interpretace. Po změně hodnot je nutné nové hodnoty uložit a pak teprve okno zavřít. Po úvodním nastavení lze doporučit si celý projekt uložit přes Projekt → Uložit.

5.3. Správa systému sítí

Na levé straně pracovního prostředí se nachází přehled systému sítí (obrázek 15). Nová síť se přidá dvojitým poklepáním na ikonku vedle názvu systému nebo dedikovaným tlačítkem na spodní hraně přehledu. Botníček uživatele pak vyzve k zadání názvu nové sítě, jež by měl být odlišný od názvu sítí již v systému přítomných (vstup je zde i v jiných obdobných instancích ověřován a na případné prohřešky je uživatel upozorněn chybovým dialogem). Po potvrzení se pak nová síť objeví v abecedně seřazeném přehledu. Dvěma poklepáními na popisek s vloženou prodlevou lze vstoupit do režimu úpravy názvu sítě či systému. Pro skončení editace stačí stisknout klávesu Enter.



Obrázek 15– přehled sítí

Jak lze ověřit i na výchozí HelloWorld, tak sítě lze mazat označením a stiskem klávesy Delete. Dvojitým poklepáním bez prodlevy na ikonku či popisek sítě dojde k otevření nového vnitřního okna s plochou, na které se provádí vlastní návrh sítě. Poslední funkcí přehledu je, že při označení dané sítě je přesunuto jedno z jejích otevřených oken do popředí.

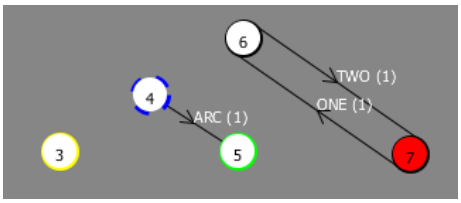
5.4. Přidávání, úprava a odstranění uzlů

Uzly lze v otevřeném okně s pracovní plochou sítě přidávat dvojitým poklepáním na prázdné místo kdekoli na ploše. Po umístění jsou vyznačeny kroužky (obrázek 16). Nově přidaný uzel má přidělen unikátní název v celém systému sítí. Všechny uzly jsou přidávány jako izolované, a takové zůstanou do okamžiku, než se uživatel rozhodne je připojit k nějakému jinému hranou. Izolované uzly (obrázek 16, uzel 3) jsou vyznačeny se žlutým okrajem a z hlediska užití při výpočtu nejsou příliš zajímavé, protože jsou z definice nedostupné.

Přepnutím módu okna sítě pomocí tlačítka s tužkou v levém sloupci či stisknutím klávesy Shift a následným dvojitým poklepání lze jakýkoli uzel přejmenovat. Uživatel je povinen užít unikátní jméno, které odpovídá definice normálního slova v jazyce AIML (tedy obsahuje pouze velká písmena a číslice, každý prohřešek proti velikosti písmen je v tichosti napraven). Všechny typy uzlů může uživatel dvojitým poklepáním přepínat mezi typem procesním a vstupním. Podmínkou je podržení klávesy Ctrl ve výchozím návrhovém módu okna či přepnutí do příslušného režimu tlačítkem se zvonkem. Vstupní typ má vnitřek kruhu červený (obr. 16, uzel 7), kdežto výchozí procesní typ jej má bílý (obr. 16, uzly 3, 4, 5 a 6). Ve shodě s popisem v kapitole 4. se vstupní typ uzlu vyznačuje tím, že vždy, když se dostane výpočet do tohoto stavu, tak je přerušen a pokračuje až tehdy, když uživatel přispěje (další) promluvou do konverzace.

Další proměnlivou vlastností uzlů je přístup k uspořádání odchozích hran během výpočtu. Pokud je zvoleno uspořádání dle priority (vyznačeno přerušovaným okrajem, obr. 16, uzel 4), jsou hrany seřazeny sestupně podle své priority. Priorita je celé nezáporné číslo. Při dalším postupu sítí se zkouší jako první hrana s nejvyšší prioritou, jako druhá s druhou nejvyšší atd. Pokud je zvoleno vážené náhodné uspořádání (vyznačeno hladkým okrajem, obr. 16, uzly 3, 5, 6 a 7), mají odchozí hrany pro každý průchod uzlem proměnlivé pořadí. Priorita má pak efekt takový, že náhodný výběr s rovnoměrným rozdělením není prováděn jen z množiny dosud nevyzkoušených odchozích hran, ale výskyty hran jsou pro okamžik výběru znásobeny podle své priority. Nastavení nulové priority hranu pak vyřazuje z výběru zcela. Způsob uspořádání odchozích hran nemá smysl pro izolované a koncové uzly, které odchozí hrany postrádají. Proto u nich ani nejde nastavit. Pro ostatní případy se nastavení provádí podržením klávesy přepínače Alt či přechodem do příslušného módu tlačítkem s šipkami a dvojitým poklepáním

Jak začne uživatel některé uzly spojovat hranami, bude se měnit barva jejich okraje. Tato barva indikuje jejich umístění v přechodové síti. Modrý okraj rámuje vstupní uzly (obr. 16, uzel 4). Vstupní uzly jsou takové, jež mají alespoň jednu odchozí hranu a žádné příchozí. Tyto uzly slouží jako brány do dané sítě a lze se na ně odkazovat, a tak zanořovat výpočet z později představených hran s rekurzivním testem. Zelená barva obroučky značí uzly koncové (obr. 16, uzel 5). Do koncových uzlů hrany pouze směřují a musí být alespoň jedna taková. Jestliže se výpočet dostane za běhu do tohoto uzlu, je považován za v dané síti uzavřený a nedochází k dalšímu backtrackingu přes dosud nevyzkoušené cesty. Posledním druhem uzlů podle umístění jsou uzly vnitřní, označené černým okrajem (obr. 16, uzly 6 a 7). Do nich přichází a z nich odchází v obou orientacích alespoň jedna hrana.



Obrázek 16 – typy uzlů

Další vlastností uzlu, kterou jde nastavit, ale která nemá vliv na průběh

výpočtu, je *poloha na ploše sítě*. Uzel lze přesouvat stiskem a podržením primárního tlačítka myši nad jeho grafickou reprezentací. Přesun je ukončen povolením tlačítka nad cílovým místem. Uzly jde též smazat, a to stisknutím kláves Ctrl a Shift najednou a dvojitým poklepáním na uzel (či lze místo užití kláves přejít pro tuto operaci do zvláštního módu tlačítkem se znaménkem minus v červeném kruhovém poli). S uzlem se odstraní i všechny navazující hrany i se svými nastaveními. Proto program vyžaduje před tímto krokem potvrzení.

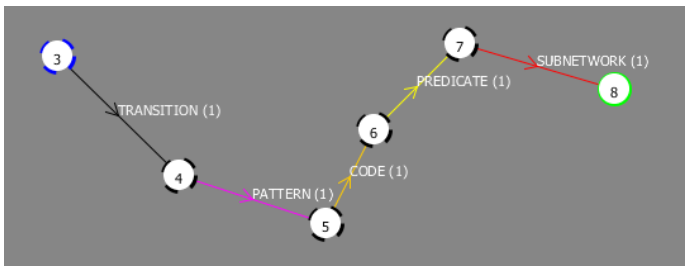
5.5. Přidávání, úprava a odstranění hran

Hranu lze vést mezi libovolnými dvěma uzly. Vytvoření začíná dvojitým poklepem na potenciální počáteční uzel ve výchozím módu návrhu. Poté je mezi kurzorem myši a počátečním uzlem znázorněna spojnice budoucí hrany. Druhým dvojitým poklepem na jiný než počáteční uzel je hrana vytvořena a uživatel vyzván k zadání jejího názvu (opět normální slovo v AIML). Pokud se uživatel rozhodne v průběhu návrhu spojení operaci přerušit, stačí mu klepnout na volnou pracovní plochu sítě. Název hrany by měl být též unikátní v rámci všech hran i uzlů systému. U duplicit prostředí zakročí a doplní název na unikátní místo toho, aby uživatele nutilo začít navrhovat hranu od začátku.

Nově vytvořená hrana má výchozí prioritu a neklade si žádné podmínky přechodu. Její nastavení lze upravovat ve speciálním okně (obrázek 13 napravo), jehož nová instance se otevře po dvojitém poklepání na blízké okolí spojnice uzlů. Obsah tohoto okna se liší podle toho, o jaký typ hrany se jedná. Na vrcholu okna se ale vždy nachází část společná pro všechny typy. Je zde pole pro nastavení názvu, pro zobrazení názvů krajních uzlů a volič priority. Pod nimi se nachází přepínače typu hrany, které kromě nastavení typu mění obsah prostřední části okna, který je pro daný typ specifický.

Ve spodní části okna mají všechny typy ještě společnou oblast pro zadání AIML kódu k provedení v případě, že je podmínka přechodu po hraně splněna. Tato oblast a jí podobné přijímají a zvýrazňují validní kód šablony kategorie jazyka AIML (tedy zde lze užít vedle prostého textu všechny značky šablony). Jakékoli úpravy nastavení hrany je třeba uložit stiskem tlačítka v patičce okna. U všech oken v

pracovním prostředí návrhu lze změnit jejich rozměr. Zatímco okna se znázorněním sítě se změnou velikosti okna mění pouze výřez pohledu na plochu sítě, u oken s podrobnostmi hran dochází pro přehlednost k přeskupování prvků formuláře.



Obrázek 17– typy hran

Po vytvoření hrany je přepínač nastaven na typu přechod (obr. 17, hrana TRANSITION). Přechodová hrana nemá žádné nastavení, neboť neprovádí žádný test. Typ hrany lze přepnout na test vzoru (obr. 17, hrana PATTERN), které pro zpracování vstupu do hrany přímo využívá porovnávání vzorů pattern a that obvyklé při výběru kategorií v AIML. Pro jejich zadání jsou dostupná pole vzor a vzor zmínky. Ta přijímají text ve formátu platném pro složený vzor, zmíněný v úvodu do prostředků jazyka AIML v kapitole 1. Tedy lze vložit normální slova oddělená mezerami, zástupné znaky „*“ a „_“ či značky <bot>.

Dalším dostupným typem hrany je hrana testující výstup generovaný úsekem kódu šablony oproti prostému vzoru (obr. 17, hrana CODE), který může obsahovat pouze mezerami oddělená normální slova a zástupné znaky. Podobné těmto hranám jsou hrany porovnávající vůči vzoru nikoli vygenerovaný text, ale přímo hodnotu uloženou v určeném predikátu (obr. 17, hrana PREDICATE).

Zcela speciální je typ zanoření (obr. 17, hrana SUBNETWORK), který sám nic netestuje, ale předává řízení do zvoleného výchozího uzlu nějaké jiné sítě (kterou může být i sít původní). Teprve pokud zpracování tohoto vstupu dojde v podsíti do koncového uzlu, tento test uspěje.

5.6. Obecné principy rozhraní a syntaktická kontrola

Návrh sítí, jejich uzlů a hran je veden v duchu rozhraní s více dokumenty (MDI). Podporuje zobrazení stejné sítě ve více vnitřních oknech, provedené změny systému v jednom okně jsou aktualizovány ve všech ostatních. Díky tomu lze

pracovat souběžně na několik sítích, či na částech jedné sítě. Úprava hran je řešena obdobně: uživatel může otevřít podrobnosti jedné hrany ve více oknech. Aktualizace všech ostatních oken s podrobnostmi či sítěmi na aktuální verzi proběhne po stisknutí tlačítka Uložit.

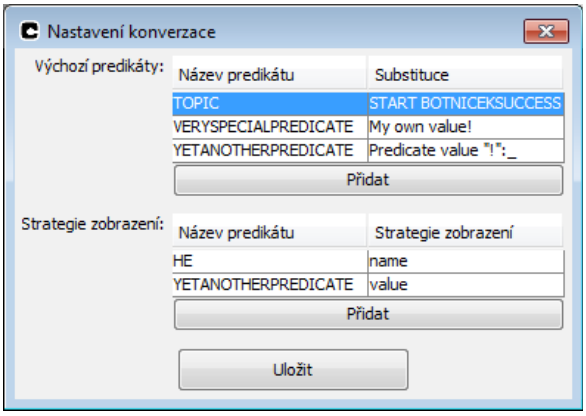
Vstup zadaný uživatelem do polí v podrobnostech hran je ihned analyzován (průběžně již během vkládání znaků) a případné chyby jsou mu ohlášeny pomocí přehledové tabulky na záložce Problémy v dolní části prostředí. Na každém řádku tabulky je zobrazena jedna chyba. Tabulka má čtyři sloupce, buňky v prvním sloupci Popis obsahují podrobnosti problému. Původce chyby (okno s otevřenými podrobnostmi) lze odečíst v druhém sloupci Zdroj problému, tam je uveden název hrany společně s identifikátorem verze okna. Poslední dva sloupce Řádek a Sloupec lokalizují výskyt v daném formulářovém prvku. U jednořádkových prvků je na místě řádku uvedena číslice 1. (Nedostatkem syntaktické kontroly oblastí pro zadání AIML kódu je chybějící lokalizace popisu chyby do češtiny. Tento modul totiž využívá tyto závislosti: výchozí parser XML a ze specifikace odvozený předpis XML schématu AIML dokumentu. Ani jedna z nich bohužel není lokalizována.)

Jestliže se v okně, které chce uživatel uložit, nachází nevyřešené problémy, nedojde k aktualizaci hrany, dokud nejsou odstraněny nebo není okno zavřeno (a tím chybné změny ztraceny) nebo je aktuální verze přepsána z jiné verze okna dané hrany. Přepnutí typu hrany způsobí odstranění hlášení o chybě, která se týkala předchozího vybraného typu, neboť jsou specifická pole vyplněna výchozími validními hodnotami.

5.7. Nastavení konverzace

Ještě před vyzkoušením navrženého robota je nutné nastavit úvodní obsah zásobníku výchozích uzlů. Pokud uživatel umístí na zásobník název nějakého výchozího uzlu, začne zpracování prvního vstupu od uživatele po zahájení konverzace právě v něm. Pokud dojde výpočet v přechodové sítě tohoto stavu na zásobníku do koncového stavu (včetně diskutovaných průchodů podsítěmi přes rekurzivní hrany), dojde k úspěšnému ukončení konverzace s robotem a na další vstupy budou vraceny pouze prázdné řetězce. Nastaví-li uživatel na zásobník

výchozích uzlů více, bude po zpracování vrcholového uzlu pokračovat konverzace obdobně přes další uzly až na dno zásobníku.



Obrázek 18– nastavení konverzace

Nastavení zásobníku se provádí přes menu a položku Běhové prostředí → Konverzace. V okně nastavení konverzace (obrázek 18) se nachází tabulka pro určení výchozích hodnot libovolných predikátů. Zásobník je uložen v predikátu s názvem TOPIC, který v AIML slouží k nastavení aktuálního tématu. Jak bylo osvětleno v kapitole 4., v robotech navržených Botníčkem témata odpovídají stavu zásobníku v čase. V novém projektu je v tabulce již záznam vytvořen a zásobník je v něm nastaven na výchozí uzel START ve vzorové síti HelloWorld.

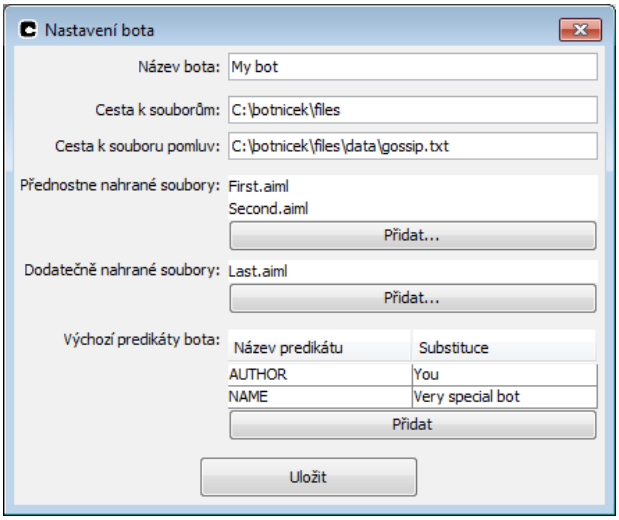
Pokud uživatel chce nastavit na zásobník sled vstupních uzlů, stačí vložit jejich názvy oddělené mezerami s tím, že směrem doprava se nachází dno zásobníku. Jedinou podmínkou, kterou je potřeba dodržet pro korektní fungování robota, je to, že na dně zásobníku bude zarážka v podobě pracovního stavu signalizujícího úspěch.

Po vytvoření nového projektu je již název stavu i se správným prefixem nastaven, stačí tedy při úpravách tento řetězec zcela napravo ponechat. Jinak lze ve stejné tabulce nastavovat výchozí hodnoty libovolných predikátů jako v běžné implementaci interpretu. Tabulka níže pak přiřazuje zobrazovací strategie zmíněně v popisu značky set v kapitole 1. (name pro „nastav a vrať název“ a value pro „nastav a vrať nastavenou hodnotu“).

5.8. Nastavení robota

Další nabízené dialogy v menu Běhové prostředí jsou již v porovnání

s nastavením jiných interpretů AIML zcela běžné. Z Běhové prostředí → Bot je dostupná konfigurace robota pro testování (obrázek 19). Některé její částí mají smysl pouze pro případný export do samostatného či provozního interpretu. Mezi takové patří „Název bota“ a „Cesta k souborům“ a seznamy pro uspořádání pořadí načítání vyexportovaných souborů.



Obrázek 19 – nastavení robota

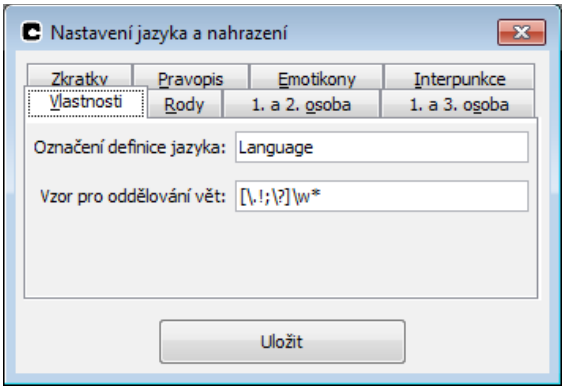
Specifikace totiž přímo neupravuje, jak se má interpret chovat tehdy, kdy soubor zdrojových kódů robota obsahuje duplicitní definici vzorů kategorie, tj. pokud se u dvou kategorií shoduje vzor pattern, that a vzor tématu. Interpret Botníčku proto podobně jako referenční Program D definice šablon duplicitních kategorií přepisuje, a to dříve načtenou šablonu šablonou načtenou později. Pokud uživatel dodává interpretu více souborů s kódem, a ví, že by mělo v některých kategoriích docházet k přepisu, měl by určit pořadí jejich načítání. Může některé soubory přidat buďto jako přednostně či dodatečně načítané vůči těm, u nichž na pořadí nezáleží. Zadává se přitom jen název souboru včetně koncovky, pro rezoluci celé cesty k němu se uvažuje adresář v poli „Cesta k souborům“ (nutně obsahující validní systémovou cesty). K dispozici jsou k tomu seznamy „Přednostně nahrané soubory“ a „Dodatečně nahrané soubory“. Pořadí položek (a tedy i načítání) lze měnit přetahováním myší a vybranou položku smazat klávesou Delete.

I při testování mají smysl výchozí predikáty robota, na které lze odkazovat pomocí značek bot (ve formátu <bot name=“PREDIKÁT“/> ve vzorech podrobností

hrany při přepnutí na typ, jež vzory testuje.

5.9. Nastavení jazyka

Nastavení jazyka (obrázek 20) sdružuje nahrazení pro první fázi předzpracování vstupu (Zkratky, Pravopis, Emotikony, zachování Interpunkce), předpisy pro substituce, které mají provádět značky person (1. a 2. osoba), person2 (1. a 3. osoba) a gender (Rody) a specifikuje oddělovače vět (karta Vlastnosti). Až na Vlastnosti se na každé kartě dialogu se nachází tabulka, do které lze vložit páry regulární výraz – řetězec nahrazení. Regulární výraz musí vyhovovat formátu, se kterým pracuje třída Pattern standardní knihovny Javy[19]. Substituční řetězec je omezen na nahrazení pouze celého textu zachyceného výrazem. Vzor pro oddělování vět ve Vlastnostech je opět určen pomocí regulárního výrazu.



Obrázek 20– nastavení jazyka

5.10. Zkušební konverzace, export a uložení projektu

V průběhu návrhu je kdykoli možné robota vyzkoušet. Stačí zahájit testovací konverzaci pomocí Běhové prostředí → Spustit test. Spodní panel se přepne na kartu Test (odtud lze spustit i první test pomocí tlačítka), kde se nachází oblast s historií konverzace a pod ní vstupní pole a tlačítko Zadat (obrázek 13 dole). Vstup lze ovšem potvrzovat nejenom tlačítkem, ale přirozeně i klávesou Enter. Pro sestavení robota jsou užity vždy pouze uložené, a tím pádem i validní změny. Tedy se není nutné obávat chyb zobrazených v tabulce Problémy, ty se týkají pouze neuložených oken. Testovací panel na druhou stranu nenačte změny, které byly na systému sítí provedeny po zahájení testu, pro to je nutné spustit nový test.

Pokud robot funguje dle uživatelových představ, je možné přistoupit k exportu zdrojových souborů. Ten je dostupný přes Projekt → Exportovat, kde po otevření stačí vybrat adresář, kam se mají soubory umístit. Exportuje se vždy jeden soubor pro každou síť, jež nese název sítě doplněný o koncovku aiml. Dále se do stejného adresáře umístí .properties soubory s konfigurací. Ty lze po konverzi použít v jiných interpretech, vhodných pro produkční nasazení.

Projekt lze pro pozdější práci uložit běžným způsobem přes menu Projekt. K dispozici jsou běžné dialogy „Uložit“ a „Uložit jako“. Projektové soubory se ve výchozím nastavení ukládají v proprietárním formátu s koncovkou btk. Lze je otevřít pro další práci opět nepřekvapivě přes Projekt → Otevřít. Před zavřením projektu, vytvořením jiného či ukončením aplikace je uživatel vždy dotázán, zda-li nechce dosavadní postup uložit. Pro všechny operace zmíněné v této podkapitole je navíc dostupná zkratka v podobě hlavní lišty, kde je každá reprezentovaná vlastním tlačítkem.

5.11. Demonstrační projekt

Přiložený projektový soubor canteen.btk obsahuje kostru robota provázejícího objednávkou jídla ve vyvařovně. Průběh rozhovoru s ním se sestává ze tří částí, které mají každá svou síť. Po spuštění testu byla na vrchol zásobníku nastavena síť Intro (Běhové prostředí → Konverzace a změna obsahu predikátu TOPIC), která zvládá pozdravit a dotázat se na to, zda-li si zákazník přeje jídlo vzít s sebou. Tuto informaci si pamatuje v predikátech a později užije. Pod ní je na zásobníku připravena síť Loop, jež kromě náhodně dávkované nabídky speciality šéfkuchaře obsahuje smyčku, ve které se robot dotazuje na položky objednávky. Na dně je umístěna síť Outro. Ta dokáže nahlásit celkovou útratu, neboť robot si pro každou položku její cenu sčítal, a podle z Intra uloženého místa konzumace mění hlášku, kterou robot vyprovodí uživatele od pokladny.

Zdárnému průběhu diskuze, jež je v jedné variantě vypsána v obrázku 21, bylo kvůli absenci většího množství záchytných vzorů napomoženo vhodným tvarem odpovědí, neboť cílem je na malém prostoru ukázat smysluplné užití co nejvíce prvků z grafického návrhu dialogu, nikoli zahltit množstvím uzlů a hran.

```
Uživatel: Good evening!
Canteen: Welcome at Chalmun's - the best canteen in town! Before we
begin, tell me please, will you dine here or is it a takeaway?
Uživatel: Takeaway, please.
Canteen: Would you like to try out our specialty?
Uživatel: Yes.
Canteen: The chef will be pleased. And anything on the menu?
Uživatel: I'll try salad.
Canteen: Fresh salad just for you! Anything else?
Uživatel: No.
Canteen: That's 5 bucks. Do you want to pay with credit card or in cash?
Uživatel: Credit card then.
Canteen: Thank you. ... Oh! I can see that your order's been already
completed. Here's your package! See you soon!
Uživatel: Thanks!
```

Obrázek 21 – příklad rozhovoru při testování demonstračního projektu

5.12. Alternativní aplikace

Jelikož je pokus o podporu řízení dialogu v čistém AIML původním přínosem této práce, v případě uživatelova zájmu o alternativní aplikace jej lze odkázat pouze na programy, jež pracují přímo s elementárními prvky jazyka. Vynikající a stále aktivně vyvíjenou aplikací pro vývoj v AIML je německý GaitoBot AIML Editor, program pro Windows běžící na platformě .NET.[20] Kromě toho, že nabízí všechny potřebné funkce pro vývoj ve formě přístupné i začátečníkům, jeho nespornou výhodou je především přímé napojení na komerční službu, která slouží pro provoz z Internetu přístupných robotů. K oblasti řízení dialogu má u něj neblíže užitečná funkce pro znázornění možných posloupnosti kategorií vzhledem k výskytům rekurzivní značky srai.

Kromě toho lze doporučit strohý, ale funkčně velmi dobře vybavený editor AIMLpad. Obsahuje kromě interpretu například i funkční rozhraní pro IRC, interpret JavaScriptu a vlastní skriptovací nástroj.[21] Kromě těchto význačných projektů je pro AIML dostupné velké množství dalších nástrojů. Mnozí autoři robotů navíc upřednostňují pro editaci, vzhledem k poměrně ploché struktuře dokumentů, tabulkový procesor, ve kterém mají vyneseny na řádcích kategorie a ve sloupcích příslušná témata, vzory a šablony.[22]

6. Programátorská dokumentace vývojového prostředí

Poznatky z předchozích kapitol byly zohledněny při návrhu a programování vlastního vývojového prostředí pro roboty v jazyce AIML. Je na místě připomenout cíle, které byly sledovány:

- Setrvání na platformě AIML,
- s tím související snaha eliminovat či výrazně omezit pokusy o změnu specifikace interpretu..
- Implementaci výpočetního procesu rozšířených přechodových sítí jakožto vhodného prostředku pro rozšíření možností jazyka
- a zároveň komfortu vývoje.

6.1. Vývojová platforma

Pro vývoj byla zvolena softwarová platforma Java. Kromě blízkosti autora práce k souvisejícím technologiím má toto řešení významné výhody, které jsou v duchu stanovených cílů:

- Poměrně snadné nasazení na různých operačních systémech.[23]
- Referenční (byť nepříliš striktní) interpret jazyka AIML, Program D, je naprogramován též v Javě a je šířen jako otevřený software.[24] Mohl tak tedy posloužit jako pomůcka při osvětlování nejasností ve specifikaci jazyka.
- Vyzrálá a multiplatformní knihovna uživatelských prvků Swing. Ta je sice postupně nahrazován modernější knihovnou JavaFX[25], stále ale zůstává doporučovaným řešením s širokou komunitní podporou.[26]
- Dostatečný výkon, který dovoluje rychle provádět transformace stromu návrhu do podoby zdrojového kódu pro jazyk AIML.
- Obecné výhody jazyka Java, zejména relativně dobrá a široká podpora technologií pro zpracování jazyka XML (jehož je AIML dialektem) již ve výchozí instalaci.[27]

6.2. Interpret

Vývoj prostředí Botníček probíhal ve dvou etapách: v první, v rámci

ročníkového projektu, byla naprogramována vlastní implementace interpretu jazyka. Druhou etapou byla samotná bakalářské práce – vývoj IDE. To používá interpret interně k testování vygenerovaného kódu a konverzaci v testovacím okně.

Vytvoření vlastní implementace interpretu mělo vícero kladných efektů:

- Striktnější řešení oproti nejvýznamnějším realizacím, jež jsou obvykle určené k tomu, aby stabilně běžely ve čtyřiadvacetihodinovém provozu na webových serverech.[24] Ne zcela formální a místy nejasná specifikace AIML má pak jako důsledek to, že nejpopulárnější stroje jsou velmi tolerantní i vůči relativně jasným prohřeškům vůči specifikaci (např. oproti specifikaci chybějící definice jmenných prostorů, ignorování velikosti písmen v hodnotách atributů[4, 28]), což snižuje přenositelnost zdrojových kódů, které se na benevolenci konkrétního interpretu do velké míry spoléhají.
- Přínosem byl též lepší náhled na mezní případy a víceznačná místa ve specifikaci a pochopení filozofie jazyka.
- Do třetice pak snazší napojení na IDE a možnost sdílení některých částí základních částí kódu a knihovní výbavy, i přes ve zpětném pohledu poměrně naivně a komplikovaně navržené rozhraní.

Navzdory snaze držet se specifikace bylo nutné přijmout některé kompromisy a doplnit mezery specifikace, které v důsledku ovlivnily i vývoj IDE. Především jazyk AIML nechává zcela na vůli interpretu, v jaké formě a do jaké míry bude provádět normalizaci uživatelského vstupu. Příčinou jsou omezené prostředky jazyka pro práci se souvislým textem, kdy specifikace nedává programům napsaným v AIML prostředky pro manipulaci se vstupem po znacích[4]. Přestože mohou mít dva interprety shodný způsob vyhodnocování samotného zdrojového kódu, nemusí podávat z tohoto důvodu shodné výsledky. Užitý interpret vychází při normalizaci vstupu z doporučení ve specifikaci a návrhu referenčního Programu D.

Dále je na autorovi interpretu, v jaké formě bude uložena a načítána konfigurace jak stroje, tak robota. Tedy i při shodě procesů normalizace je nutné pro zajištění stejných výsledků vyřešit konverzi formátů, pokud to je vůbec možné. Tento problém byl v Botníčku vyřešen tak, že interpret sice využívá obdobné dělení konfigurace jako Program D, nicméně pro její načtení očekává ještě jednodušší

konfigurační formát, který je ovšem na obvyklé definice ve tvaru „regulární výraz – substituce“ naprosto postačující a při užití vhodného editoru i přehlednější. Interpret Botníčku ukládá veškerou konfiguraci v .properties souborech[29], jež jsou častým prostředkem pro uložení konfigurace při programování v Javě.

6.3. Další závislosti

Jedinými celistvým převzatými úseky aplikace mimo standardní knihovnu Javy jsou zvýrazňovač syntaxe XML XMLEditorKit z knihovny Bounce[30] a rozvržení formulářových prvků s názvem WrapLayout od Roba Camicka, které opravuje chování výchozího FlowLayout ve Swingu[31]. Zbývající externí závislostí, užitou napříč kódem funkční části aplikace (její testy jsou osvětleny dále), je knihovna Google Guava[32]. Ta poskytuje programátorovi prověřené základní stavební bloky, které by jinak musel pro každý projekt, a tedy i Botníček, napsat sám (např. velmi užitečné neměnné – immutable kolekce, řetězcové operace, třídění) a také napravuje některé deficience Javy v užité verzi (např. volitelný typ místo užívání prázdné reference). Je hojně nasazena v zapouzdřeném implementujícím kódu, ale veškerá veřejná i interní rozhraní jsou navržena tak, aby se vyhýbala přiznanému užití jejích rozšíření.

6.4. Sestavení

Pro sestavení je užit nástroj Apache Ant[33]. Jeho konfigurační soubor build.xml se nachází v kořenové adresáři části kódu pro vývojové prostředí: botnicek/Botnicek IDE. Kromě samovysvětlujících hlavních cílů sestavení jako build, clean, run a další obsahuje i mnoho dalších, které mají za úkol spouštět sady testů či demonstrovat navržené modifikace prvků grafického rozhraní.

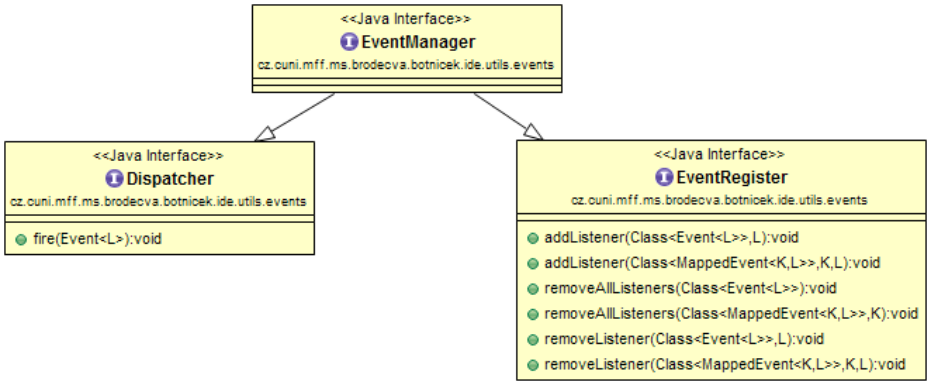
6.5. Struktura programu

Zdrojové kódy tříd vývojového prostředí Botníček jsou rozděleny do balíků v kořeni cz.cuni.mff.ms.brodecva.botnicek.ide, které zhruba odpovídají fázím práce s prostředím. Mnohé balíky pak, v případě, že je v nich realizována MVC architektura (ve variantě založené na doporučeních v [34]), dodržují vnitřní

uspořádání na pod-balíky: model, view, controllers a events (neboli balíky s modelem, pohledy, řadiči a předpisy událostí, o kterých se zpravují).

6.6. Zasílání zpráv o událostech a provedení MVC

Při návrhu částí aplikace, které jsou v přímé interakci s uživatelem prostředním grafického rozhraní, bylo užito architektury MVC, která je jedním ze ustálených řešení tohoto problému. Implementovaná varianta se vyznačuje posílením role řadiče, přes který v ní probíhá veškerá komunikace mezi modelem a pohledy. Cílem je dosáhnout většího oddělení modelu od pohledů, včetně přesunutí nezbytné logiky zasílání zpráv o změnách v co největší míře do řadiče místo něj.



Obrázek 22 – UML diagram rozhraní správce událostí

Pro samotné zasílání zpráv byl navržen vlastní správce událostí (balíček cz.cuni.mff.ms.brodecva.botnicek.ide.utils.events, obr. 22), který má oproti oproti běžným alternativám (např. jako se nachází v [34]) několik výhod:

- Díky užití generických typů v popisu událostí a jejich posluchačů je typově bezpečnější (řešení inspirováno [35]).
- Lze užít i mimo kontext grafických rozhraní.
- Nevyžaduje odhlašování posluchačů událostí tehdy, když už nejsou potřeba, protože pro jejich uložení užívá mapu se slabými referencemi[36]. Tím zároveň zjednodušuje kód svého klienta a zabraňuje únikům paměti.
- Posluchače se mohou přihlásit k odběru události metodou se dvěma přetíženími, jedno vyžaduje klíč a druhé je bez klíče (obrázek 22 vpravo). Pokud je pak událost podporující klíče vytvořena a publikována, jsou zpraveny posluchače, které byly zaregistrovány se shodným klíči či jej

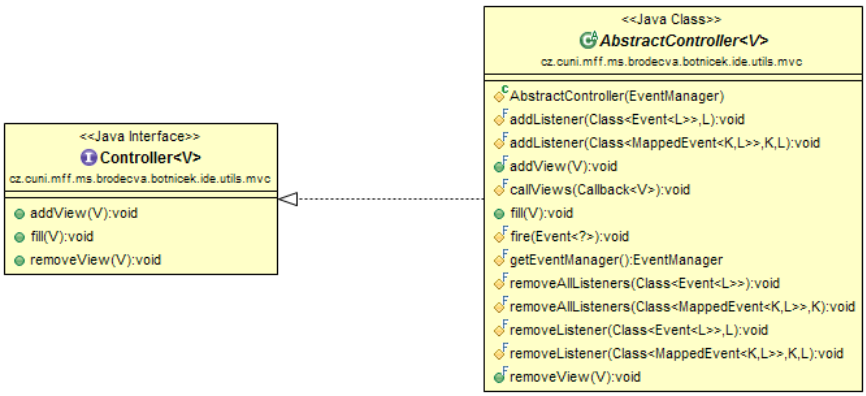
ignorovali. Toto řešení inspirované mechanismem signálů, slotů a mapovače signálů v knihovně Qt[37], umožňuje zúžit okruh adresátů, aniž by si každý z nich ověřoval, jestli se přijatá událost týká právě jeho.

Životní cyklus užitého MVC s tímto správcem událostí lze popsat následovně: *Pohled* v tovární metodě přijímá jako parametr referenci na řadič. Metody řadiče jsou pak volány posluchači ovládacích prvků grafického prostředí za účelem modifikace modelu. Pohled samotný se pomocí metody addView registruje na řadiči mezi ostatní již registrované pohledy, aby byl zpraven o budoucích aktualizacích modelu.

Řadič přijímá v tovární metodě referenci na model, kterou si uloží do soukromého pole, aby na ni podle potřeby mohl volat metody upravující model. Dále je řadiči při konstrukci dodán správce událostí, na který zaregistruje posluchače událostí modelu. Posluchače událostí jsou implementovány jako vnitřní třídy řadiče, aby měly přístup k pohledům registrovaným na něm, které budou po obdržení zprávy aktualizovat.

Stejná instance správce událostí je dodána jako součást i třídám *modelu*, jež budou ohlašovat vznik událostí. Třídám modelu je vhodné předávat jenom část rozhraní správce událostí, pod názvem Dispatcher, ve stejném balíku jako správce, byť je samozřejmě teoreticky možné je na původní typ přetypovat a registrovat události pokoutně také.

Jestliže pohled zavolá modifikující metodu řadiče, řadič dle svého usouzení zavolá příslušnou metodu modelu. Model po provedení úprav propaguje změny ohlášením vytvořeného objektu události pomocí metody fire na Dispatcheru. Na tuto událost zareaguje posluchač, vnitřní třída řadiče, který aktualizuje pohledy přímým voláním metod.



Obrázek 23 – Rozhraní řadiče a jeho podpůrná implementace

Ve skutečnosti je celý proces o úroveň jednodušší, neboť všechny plnohodnotné řadiče v projektu dědí od třídy AbstractController z balíku cz.cuni.mff.ms.brodecva.botnicek.ide.utils.mvc (obrázek 23 napravo), jež poskytuje svým potomkům širokou paletu ulehčujících metod. Například obstarává přidávání a odebírání pohledů a dovoluje na nich všech volat pomocí zpětného volání (callbacku) libovolné metody.

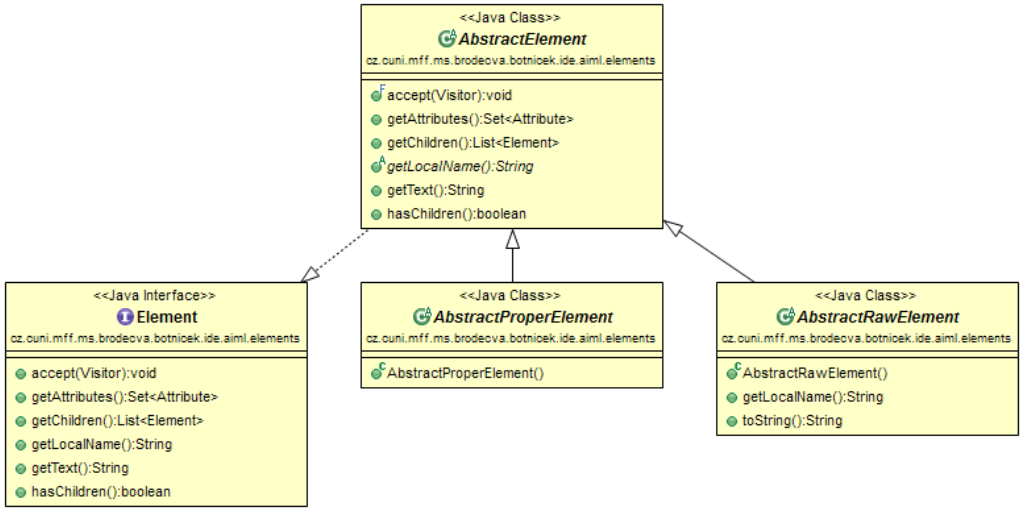
6.7. Odlehčený objektový model dokumentu

Hlavním úkolem Botníčku je poskytnout prostředky pro návrh robota v prostředí inspirovaném DAF a RPS, a tento návrh pak efektivně převést do podoby zdrojového kódu jazyka AIML. Při analýze problému vyvstala nutnost modelovat prvky dokumentu jazyka AIML v podobě, která bude snadno použitelná při popisu transformací, jež se provedou na výchozím modelu systému sítí. Zároveň by mělo být snadné z jejich stromu vygenerovat výstupní dokument.

Pro tyto potřeby je v balíčku aiml vytvořena odlehčená objektová reprezentace všech prvků jazyka AIML, přesněji v podbalíku elements. Je inspirována standardní konvencí DOM konsorcia W3C pro dokumenty XML (a XHTML či HTML)[38]. Její zaměření je však mnohem užší, a proto je rozhraní objektů jednodušší. Předně strom a jeho prvky není nutné po vytvoření modifikovat a není nutné podporovat jakoukoli funkcionalitu ani druh obsahu, který nebude využit při generování. Proto místní rozhraní Element (obrázek 24 vlevo) definuje pouze nejnutnější metody pro výpis atributů, přímých potomků, názvu prvku a výpis textového obsahu. Navíc je ještě poskytnuta podpora pro průchod stromem prvků

prostřednictvím návrhového vzoru návštěvník.[39]

Z hlediska implementačního je důležitá třída AbstractRawElement, která na rozdíl od AbstractProperElement (oboje též v obrázku 24), od níž dědí všechny prvky předdefinované podle specifikace AIML, je určena pro vložení takového obsahu do dokumentu, co pochází z některého ze vstupních polí vlastnosti hrany a již prošel syntaktickou kontrolou. Jinak je zbytek balíku aiml naprogramován paralelně vůči specifikaci, včetně definic datových typů v podbalíku types, které je třeba v AIML rozlišovat (vzory, normální slova, kód šablony).



Obrázek 24 – vrcholové třídy odlehčeného objektového modelu AIML

6.8. Systém sítí

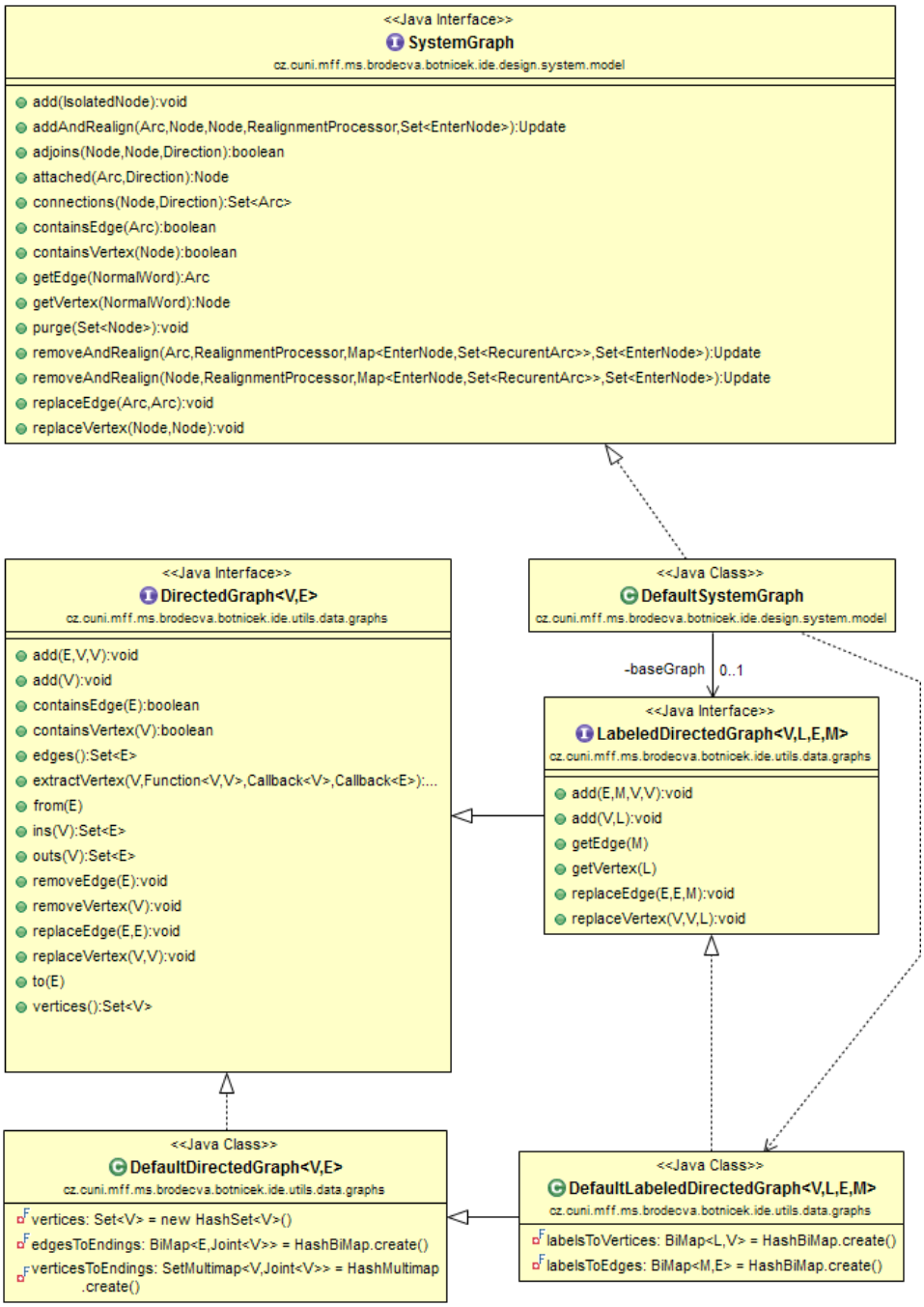
Implementace návrhu dialogu od uzlů až po celý systém sítí je umístěna v balíčku design. Přestože se systém z vnějšku jeví jako množina relativně nezávislých sítí, tak i těch několik málo způsobů, kterými se jednotlivé sítě ovlivňují, se ukázalo natolik komplexních, že bylo rozhodnuto sloučit veškeré manipulace se systémem pod jedno rozhraní System (balík system), a jeho části v náležitých kontextech exponovat přes dílčí rozhraní pro sítě (rozhraní Network v balíku networks), uzly (Node, nodes) a hrany mezi nimi (Arc, arcs). Pokud tedy např. je požadavkem přidat uzel do sítě, lze to provést přes instanci naplňující rozhraní Network, ale výchozí implementace přitom bude volat metodu rodičovské instance třídy System, s tím, že síť předá jako další parametr.

Jednotné rozhraní ovšem neznamená monolitickou implementaci. Výchozí

implementace DefaultSystem deleguje podstatnou část svých povinností na své komponenty. Systém musí především udržovat unikátnost pojmenování stavů (tj. názvů uzlů a hran), případně provozních predikátů. K tomu mu pomáhají pojmenovávací autority (NamingAuthority v balíku systém), které přidělují, generují a vedou záznamy o názvech stavů a predikátů. Jejich unikátnost je zárukou korektnosti průchodu sítěmi. Užitá implementace přímo podporuje formát normálních slov tak, jak je běžný v AIML.

Neméně podstatnou povinností systému je mít přehled o tom, jaké se v něm nacházejí uzly a jakými hranami jsou propojeny. Výběr vhodné reprezentace těchto relací prošel drobnými změnami s tím, jak na něj klesaly požadavky. Např. původní návrh počítal s tím, že bude možné přímo propojovat dva uzly více hranami stejné orientace. Toto řešení bylo po úvaze opuštěno s tím, že ušetření pomocného uzlu oproti výslednému řešení v podobě orientovaného grafu bez rovnoběžných hran a smyček, by bylo vykoupeno neúměrnou komplikací znázornění hran a sníženou přehledností.

Jak je ilustrováno na diagramu v obrázku 25, hrany a uzly jsou vkládány do vlastní implementace grafu SystemGraph, která obaluje konkretizaci generické třídy DefaultLabeledGraph, jež dovoluje vkládat orientované hrany a uzly sítí a nezávisle je pojmenovávat normálními slovy. Samotný graf je pak implementován přímočaře pomocí množiny vrcholů, bijekce množiny hran na množinu spojnic vrcholů a zobrazení množiny vrcholů na množiny spojnic, v nichž se vrchol nachází na jednom z krajů. To, do jakých sítí uzly a hrany patří, je uloženo odděleně přímo v DefaultSystem, neboť tato informace má zásadní dopad na rozsah modifikací, které je nutno provést při elementárních operacích se systémem.



Obrázek 25 – hierarchie implementace systémového grafu

6.9. Úpravy systému

Nejjednodušší operací z hlediska implementace je přidání nového uzlu, který se pro zjednodušení vkládá vždy jako izolovaný uzel. Kromě přiřazení do sítě dochází přitom jen k nutnému vygenerování unikátního jména pomocí autority.

Uzly a hrany jsou implementovány jako jednoduché, téměř neměnné

objekty (téměř, neboť obsahují neměnnou referenci na rodičovskou síť, která samotná ale proměnlivá je). Toto rozhodnutí se ukázalo jako dvojsečné, neboť na jednu stranu výrazně zjednodušilo návrh (dovoluje s těmito objekty snadněji nakládat v kolekcích, předávat ve zprávách,...), na druhou stranu zkomplikovalo operace, které mění vlastnosti objektů či modifikují graf.

Při každé z nich je nutné starou verzi uzlu či hrany nahradit novou všude, kde je potřeba. Například každé dva uzly v síti jdou propojit hranou, pokud už mezi nimi v daném směru žádná není či neexistuje jiná zásadní překážka. Speciálně u dvou do té doby izolovaných uzlů tato operace vyvrcholí v jejich nahrazení v grafu uzly novými, jeden bude výchozí pro danou síť, druhý pak koncový.

S tím, že přidání hrany může změnit druh uzlu z hlediska jeho logické polohy, systémový graf počítá. Po každém přidání hrany dochází k přepočítání hran vedoucích z a do krajních uzlů. Pak je uzel nahrazen obdobným, až na typ. Ten se podle konfigurace hran může změnit z izolovaného na výchozí či koncový a z výchozího či koncového na vnitřní.

Přidávání hran ovšem nemusí proběhnout zdaleka hladce. V uživatelské dokumentaci byl představen typ hrany, která odkazuje na nějaký vstupní uzel své vlastní či cizí sítě. Pokud se uživatel pokusí přidat příchozí hranu k vstupnímu uzlu sítě, na který je takto odkazováno, je žádoucí, aby mu to nebylo umožněno. Proto si systém sítí vede v příslušně pojmenovaných proměnných záznamy o dostupných výchozích uzlech, odkazovaných uzlech a odkazujících hranách, aby tyto situace mohl detekovat.

Podobnému okruhu problémů systém čelí, pokud je naopak nějaká hrana odstraňována. Při tom může dojít k degradaci vnitřního uzlu na koncový či krajní a koncového či výchozího uzle na izolovaný. Pokud je na vnitřní uzel odkazováno a nejde o výjimečný případ, kdy na něj odkazuje jenom právě odstraňovaná hrana, opět musí systém operaci zamezit a dát uživateli vědět, aby závislosti před dalším postupem vyřešil.

Největší dopad má odstranění uzlu, neboť se dotýká všech přilehlých hran a uzlů na jejich opačných koncích. Toto nejbližší okolí je nutné otestovat a eventuálně opravit jako u odstraňování hran. Přitom je třeba zajistit, aby při pochybení a

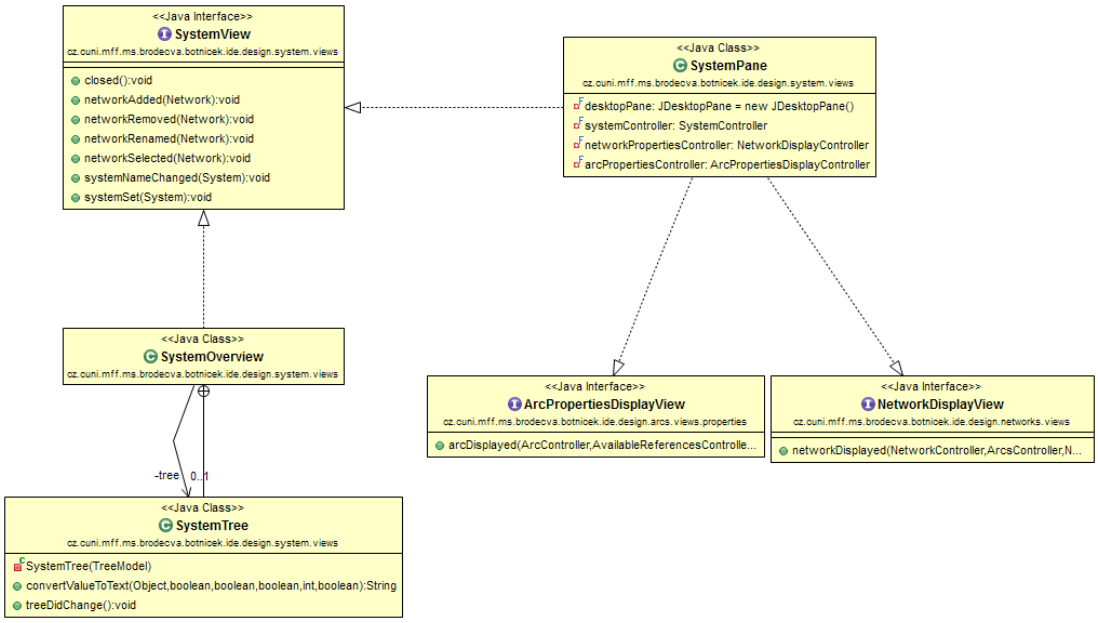
následném vyhození výjimky nebyl systém zanechán v nekonzistentním stavu. Proto graf systému po úspěšné aktualizaci vrací provedené změny v podobě objektu Update, který popisuje provedené aktualizace, jež jsou následně aplikovány na pomocné datové struktury systému, které monitorují reference a složení sítí.

Ve většímu měřítku probíhají obdobné procesy při odstranění celé sítě, kdy je nutno odebrat všechny její uzly a hrany. Při tom je naštěstí nutné vyřešit pouze problém odkazů vedoucích z jiných sítí (a podle potřeby odstraňování opět zastavit a informovat o příčině uživatele). Pak už jen stačí smazat všechny uzly (navázané hrany se odstraní v grafu systému automaticky), aniž by bylo třeba nějak přepočítávat změny jako při inkrementálním postupu.

Uzly se přirozeně nahrazují i tehdy, když není třeba měnit jejich typ podle umístění v grafu, ale například se přepíná mezi náhodným a uspořádaným výběrem hran pro přechod. Tehdy je však v systému nahrazení lokalizované skutečně jen na reference pro daný uzel. Stejně tak, pokud se mění typ hrany. Systém si akorát navíc u hran, které odkazují na vstupní uzly, zapíše existenci odkazu pro výše uvedenou kontrolu.

6.10. Zobrazení částí systému

Pro zobrazení částí systému (obrázek 26) jsou určeny pohledy SystemView, které jsou jako zbytek celého grafického prostředí implementovány pomocí toolkitu Swing[40]. V SystemOverview se lehce upravuje výchozí implementace zobrazení stromu JTree. Uzpůsobuje se pro znázornění obsahu systému, který na rozdíl od obecného zakořeněného stromu má vždy jen jedno další patro, a u nějž tedy není důvod pro to, aby nebyly všechny sítě za všech okolností zobrazeny, pokud se vejdou do okna. SystemPane pak skrývá běžnou plochu JDesktopPane pro vnitřní rámy se sítěmi a podrobnostmi hran.



Obrázek 26 – zobrazení částí systému

6.11. Sítě

Model a řadič sítě v networks svou práci delegují na kód v balíčku systém. Těžištěm této části projektu je pohled na plochu pro návrh sítě, který je implementován třídou NetworkInternalWindow, obsahující vnitřní rám JInternalFrame, jež je vkládán jako synovská komponenta do panelu JDesktopPane skrytém v system.SystemPane. Do vnitřního rámu sítě lze umisťovat uzly a ty propojovat hranami. Tyto prvky jsou samotné implementovány jako komponenty, u nichž je předefinována metoda pro vykreslení a obsluhu událostí tak, aby měly podobu a chování barevných kruhů a spojnic mezi nimi.

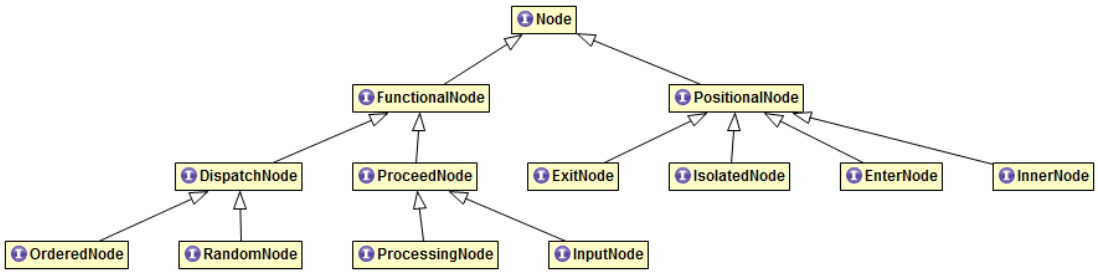
Obavy o plýtvání zdroji kvůli tomu, že každý jednotlivý prvek je kompletní zobrazovanou komponentou prostředí Swing JComponent, se nakonec experimentem ukázaly jako liché. Snadnost zvoleného řešení oproti případnému pokusu o emulaci chování skrze komplexní předefinování metod pro vykreslení a obsluhu událostí nadřazeného rámu bohatě vynahrazuje potenciální výkonnostní zisky, které by se projevily až při výskytu extrémního množství uzlů a hran v síti.

Přesto bylo nutné rám sítě přeci jen mírně přizpůsobit, aby nedocházelo při vytváření nových oken na vnitřní ploše k jejich překrývání. Také byl na jeho panel s obsahem navěšen posluchač ArcDesignListener pro návrh hran mezi uzly, který po

kliknutí vykresluje spojnice mezi adepty na kraje hrany. Tyto spojnice jsou opět odvozené od JComponent s tím, že jejich parametry jsou dynamicky měněny podle polohy myši. Počáteční uzel a kurzor myši vymezuje obdélník komponenty, z níž je vykreslena pouze vhodná diagonála.

6.12. Uzly

Způsob reprezentace uzlů, přesněji jejich možných druhů, je výsledkem kompromisu. Současná podoba implementuje hierarchii rozhraní (obrázek 27) dvanácti třídami, jež až na nekompatibilní kombinace odpovídají kartézskému součinem hodnot, kterých mohou tři rozlišované kvality uzlu nabývat. Uzel může být dle umístění v síti výchozí, krajní či koncový. Podle toho, jestli se v něm výpočet zastaví, aby počkal na další vstup uživatele se rozlišují uzly *procesní* (nezastaví) a *vstupní* (zastaví). Konečně dle způsobu výběru další hrany při backtrackingu simulujícím nedeterminismus RPS jsou uzly *náhodné* či *uspořádané*. Výsledkem jsou pak implementující třídy, nacházející se v nodes.model.implementations pod názvy jako EnterRandomProcessingNode či ExitInputNode.



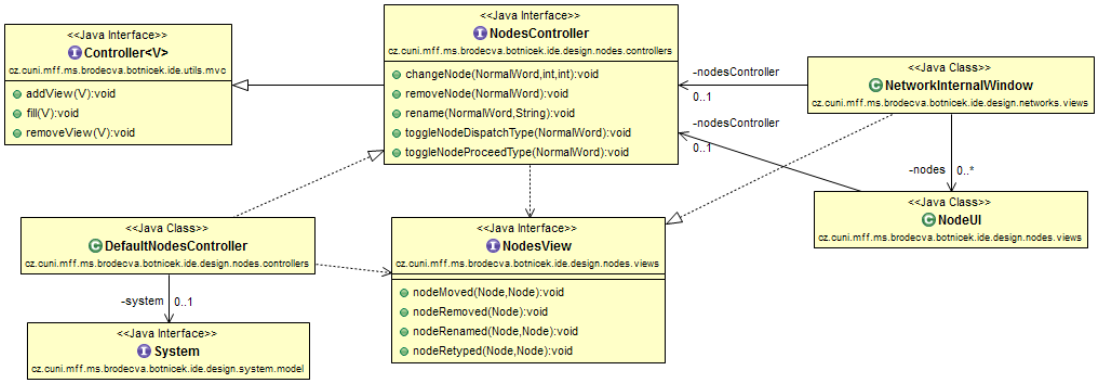
Obrázek 27 – hierarchie rozhraní uzlů

Tato reprezentace na první pohled není příliš pružná a rozšiřitelná, neboť například při potenciální extenzi možných kvalit uzlu o jedinou další by se počet implementaci až zdvojnásobil. Přesto nakonec zvítězila, neboť má významné klady oproti dynamickým řešením (popisnost, přirozené užití v algoritmech), a negativum v podobě obtížné rozšiřitelnosti je vyváženo tím, že implementace samotné neobsahují téměř žádný kód a že je nastavení velmi nepravděpodobné.

Hlavním úkolem odlišného zpracování různých typů uzlů je implementace algoritmu pro převod sítě na odlehčený objektový model dokumentu jazyka AIML. Jeho podstata spočívá v průchodu grafem sítě, při čemž na každém uzlu (a hraně, viz

dále) se podle jeho kvalit rozhoduje, jakým způsobem bude přepsán do podoby tématu v AIML dokumentu. Pro tento typ rozhodování je nesmírně užitečné porovnávání vzorů tak, jak je známé v některých jiných, často funkcionálních jazycích[41], či alespoň dynamický vícenásobný výběr, tedy výběr přetížení metody podle běhového typu parametru. Bohužel ani tento koncept standardní Java nepodporuje.[42]

Vícenásobný dynamický výběr lze simulovat pomocí návrhového vzoru návštěvník, který se aplikuje mnoha způsoby. Pro implementaci algoritmu byla zvolena varianta klasické implementace převzatá z [43], která navrhuje řešení návštěvníka (v instancích, kdy je užíván jako náhrada za vícenásobný výběr) pomocí neměnné třídy (vzor návštěvník je jinde v projektu užíván v proměnné variantě, kdy je jeho schopnost akumulovat vstup naopak žádoucí). Návštěvník má značnou nevýhodu v tom, že vytváří dodatečné nároky na třídy, které jím mají být zpracovatelné, v podobě akceptačních metod. Tedy představuje další zásah do šance na rozšíření, který lze ovšem přehlédnout se stejnými argumenty. Odměnou je průzračný zápis algoritmů.



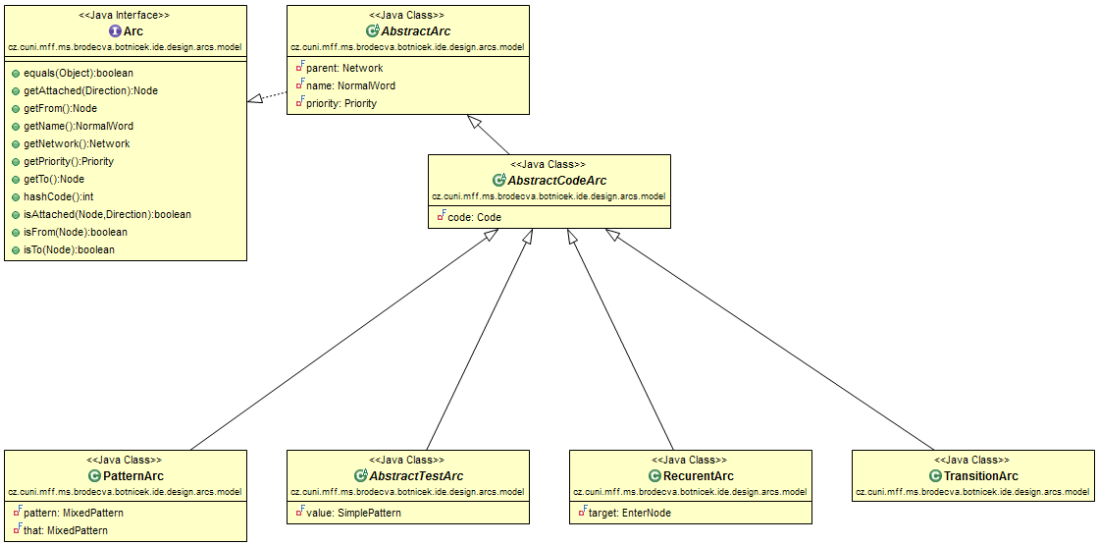
Obrázek 28 – MVC pro uzly

Pro převod uzlů na kompatibilní typy slouží instance NodeModifier, výchozí implementace DefaultNodeModifier obsahuje relativně rozsáhlou tabulku pravidel, která zachycuje, jaký typ uzlu lze převést na který. Změnu implementace uzlu podle polohy, která není libovolná, ale závisí vždy na počtu příchozích a odchozích hran, obvykle zajišťuje DefaultRealignmentProcessor, jež naplňuje rozhraní RealignmentProcessor.

V zobrazení sítě jsou uzly reprezentovány pomocí komponenty NodeUI v podbalíku views. Každá obsahuje referenci na řadič NodesController, pomocí nějž ovlivňuje svůj model. Komponenty samotné už nejsou při změně uzlu aktualizovány přímo, jako tomu je u nadřazených částí systému sítí, ale voláním metod na NodeUI prostřednictvím zobrazení sítě, ve které se komponenta nachází. Důvod, proč sítě implementují pohled na všechny uzly sítě NodesView, místo aby byly uzly aktualizovány přímo, je ryze praktický: při větším počtu zobrazení jedné sítě s mnoha uzly by neúnosně narostl počet registrovaných posluchačů. Alternativním řešením by bylo sdílet posluchače modelu uzlu napříč pohledy na daný uzel, to by však odhalilo vnitřní mechanismus provedení MVC architektury. Celá situace je znázorněna na obrázku 28.

6.13. Hrany

Nastavení hran je mnohem komplikovanější, proto je jim vyčleněno zvláštní okno spravované třídou views.properties.ArcInternalWindow, které se otevírá po poklepání na zobrazení hrany v grafu. U hran na rozdíl od uzlů se sleduje pouze jedna kvalita, a tou je jejich druh. Jednotlivé druhy se od sebe ovšem značně liší, co se týče možností nastavení, proto je obsah okna s nastavením hrany přepínatelný pomocí komponent JRadioButton. Pokud uživatel přepne druh hrany, vyplní správně požadovaná pole a uloží změny, projeví se to na modelu tak, že daný uzel bude v grafu nahrazen novou instancí, která bude inicializována podle údajů v aktuálním okně hrany. Třída instance je určena vybraným druhem.



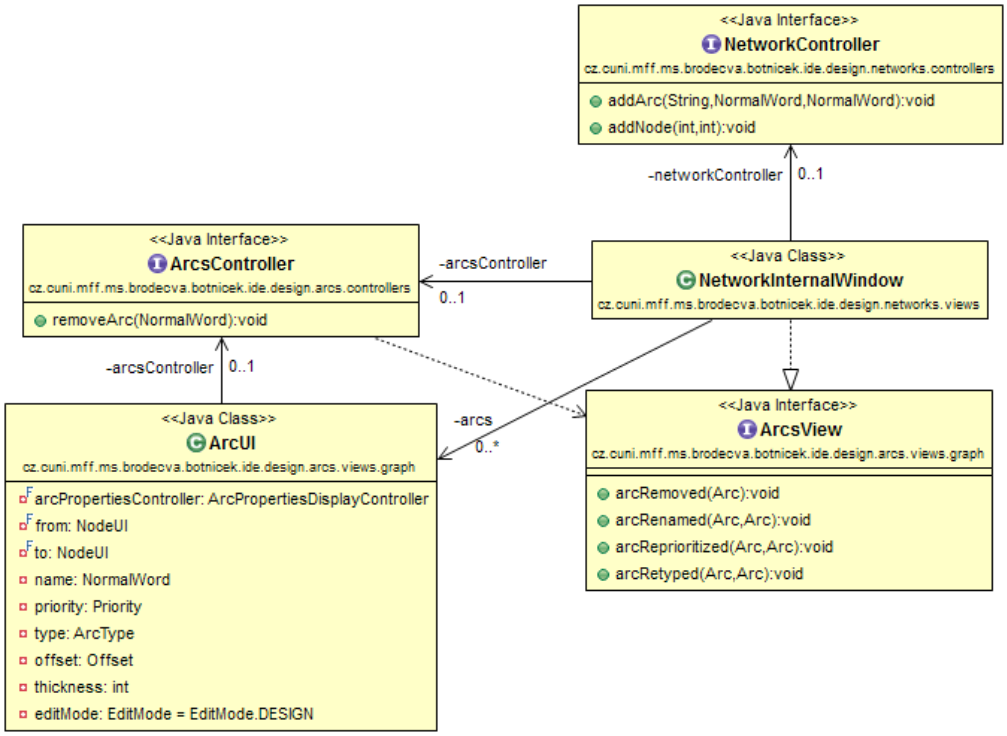
Obrázek 29 – hierarchie hran

Hierarchie hran je oproti uzlům jednoduchá (obrázek 29). Přestože všechny hrany dědí od AbstractArc, která obsahuje obecně platná pole jako název hrany, její prioritu a nadřazenou síť, tak nejsou jejich třídy přímými potomky. Ačkoli tomu tak nebylo v počátcích vývoje, tak je tomu vždy prostřednictvím AbstractCodeArc, která udržuje kód, který je vykonáván při průchodu sítí po hraně. V původním návrhu totiž existoval jeden speciální druh hrany, který kromě toho, že netestoval žádnou podmínku (jako to dělá přechodová hrana), tak navíc ani nevykonával žádnou činnost. Tedy šlo de facto o identitu uzlů (z pohledu umístění v grafu, pokud se zanedbají typy uzlů). Tento druh byl ale v průběhu vývoje vypuštěn, neboť jde snadno simulovat přechodem s prázdným kódem.

Nejjednodušší je již zmiňovaná přechodová hrana TransitionArc, která ke své funkcí nepotřebuje žádný vlastní atribut. Z hlediska zpracování interpretem AIML nejpřirozenější je PatternArc, která obsahuje složený vzor pattern a that. Ty se při kompilaci do AIML převedou přímo na vzor pattern a that kategorie. RecurentArc stačí k popisu dokonce jen jedna položka target, popisující vstupní uzel, jenž je cílem zanoření. To, že je tento cíl existujícím vstupním uzlem je ověřováno systémem při vložení instance tohoto druhu hrany.

Funkcí podobné hrany CodeTestArc a PredicateTestArc dle očekávání dědí od společného předka AbstractTestArc, který uchovává prostý vzor, oproti němuž je testovaná hodnota, ať už je to vyhodnocený úsek kódu či hodnota uložená

v predikátu (s případným předzpracováním). Konečně převod mezi hranami obstarává výchozí implementace rozhraní ArcModifier, jež vezme původní hranu, Class objekt jejího nového druhu a pole argumentů nutných ke konstrukci a pomocí reflexe zavolá tovární statickou metodu ve třídě druhu. Na rozdíl od paralelní třídy NodeModifier ovšem nemusí nijak zjišťovat, zda-li je daný převod validní, neboť druhy hran jsou zaměnitelné, aniž by to mělo jiný než místní dopad.



Obrázek 30 – přidávání, zobrazení a odebírání hran v grafu

Reprezentace hrany v grafu ArcUI (obrázek 30 vlevo dole) se odvíjí od krajních uzlů, mezi nimiž je vedena spojnice. Protože je oblast okolo spojnice vyznačena uzly, dojde při jejich přemístění i k přepočítání polohy spojnice. To je zabezpečeno tak, že každá instance NodeUI si pamatuje, jaké instance ArcUI z něj vychází či do něj vchází a pokud je to nutné, o změně polohy je zpraví. Problémem, který bylo nutné vyřešit, je zobrazení protisměrných hran, tj. hran, u nichž výchozí uzel jedné je koncovým uzlem druhé a naopak. Místo toho, aby se dělily o oblast spojnice krajních uzlů, a následný výběr chtěné hrany byl řešen dodatečně, bylo rozhodnuto je vést jako úseky tečen na kruhy vyznačující oba uzly s tím, že toto přímé řešení je pro prostý graf dostačující. Pro to je ale nutné zabezpečit, že

existující hrana reaguje na přidání či odebrání protisměrné hrany. Podobně jako s pohybem uzlu, jsou hrany uzlem, ke kterému byly připojeny, zpraveny o přidání protisměrné hrany. Pak dojde k posunu o poloměr kruhu uzlu ve směru kolmém na spojnici. Obdobně pokud je jedna z protisměrných hran odstraněna, je její protějšek zpraven a posune se zpět na spojnici.

Pro zobrazení vlastností hrany byly vyvinuty modifikace běžných formulářových komponent, nacházející se v design.arcs.view.properties, založené buďto na obecně použitelných komponentách (byť též pro tuto příležitost vytvořených) z utils.swing.components či XMLEditorKitu pro panel JEditorPane z knihovny Bounce. Mezi takové patří například ReferenceHintingComboBox, rozšíření JComboBoxu, které dokáže napovídat názvy dostupných uzlů k zanoření, či textová pole a oblasti podporující okamžitou syntaktickou kontrolu (SimplePatternTextField, MixedPatternTextField a CodeEditorPane). Výše zmíněné komponenty jsou hojně užívány zejména v sourozeneckém balíku types, kde tvoří části panelů, které se mění v ArcPropertiesWindow při užití přepínačů druhu.

6.14. Syntaktická kontrola uživatelského vstupu

Volný textový vstup, který je zadáván do polí formulářů, podléhá validaci. Přestože některé typy uživatelského vstupu jsou z hlediska generování kódu robota neškodné, je vhodné na ně uživatele upozornit také, neboť by vedly k vytvoření zdrojového kódu, který by nebyl platným kódem jazyka AIML. Příkladem takového vstupu je kód, který se má provést v případě úspěchu testu: z hlediska generování nehraje žádnou roli, neboť je beze změny vložen do pracovního kódu vytvořeného Botníčkem. Ten je dokonce navržený tak, že za běhu odolá i manipulaci s obsahem predikátu topic, který je jinak esenciální pro chod interního zásobníku.

Uživatelský vstup je dodatečně validován ve všech polích na určení názvů prvků systému, napříč jeho vrstvami. Vstup, který nejde triviálním způsobem převést na validní (typicky malá na vyžadovaná velká písmena pro normální názvy), je dodatečně reklamován obvykle v podobě vyskakovacího chybového okna. Oproti tomu vstup, který je zadáván do okna s vlastnostmi ArcPropertiesWindow, je validován interaktivně, jak uživatel vpisuje znaky.

Především k interaktivní validaci uživatelského vstupu slouží validátory v balíku check. V jeho podbalíku common se nachází sdílený kód, který je využíván na integraci s tabulkou na zobrazení chyb, jež přijímá výsledky z validátorů všech významných datových typů. Validují se tedy normální slova (pro názvy uzlů, hran a predikátů), jednoduché (testovací výrazy) i složené vzory (vzory pattern a that) a konečně uživatelský kód šablony (kód prováděný při splnění podmínky hrany, testovaný kód či kód prováděný před testováním hodnoty predikátu). Každý typ k validování má svůj vlastní balík, který je rozdělen oproti běžnému uspořádání pouze na řadiče a model, pohled je pro všechny typy stejný a je jím zmíněná tabulka chyb v common. Model validátoru typů je dále rozdělen na balíky checker, validator a builder. Zatímco checker obsahuje samotnou logiku validátoru, tak v balíku validator je rozšířen o mechanismus zasílání zpráv o výsledku validace. Builder pak zajišťuje převod validovaného textu na objektovou reprezentaci.

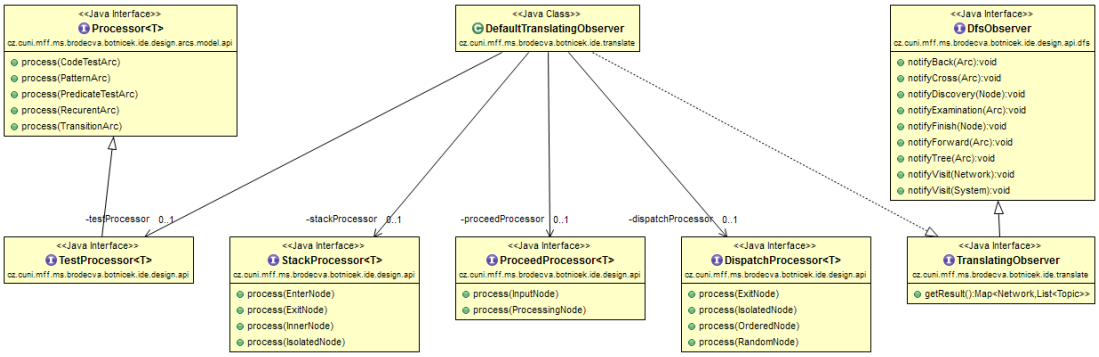
Validace normálních slov a prostých vzorů vychází přímo z definice. Pro zdrojový kód šablony se využívá přímo parser SourceParser z interpretu. Tomu je po nezbytné inicializaci předán kód šablony obalený základní kostrou AIML dokumentu. Chyby, které parser objeví, jsou pak zachytávány a předávány jako výsledky validace. Provádí se nejen kontrola správného utvoření , ale i vůči XML schématu AIML. Složený vzor se validuje stejně jako prostý s tím rozdílem, že potenciální výskyty značky bot jsou předávány validátoru kódu.

K výměně informací o chybách slouží instance tříd common.model.DefaulCheckResult, jež naplňuje rozhraní CheckResult. Kromě obvyklých popisů umístění chyby (řádek a sloupec), textového popisu (zpráva) a zdroje (což je nejčastěji objekt, který si vyžádal validaci) obsahuje tzv. předmět validace. Předmětem validace může být libovolný objekt, který by ovšem měl být ekvivalentní pro související validace. Tabulka výsledků užívá předměty pro určení chybových hlášení, která mají být aktualizována či v případě, že byl popsaný problém opraven, smazána.

6.15. Překlad systému sítí do modelu dokumentu jazyka AIML

Ačkoli je překlad zastřešen rozhraním Compiler z balíku compile, jehož

jediná metoda převádí systém sítí na seznamy AIML témat pro každou síť, jeho srdce se nachází v balíku translate. V něm je definovaný pozorovatel (další klasický návrhový vzor[39]) s názvem TranslatingObserver (obrázek 31), který umí sledovat průběh průchodu do hloubky systémem (podporovaný v design.api.dfs) a překládá při tom uzly a hrany, na které narazí, do podoby stromu odlehčeného objektového modelu dokumentu jazyka AIML.



Obrázek 31 – překládající pozorovatel a procesory uzlů a hran

Drží se při své činnosti návrhu z kapitoly 4. Má k tomu k dispozici sadu procesorů v podbalíku processors, které aplikuje podle objektu, jež je mu při průchodu nahlášen. K zpracování hrany vystačí procesor jediný, a to DefaultTestProcessor, který vrací seznam témat nutných k implementaci oznámeného druhu hrany. Pro uzly je třeba spolupráce až tří procesorů. První z nich, DefaultStackProcessor, vrací u výchozích a cílových uzlů instrukce pro vložení příslušné úklidové značky na zásobník (pro odstranění nenavštívených stavů při úspěšném průchodu mezi oběma typy uzlu v jedné síti).

Druhý v pořadí je DefaultDispatchProcessor. Ten u uzlů s náhodným uspořádáním odchozích hran (RandomNode) vrací instrukce, které nejprve pošlou prioritami zmnožený seznam hran do procedury pro náhodné uspořádání a pak se její výsledek umístí na zásobník. U uzlů s deterministický uspořádáním (OrederedNode) přímo seřadí hrany podle priorit, neboť se nemusí spoléhat na zpracování za běhu.

Poslední procesor DefaultProceedProcessor je velmi jednoduchý a nepracuje se zásobníkem. U procesních, tj. neblokujících uzlů (ProcessingNode), vrátí instrukci pro rekurzivní zpracování vstupu (zachyceného univerzálním záchytným vzorem pattern „*“), a tak dovolí při interpretaci výpočtu probíhat dále. U

vstupních, blokujících uzlů (InputNode) nevrací nic a interpretu pak nezbude než před dalším postupem vyčkat na zadání od konverzujícího uživatele.

Jakmile je průchod systémem hotový, je možné z překládajícího pozorovatele odečíst pro každou síť seznam témat, na která byla převedena. Ta pak už stačí jen vložit do kořenového prvku Aiml a dokument je připraven k převodu na kód. Pro správný chod robota je takto nutné ale zkompilovat navíc i knihovní kód, poskytovaný v balíku compile.library. První zde se nacházející třída Randomize vrací v metodě getLibrary (na základě maximální povolené priority hrany a větvícího faktoru) kód pro provádění náhodného výběru odchozích hran. Druhá knihovní třída Recursion svým příspěvkem zajišťuje mechanismus rekurzivního zanořování a vynořování z podsítí. Z ní odvozený kód provádí čištění zásobníku mezi úklidovými značkami, detekuje úspěšný i neúspěšný průchod sítí a vrací po něm manipulací zásobníku výpočet do správného místa.

6.16. Vytváření zdrojového kódu z modelu

Vytváření kódu z odlehčeného objektového modelu je řešeno v balíku render, kde výchozí implementace rozhraní Render po dodání prvku Element metodě render z odlehčeného objektového modelu vytvoří jeho textovou reprezentaci včetně vnitřního obsahu. Interně postupuje tak, že prochází podstromem modelu určeným parametrem Element do hloubky (tím bývá pro vypsání celého dokumentu kořenový prvek aiml). Návštěvník, který provádí výpis, akumuluje výsledek tak, aby se v něm ve správném pořadí vytvořil příslušný fragment dokumentu. Prvky jazyka AIML a jejich atributy jsou vypisovány s prefixy nastavenými v konfiguraci.

Pro potřeby exportování zdrojový kód prochází navíc formátovačem skrytým v balíku print, který jej převede do úhledné podoby. Pro generování zdrojového kódu, který je načten prostředím při testování robota, tento krok není nutný, a tak je vynecháván. Formátovač Printer je implementován třídou DefaultPrettyPrinter, která užívá na formátování standardní knihovny, jmenovitě dodané implementace rozhraní Transform v java.xml.transform.

6.17. Testovací běhové prostředí

Testovací běhové prostředí je postavené na již zmiňovaném dřívějším projektu – interpretu AIML. Ten je pro potřeby prostředí zapouzdřen v instanci modelového rozhraní Runtime ve stejnojmenném balíku. Instance Runtime umožňuje načítat části zdrojového kódu (vygenerované v předchozích krocích) a samotnou konverzaci pak spustit metodou run, jež vrací instanci rozhraní Run. Run poskytuje jedinou metodu tell, přijímající řetězec se vstupem uživatele.

Odpovědi od testovaného robota jsou distribuovány přes zavedený mechanismus událostí v podobě události SpokenEvent, jejíž posluchač jako argument přijímá výstup robota. Obdobně je postaráno i o výjimečné či chybové stavy skrze událost ExceptionalStateCaughtEvent. Události zachytává jediný dostupný prostředek pro interakci – panel se vstupním polem, potvrzovacím tlačítkem a přehledem historie konverzace TestPanel.

Velký vliv na chování robota za běhu mají nastavení běhového prostředí, dostupná ze stejnojmenné nabídky. Nastavení jazyka, robota a konverzace zde provedená jsou postupována interpretu. Stejně jako u změny v návrhu robota je i po jejich změně nutné spustit testování znovu, aby došlo k jejich předání nové verzi běhového prostředí.

6.18. Ukládání projektu

V prezentované verzi je uložení projektu pro pozdější užití vyřešeno serializací objektu Project. Serializací se v tomto případě rozumí automatická serializace dostupná v Javě[44]. Velkou výhodou tohoto řešení je kromě relativně nízkých implementačních nároků to, že dokáže spolehlivě a bez asistence zachytit a následně zrekonstruovat všechny rozumně definované objekty, včetně vzájemných referencí. To je nedocenitelné zejména u vysoce souvislých objektových struktur, jako jsou právě užité grafy sítí. V moderních verzích běhové prostředí Javy je navíc tato výchozí serializace pro potřeby prostředí dostatečně rychlá.

Pro zavádění serializace je velmi důležitý návrhový vzor proxy[45]. Dobrým příkladem jeho užití v projektu je již zmiňovaná výchozí implementace rozhraní Printer, která formátuje zdrojový kód. Místo toho, aby se serializoval

potenciálně paměťové nákladný objekt naplňující rozhraní javax.xml.transform.Transformer, je za veřejně viditelnou verzi třídy DefaultPrettyPrinter převáděna pouze její odlehčená soukromá proxy verze, jež obsahuje jen nezbytné konfigurační údaje. Během obnovení objektu ze serializované podoby je pak vytvořen nový objekt třídy DefaultPrettyPrinter s těmito údaji. Pouze je nutné pamatovat, že si automatická serializace s proxy objekty nedokáže při výskytu kruhových závislostí dobře poradit.

Řešení skrze výchozí serializaci ovšem není z pohledu přenositelnosti a robustnosti projektových souborů ideální. Při dalším vývoji by bylo vhodné přejít na textový, čitelný, jasně specifikovaný, přenositelný formát. Do té doby je z pohledu uživatele formát projektového souboru nespecifikován s tím záměrem, aby se nepodporoval vznik doplňkových nástrojů, které by na aktuálním nevhodném formátu závisely.

Ukládání stavu pracovního prostředí, přesněji rozměrů a polohy hlavního okna či vnitřních oken se sítěmi a podrobnostmi hran bylo pro prezentovanou verzi aplikace odloženo s tím, že byť není pro pohodlnou práci zcela nezbytné, tak by bylo vhodné je v dalších verzích zavést. Vhodným kandidátem na realizaci je Java Preferences API[46], což je ustálený prostředek pro na platformě nezávislé ukládání a načítání údajů tohoto druhu.

6.19. Testy

Stejně jako předcházející ročníkovém projekt interpretu, je i vývojové prostředí doplněno o množství jednotkových a integračních testů, které vznikaly v průběhu vývoje. Jsou spustitelné jako cíle sestavovacího souboru pro Ant. V projektu jsou umístěny v separátním adresáři tests, který ovšem kopíruje hierarchii balíčků s produkčním kódem. Toto řešení bylo oproti jiným (např. zvláštní balík test v každém balíku) zvoleno z důvodu snazšího importu testovaných a souvisejících tříd, kdy je možné pro třídy ze stejného balíku importující deklaraci vynechat. Též je možné omezit viditelnost některých tříd balíku na výchozí tak, aby sice byly dostupné pro sestavení testu, ale nikoli již z jiných balíků.

Je využíván široký aparát prostředků pro usnadnění testování. Kromě

standardního nástroje pro (nejen) jednotkové testování JUnit[47] je na vytváření atrap (dummies), pahýlů (stubs) a náhradních objektů (mocks) užívána knihovna EasyMock[48]. Nutnost absolutní izolace objektu někdy vyžaduje i manipulaci s instrukčním kódem Javy za běhu, například aby bylo možné vytvářet pomocné testovací objekty neměnných tříd. Proto našel uplatnění i nástroj PowerMock[49], jež ve spolupráci s EasyMock těmito schopnostmi disponuje.

Ucelené sady JUnit testů jsou dostupné v balíku cz.cuni.mff.ms.brodecva.botnicek.ide.utils.test. Pokrytí tříd testy bylo oproti vývoji interpretu, kdy byly nároky na bezvadné vyhodnocování velmi vysoké, voleno střízlivěji. Důraz je kladen na pokrytí tříd modelu a tříd, jež provádějí netriviální úkony. Naproti tomu třídy, které specifikují vzhled a chování grafického prostředí, a jež jsou notoricky těžko postihnutelné jednotkovými testy, jsou testovány minimálně a spoléhalo se u nich především na uživatelské testování. Integrační testy mapují především spolupráci validátorů a generátorů kódu.

6.20. Pomocné nástroje

Kromě dříve zmíněných nástrojů byl použity ještě následující:

- IzPack[50]: pro sestavení instalátoru.
- Eclipse[51] a ObjectAid UML Explorer[52]: jako IDE a nástroj pro generování UML diagramů tříd.

Závěr

V rámci bakalářské práce bylo nejprve provedeno zhodnocení jazyka AIML, stojící na úvodním přehledu jeho prostředků. Za pomoci srovnání s pokročilejšími dialogovými systémy byly pak zjišťovány možné směry rozšíření, při čemž byla obhájena volba řízení dialogu, především díky perspektivě realizace, při níž by nebylo nutné rozšiřovat či měnit specifikaci interpretu. Následně byly probrány dva klasické přístupy k řízení dialogu v dialogových systémech a prodiskutována vhodnost jejich adaptace pro potřeby robotů v AIML. Jako přínosnější řešení bylo zvoleno modelování dialogu pomocí grafů, speciálně výpočetním procesem probíhajícím v rozšířených přechodových sítích (RPS).

Implementace řízení dialogu založená na RPS stojí na využití dříve představených prostředků jazyka AIML. Klíčovým úspěchem bylo navržení kódování explicitního zásobníku (ve smyslu abstraktního datového typu). Díky němu je možné pro AIML přirozeným způsobem provádět rekurzi přerušovanou vstupy uživatele a simulovat nedeterministický průběh výpočtu RPS. Kvůli omezení na jediný dostupný zásobník byl též vyvinut způsob, jak mimo obsluhu jejich stavů na něm manipulovat i se samotnými sítěmi. Kromě rekurzivních hran, které se zanořují do podsítí, jsou pak podporovány i další testovací hrany, jež mají pro roboty napsané v čistém AIML smysl. Ačkoli je ve standardním AIML dostupná značka pro náhodný výběr s rovnoměrným rozdělením, tak bylo pro jemnější řízení konverzace naprogramováno řešení pro vážený náhodný výběr následující testující hrany, které doplňuje pevné určení pořadí dle priorit.

Teoretickou část práce se povedlo převést do podoby prakticky využitelného softwaru – vývojového prostředí Botníček. Jde o aplikaci běžící na platformě Java. Nabízí vícedokumentové grafické prostředí, v němž lze navrhovat podobu systému sítí a editovat jeho jednotlivé prvky. Dále poskytuje pomoc v podobě okamžité syntaktické kontroly kódu šablon a nabídek pro nastavení všech potřebných parametrů robota. Esenciální součástí je napojení na testovací interpret, jež je výsledkem předcházejícího ročníkového projektu a který svým striktním přístupem k vyhodnocování kódu zaručuje maximální možnou kompatibilitu napříč

produkčními interprety. Hotového robota lze jednoduše vyexportovat do zvoleného adresáře v podobě přímo interpretovatelných zdrojových souborů a snadno adaptovatelné konfigurace. Aplikace v současné verzi používá vlastní uzavřený formát pro uložení a pozdější načtení rozpracovaného stavu.

Přestože se povedlo naplnit určený cíl práce, z uživatelského hlediska by bylo dále vhodné rozšířit IDE o podporu pro začínající programátory v AIML. Těm by mohla pomoci nabídka standardních značek a nápověda při jejich vkládání do prvků sítě. Z funkčního hlediska se jeví jako jako opodstatněné dovolit vytvářet na aplikovaném systému řízení nezávislé procedury, jež by byly znovupoužitelné v testech a vykonávaném kódu, či zabezpečit spolupráci s externími procesory vstupu. Speciální pozornost při budoucím rozšiřování zaslouží také interakce s částmi robota, které se programátor rozhodne navrhovat bez podpory řízení dialogu.

Implementace řízení dialogu prokazuje překvapivou teoretickou sílu jazyka AIML, navzdory jeho četným praktickým omezením. I přes tento dílčí úspěch je však nutné připustit, že AIML je nástroj, jehož význam je bez nestandardních rozšíření pro další výzkum především historický. Na druhou stranu mu nelze upřít dostatečnou efektivitu, je li podpořen obsáhlou, promyšleně zkonstruovanou databází a především střízlivou volbou vhodného nasazení. Společně s jeho neoddiskutovatelnou přístupností z něj toto činí stále zajímavou volbu v komerční a hobby sféře, kde může výsledek práce, vývojové prostředí, nalézt příznivý ohlas.

Seznam použité literatury

[1] WILCOX, Bruce. *Gamasutra - Beyond Façade Pattern Matching for Natural Language Applications* [online]. [vid. 21. září 2014]. Dostupné z: http://www.gamasutra.com/view/feature/6305/beyond_façade_pattern_matching_.php

[2] LOEBNER, H. *Home page of the Loebner prize* [online]. 2003 [vid. 21. září 2014]. Dostupné z: http://www.loebner.net/Prizef/loebner-prize.html

[3] PANDORABOTS INC. *What are some of the potential applications for the Pandorabots API?* [online]. [vid. 22. listopad 2014]. Dostupné z: https://developer.pandorabots.com/#features-hover

[4] WALLACE, Richard, Noel BUSH, Thomas RINGATE, Anthony TAYLOR, Jon BAER a Bush NOEL. *Artificial Intelligence Markup Language (AIML) Version 1.0.1 A.L.I.C.E. AI Foundation Official AIML 1.0.1 standard Adopted October 30, 2011 (rev 008)* [online]. 2011 [vid. 9. leden 2014]. Dostupné z: http://alicebot.org/TR/2011/WD-aiml/index.html

[5] WALLACE, Richard. *Alicebot AIML 2* [online]. [vid. 21. září 2014]. Dostupné z: http://alicebot.blogspot.cz/2013/01/aiml-20-draft-specification-released.html

[6] WALLACE, R. The elements of AIML style. *Alice AI Foundation*. 2003.

[7] BERNERS-LEE, T a R FIELDING. RFC3986 - Uniform Resource Identifier (URI): Generic Syntax. *Internet Official Protocol Standards (STD 1)*. 2005, s. 1–62. ISSN 20701721.

[8] BRAY, T a J PAOLI. Extensible Markup Language (XML). … . *org/TR/1998/REC-xml- ...* [online]. 2006, s. 115–146. Dostupné z: doi:10.1007/978-1-4302-0187-8_6

[9] CLARK, J. Xsl transformations (xslt). … *Consortium (W3C). URL http://www. w3. org/TR/xslt*. 1999.

[10] SCERRI, Darren a Alexiei DINGLI. Analysing Input in Dialog Systems. 2012, roč. 1, č. 2, s. 37–43.

[11] A.L.I.C.E. AI FOUNDATION, Inc. *AIML Implementations* [online]. [vid. 7. listopad 2014]. Dostupné z: http://www.alicebot.org/downloads/programs.html

[12] PATRICK, Blackburn, Johan BOS a Kristina STRIEGNITZ. *Learn Prolog Now! - Negation as Failure* [online]. 2012 [vid. 24. listopad 2014]. Dostupné z: http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse45

[13] JOKINEN, K a M MCTEAR. Spoken Dialogue Systems. *Synthesis Lectures on Human ...*. 2009.

[14] XU, Wei a Alexander I. RUDNICKY. Task-based dialog management using an agenda. *ANLP/NAACL 2000 Workshop on Conversational systems* -. 2000, roč. 3, s. 42–47.

[15] GRAHAM, Paul. *On LISP: Advanced Techniques for Common LISP* [online]. 1993. ISBN 0130305529. Dostupné z: doi:10.1055/s-0030-1263208

[16] WOODS, W. A. *Transition network grammars for natural language analysis* [online]. 1970. Dostupné z: doi:10.1145/355598.362773

[17] *The COMPANIONS project* [online]. [vid. 24. listopad 2014]. Dostupné z: http://www.companions-project.org/

[18] UNIVERSITY OF SHEFFIELD. *Dialogue Manager Senior Companion*. 2008

[19] ORACLE CORPORATION. *Pattern (Java Platform SE 7)* [online]. [vid. 24. září 2014]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

[20] SPRINGWALD SOFTWARE. *GaitoBot - aiml chatbot hosting* [online]. [vid. 24. listopad 2014]. Dostupné z: https://www.gaitobot.de/gaitobot/

[21] DUBUQUE, Garry. *AIMLpad Home Page* [online]. [vid. 24. listopad 2014]. Dostupné z: http://program-n.sourceforge.net/

[22] *Masterlist of AIML Editors* [online]. Dostupné z: https://www.chatbots.org/ai_zone/viewthread/1128/

[23] GOSLING, J a H MCGILTON. *The Java language environment*. 1995.

[24] BUSH, Noel. *Program D - aitools* [online]. [vid. 21. září 2014]. Dostupné z: http://aitools.org/Program_D

[25] ORACLE CORPORATION. *JavaFX FAQ* [online]. [vid. 21. září 2014]. Dostupné z: http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6

[26] ROSS, Ivan. *Java+Swing in 2013* [online]. [vid. 21. září 2014]. Dostupné z: http://java.dzone.com/articles/javaswing-2013-it-worth-it

[27] ORACLE CORPORATION. *Java XML Technology Enhancements* [online]. [vid. 21. září 2014]. Dostupné z: http://docs.oracle.com/javase/7/docs/technotes/guides/xml/enhancements.html

[28] PANDORABOTS INC. *Free A.L.I.C.E. AIML Set* [online]. [vid. 3. září 2014]. Dostupné z: https://code.google.com/p/aiml-en-us-foundation-alice/

[29] ORACLE CORPORATION. *Properties (Java Platform SE 7)* [online]. [vid. 7. listopad 2014]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html#load%28java.io.Reader%29

[30] DANKERT, Edwin. *Bounce XMLEditorKit* [online]. 2007 [vid. 25. listopad 2014]. Dostupné z: http://www.edankert.com/bounce/xmleditorkit.html

[31] CAMICK, Rob. Wrap Layout. *Java Tips Weblog* [online]. [vid. 26. listopad 2014]. Dostupné z: http://tips4java.wordpress.com/2008/11/06/wrap-layout/

[32] GOOGLE INC. *PhilosophyExplained - guava-libraries - Some points on the Guava philosophy, explained* [online]. 2012 [vid. 25. listopad 2014]. Dostupné z: https://code.google.com/p/guava-libraries/wiki/PhilosophyExplained

[33] MOODIE, M. Pro Apache Ant. *Book*. 2005, s. 341. ISSN 9780951169568.

[34] ECKSTEIN, R. Java se application design with mvc. *Sun Microsystems Inc*. 2007.

[35] DALTON, Conan. *Enforce strict type safety with generics* [online]. [vid. 24. září 2014]. Dostupné z: http://www.javaworld.com/article/2072949/core-java/enforce-strict-type-safety-with-generics.html

[36] ORACLE CORPORATION. *WeakHashMap (Java Platform SE 7)* [online]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/java/util/WeakHashMap.html

[37] DIGIA PLC. *Signals & Slots* [online]. [vid. 24. září 2014]. Dostupné z: http://qt-project.org/doc/qt-5/signalsandslots.html

[38] WORLD WIDE WEB CONSORTIUM. W3C Document Object Model. 2006.

[39] GAMMA, Erich, Richard HELM, Ralph E. JOHNSON a John VLISSIDES. *Design patterns: elements of reusable object-oriented software* [online]. 1995. ISBN 0201633612. Dostupné z: doi:10.1093/carcin/bgs084

[40] ORACLE CORPORATION. *javax* [online]. [vid. 24. září 2014]. Dostupné z: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

[41] PEYTON, Simon. Haskell 98 Language and Libraries The Revised Report. *Language* [online]. 2003, roč. 13, s. 1–277. ISSN 0956-7968. Dostupné z: doi:10.1017/S0956796803000315

[42] CLIFTON, Curtis, Todd MILLSTEIN a Craig CHAMBERS. MultiJava : Design Rationale , Compiler Implementation , and Applications. *Design* [online]. 2006, roč. 28, s. 517–575. ISSN 01640925. Dostupné z: doi:10.1145/1133651.1133655

[43] NUTTYCOMBE, Kris. *Correcting the Visitor pattern* [online]. [vid. 24. září 2014]. Dostupné z: http://logji.blogspot.cz/2012/02/correcting-visitor-pattern.html

[44] OPYRCHAL, L. a A. PRAKASH. Efficient object serialization in Java. *Proceedings. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware* [online]. 1999. Dostupné z: doi:10.1109/ECMDD.1999.776421

[45] BLOCH, Joshua. *Effective Java* [online]. 2008. ISBN 9780321356680. Dostupné z: doi:10.1016/B978-075067929-9/50038-5

[46] ORACLE CORPORATION. *Preferences API Overview* [online]. [vid. 29. říjen 2014]. Dostupné z: http://docs.oracle.com/javase/7/docs/technotes/guides/preferences/overview.ht

ml

[47] JUNIT. *JUnit - About* [online]. [vid. 24. listopad 2014]. Dostupné z: http://junit.org/index.html

[48] TODD, Alistair a Jérémy BUGET. *EasyMock* [online]. [vid. 24. listopad 2014]. Dostupné z: http://easymock.org/

[49] HALEBY, Johan. *Motivation - powermock* [online]. [vid. 24. listopad 2014]. Dostupné z: https://code.google.com/p/powermock/wiki/Motivation

[50] *IzPack - Package once. Deploy anywhere.* [online]. [vid. 24. listopad 2014]. Dostupné z: http://izpack.org/

[51] THE ECLIPSE FOUNDATION. *Eclipse desktop & web IDE* [online]. 2014 [vid. 24. listopad 2014]. Dostupné z: http://www.eclipse.org/ide/

[52] OBJECTAID LLC. *ObjectAid UML Explorer - Class Diagram* [online]. [vid. 24. listopad 2014]. Dostupné z: http://www.objectaid.com/class-diagram

Seznam použitých zkratek

AIML	Artificial Intelligence Markup Language (značkovací jazyk pro umělou inteligenci)
DAF	Dialogue Action Forms (jedna z aplikací rozšířených přechodových sítí)
IDE	Integrated developement environment (integrované vývojové prostředí)
JRE	Java Runtime Environment (běhové prostředí platformy Java)
MDI	multiple document interface (rozhraní s více dokumenty; grafické rozhraní, ve kterém se okna nachází pod jediným rodičovským)
MVC	Model-View-Controller (Model – pohled – řadič; softwarová architektura oddělující funkční části uživatelské aplikace)
RPS	rozšířené přechodové sítě (v orig. Augmented Transition Networks; datová struktura)
XML	Extensible Markup Language („rozšířitelný značkovací jazyk“; obecný jazyk vyvinutý konsorciem W3C)

Přílohy

Příloha A – Obsah CD

- **demo**
  - canteen.btk – *demonstrační projektový soubor*
- **instalace**
  - botnicek-installer.jar – *multiplatformní (Windows, UNIX, OS X) instalátor*
  - botnicek.jar – *přímo spustitelná aplikace*
- **javadoc** – *generovaná dokumentace (lze vytvořit i s pomocí nástroje Ant po dodání konfiguračního souboru javadoc.xml v kořenovém adresáři Botnicek IDE)*
- **ostatní**
  - rocnikovy-projekt
    - dokumentace – *dokumentace předcházejícího ročníkového projektu*
- **text**
  - cuni_2014_brodec_vaclav.pdf – *text práce*
- **zdrojove-soubory**
  - botnicek
    - Botnicek – *kořenový adresář zdrojových souborů knihovny interpretu (předchozí projekt)*
    - Botnicek IDE – *kořenový adresář zdrojových souborů výsledku bakalářské práce, vývojového prostředí*