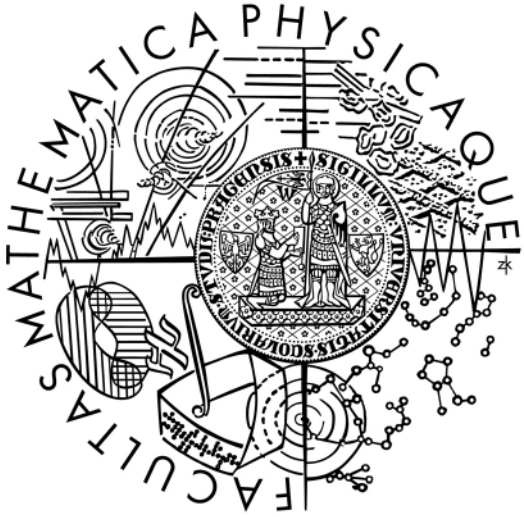
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Martin Ralbovský

Využití doménových znalostí při aplikacích GUHA procedur

Katedra softwarového inženýrství
Vedoucí diplomové práce: Doc. RNDr. Jan Rauch, CSc.
Studijní program: Informatika

Chtěl bych poděkovat zejména vedoucímu práce Doc. RNDr. Janu Rauchovi, CSc. za skvělé vedení diplomové práce a za cenné rady a připomínky. Dále bych chtěl poděkovat Ing. Vojtěchu Svátkovi, Dr. za cenné vedení a rady v oblasti ontologií. Chtěl bych také poděkovat všem členův vývojářského týmu Ferda, kteří mi svými radami a pomocí umožnili tuto práci dokončit. V neposlední řadě bych chtěl poděkovat Vratislavu Benešovi za pravidla *background knowledge* z oblasti pivovarnictví a Miroslavu Kudelovi za pomoc s medicínskými termíny.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 1. 8. 2006 Martin Ralbovský

Obsah

Název práce: Využití doménových znalostí při aplikacích GUHA procedur
Autor: Martin Ralbovský
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Doc. RNDr. Jan Rauch, CSc.
e-mail vedoucího: rauch@vse.cz
Abstrakt:

Práce vychází z předpokladu, že lze rozšířit možnosti GUHA procedur využitím vhodně reprezentovaných doménových znalostí. V práci jsou použity dva typy doménových znalostí: background knowledge a ontologie. Background knowledge je poměrně neznámý avšak perspektivní typ znalostí obsahující pravidla ve slovní formě od doménových expertů. Práce popisuje tento typ znalosti a zavádí vhodnou formalizaci pravidel. Dále je popsán a implementován nástroj na validaci těchto pravidel. Implementace je provedena pomocí krabiček v prostředí Ferda, což je modulární prostředí na vizuální dobývání znalostí pomocí GUHA procedur. Práce také popisuje první praktické zkušenosti s nástrojem na validaci pravidel. Co se týče doménových ontologií, práce podstatným způsobem rozšiřuje současné poznatky o využití ontologií pro dobývání znalostí pomocí GUHA procedur. Jsou zde zmíněny a rozpracovány všechny minulé postupy využití ontologií, vymyšleny nové, a u vhodných postupů je navržena implementace v prostředí Ferda. Výsledky práce jsou posuzovány s ohledem na možné použití v systému Ever-Miner, což je nástroj příští generace dobývání znalostí.

Klíčová slova:

dobývání znalostí, doménové znalosti, metoda GUHA, ontologie, background knowledge, prostředí Ferda

Title: Usage of Domain Knowledge for Applications of GUHA Procedures
Author: Martin Ralbovský
Department: Department of Software Engineering
Supervisor: Doc. RNDr. Jan Rauch, CSc.
Supervisor's email address: rauch@vse.cz
Abstract:

We presume for this work, that the GUHA procedures' abilities can be extended by using domain knowledge in a suitable form. There are two types of domain knowledge used in this work: background knowledge and ontologies. Background knowledge is a relatively unknown, yet perspective type of domain knowledge containing rules from domain experts in a written form. The work describes this knowledge and introduces suitable rule formalization. Furthermore, a tool for rule validation is described and implemented. The implementation is done in the Ferda environment, a modular environment for visual GUHA data mining. The work also describes first practical experience with this tool. Concerning domain ontologies, the work greatly extends up-to-date knowledge of usage of ontologies in the GUHA data mining. There are all the former techniques for ontology usage mentioned and extended; in suitable cases the implementation design in Ferda environment is presented. The results of the work are discussed with respect to further usage in the Ever-Miner system, a next generation knowledge discovery tool.

Keywords:

data mining, domain knowledge, GUHA method, ontology, background knowledge, Ferda environment

1. Cíl a struktura diplomové práce

1.1. Cíl diplomové práce

Cílem diplomové práce je přispět k využití doménových znalostí při dobývání znalostí pomocí GUHA procedur.

Práce vychází z předpokladu, že lze rozšířit možnosti GUHA procedur v současné době implementovaných v rámci systému LISp-Miner využitím vhodně reprezentovaných doménových znalostí. Těmito znalostmi zde rozumíme znalosti o oblasti, které se týkají analyzovaná data. Vhodně reprezentované znalosti lze použít pro definici atributů, které vstupují do jednotlivých procedur. Lze je využít i pro odstranění vztahů vyjadřujících známé závislostí, které jsou součástí výstupu jednotlivých procedur. Další jejich použití je možné pro upřesnění zadání analytických úloh různých typů, například pro analytické úlohy typu „jaké výjimky z obecných znalostí platí v analyzovaných datech“.

Práce se zabývá dvěma typy doménových znalostí: *ontologiemi* a *background knowledge*. Cílem je vyzkoumat možnosti využití těchto znalostí a vhodné postupy implementovat v prostředí Ferda.

1.2. Struktura diplomové práce

Pro účely práce byl nastudován princip dobývání znalostí pomocí metody GUHA a autor se seznámil s nástroji implementující tyto metody. O tomto tématu hovoří 2. kapitola. Nejdříve se v ní mluví o obecných základech procedury GUHA. Poté se představí prostředí Ferda jako alternativní uživatelské prostředí pro systém LISp-Miner. Dále se definují GUHA procedury implementované ve Ferdovi, zejména 4FT, KL a CF. Kapitola poté obsahuje stručný popis systému Ever-Miner jakožto představitele příští generace nástrojů na dobývání znalostí pomocí metody GUHA. V závěru kapitoly představujeme první návrh implementace systému Ever-Miner v prostředí Ferda, který byl autorem vymyšlen v průběhu práce.

Typu doménové znalosti *background knowledge* je věnovaná třetí kapitola. Protože se v literatuře neexistuje práce na toto téma, kapitola je prvním pokusem o definici a formalizaci *background knowledge* v souvislosti s dobýváním znalostí pomocí GUHA procedur. Čtenář je nejdříve obeznámen s pojmem *background knowledge.* Po diskusi o nevhodnosti stávající formalizace *background knowledge* je vymyšlena formalizace nová, která velmi dobře využívá vlastností GUHA procedur. V závěru kapitoly po představení ontologií uvádíme srovnání ontologií a *background knowledge*.

Protože práce není prvním pokusem o využití doménových znalostí při dobývání znalostí, ve 4. kapitole je čtenář seznámen s pracemi na toto téma v minulosti a se širším kontextem problému. Kapitola identifikuje postupy a nápady z minulých prací, vysvětluje souvislosti a další možné použití těchto postupů.

Pro implementační část diplomové práce byla vybraná validace pravidel *background knowledge*. Této problematice se věnuje pátá kapitola. Čtenář se zde nejdříve dozví o argumentech proč implementovat validaci pravidel. Dále je podrobně rozebrán návrh implementace v prostředí Ferda, vytvořené moduly a algoritmy používané při validaci. Na závěr je provedena diskuze o užitečnosti a použitelnosti implementovaného nástroje.

Šestá kapitola pojednává o využití ontologií pro obohacení dobývání znalostí pomocí GUHA procedur. Kapitola zmiňuje všechny postupy využití ontologií z dosavadní literatury a přidává nové. U každého postupu je provedena diskuse o

vhodnosti automatizace v prostředí Ferda a u vhodných postupů je navržena jeho implementace v prostředí Ferda. Kapitola tímto podstatným způsobem rozšiřuje současné poznatky o využití ontologií pro dobývání znalostí pomocí GUHA procedur.

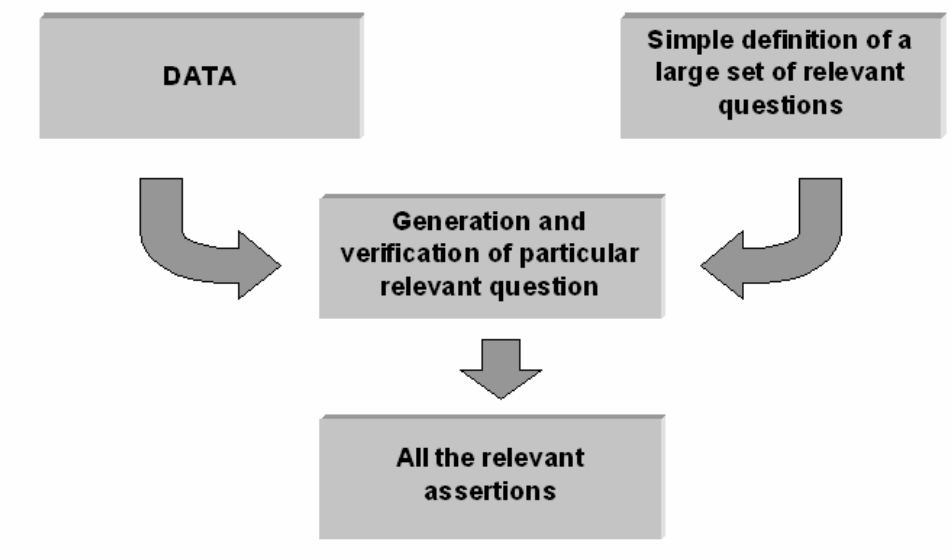
O testování implementovaného nástroje na validaci pravidel *background knowledge* na obsáhlejším praktickém úkolu pojednává kapitola 7. Jako úkol byla vybrána validace pravidel datového zdroje STULONG. Čtenář se v této kapitole dozví o jednotlivých fázích testování, a problémech a jejich řešeních, které se vyskytly v jednotlivých fázích. Dále jsou zde popsány zkušenosti s funkčností nástroje na validaci pravidel i systému Ferda. V závěru kapitoly je testování vyhodnoceno a jsou zde uvedeny návrhy a předpoklady pro další rozvoj validace pomocí *field knowledge*.

Poslední osmá kapitola je věnovaná závěru. Kapitula poskytuje souhrn vykonané práce a možnosti jejího využití do budoucna.

2. Použité metody a nástroje pro DZD

2.1. Metoda GUHA

GUHA (General Unary Hypotheses Automaton) je původní česká metoda pro získávání znalostí z dat. Teoretický rámec pro metodu vznikl v šedesátých letech a je popsán například v [12]. Základní princip metody je vyjádřen na obrázku 1.



Obrázek 1: Princip metody GUHA

Na realizaci GUHA metody se používají GUHA procedury. Jsou to programy, jejichž vstupy jsou analyzovaná data a jednoduché zadání zajímavých (relevantních) vzorů[1]. GUHA procedura automaticky generuje všechny vzory a testuje u nich, jestli jsou pravdivé ve vstupních datech. Výstupem procedury jsou všechny vzory, které jsou pravdivé v datech a nejsou součástí jednoduššího vzoru.

Nejvýznamnější GUHA procedurou je procedura ASSOC [12], která hledá asociační pravidla. Oproti klasickým asociačním pravidlům ASSOC může vybírat i pravidla odpovídající statistickým testům hypotéz (a jiné). Procedura 4FT, zásadní GUHA procedura pro tuto práci, je nejnovější implementací procedury ASSOC v rámci systému LISp-Miner[11]. Procedura bude podrobně vysvětlena v dalších odstavcích této kapitoly.

2.2. Systém Ferda

Od poloviny devadesátých let se na VŠE v Praze vyvíjí akademický softwarový systém pro výuku a výzkum v oblasti dobývání znalostí z dat LISp-Miner[37]. Je to systém, jehož základem jsou GUHA procedury pro dobývání znalostí, z nichž nejvýznamnější je procedura 4FT. Dále obsahuje procedury pro transformaci dat a proceduru strojového učení. Ačkoli je LISp-Miner úspěšný a rozvíjející se systém, svým uživatelským prostředím a některými svými vlastnostmi neodpovídá požadavkům na moderní systém pro dobývání znalostí. Jedná se zejména o požadavky jednotné uživatelské práce a větší modularity systému. Proto vznikla v minulých letech iniciativa na vytvoření nového vizuálního prostředí pro systém LISp-Miner[2]. Ferda vznikl jako

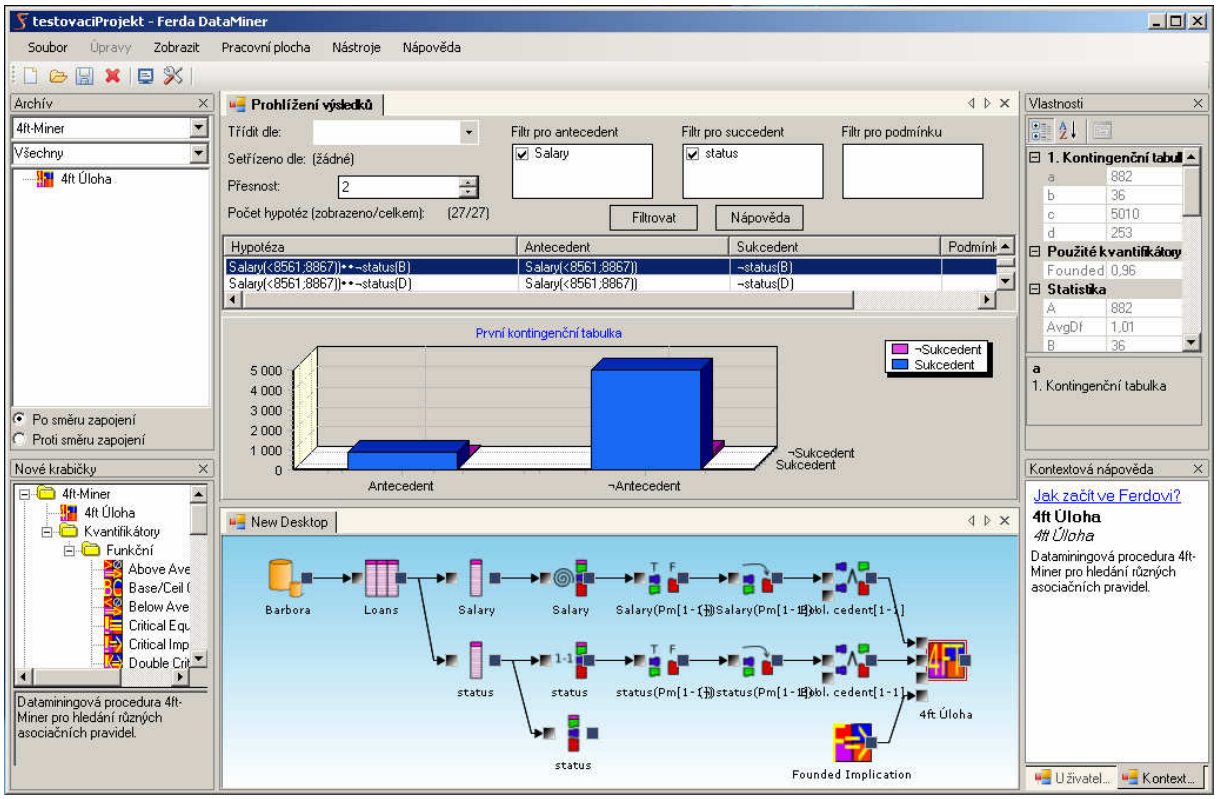
[1] Používáme slovo vzor jako nejpřesnější překlad anglického slova *pattern*. I když jsme si vědomi drobného významového posunu, nenašli jsme lepší český ekvivalent.
[2] Vzorem pro nové prostředí měl být komerční systém pro dobývání znalostí Clementine[2]

softwarový projekt na MFF UK a měl naplnit iniciativu na vznik nového vizuálního prostředí.

Ferda byl vyvíjen více než dva roky a v dubnu 2006 obhájen jako softwarový projekt na MFF. Prostředí poskytuje nový vizuální pohled na proces zadávání úlohy, jednotné úložiště pro všechny části úlohy vytvořené uživatelem, větší znovupoužitelnost postupů mezi jednotlivými procedurami, robustní architekturu podporující distribuované výpočty a mnohé další. Největší přínos však tvůrci Ferdy vidí tzv. *krabičkovém modelu*. Krabička ve Ferdovi je vizuální prvek a uživatel pomocí zapojování a nastavování vlastností krabiček tvoří úlohy. Má však také význam funkce, která pomocí vstupních parametrů počítá svůj výstup[1]. Výstup je definován pomocí funkčního rozhraní a toto rozhraní může sloužit jako vstupní parametr pro jinou krabičku. Stávající konstrukce úlohy může být tedy obohacena o nové nástroje v podobě krabiček, které se vhodně „vloží“ mezi krabičky stávající[2]. V této diplomové práci budou navrženy krabičky obsahující (v jisté formě) doménové znalosti tak, aby se dali zapojit do stávající konstrukce úloh. Vhodné krabičky budou implementovány (viz 5. kapitola).

Práce nepracuje přímo se systémem LISp-Miner a proto ho zde nebudeme popisovat. Protože však LISp-Miner zavádí terminologii, kterou převzal Ferda, je důležité, aby čtenář byl seznámen s významem jednotlivých termínů jako cedent či atribut, které se budou hojně vyskytovat. Dodatek B vysvětluje tyto termíny.

Článek [15] podrobněji popisuje systém Ferda. Systém je také volně ke stáhnutí v [34]. Na obrázku 2 vidíte prostředí Ferda a úlohu sestavenou v něm.



Obrázek 2: Prostředí systému Ferda

[1] V ostatních (komerčních) vizuálních systémech pro dobývání znalostí má vizuální prvek spíše význam části procesu dobývání znalostí, než abstrakce funkce. Ferda v tomto ohledu umožňuje širší možnosti zapojování krabiček za cenu menší přehlednosti. Podrobnosti na toto téma poskytuje [15].

[2] Tvůrci Ferdy vytvořili například ukázkovou krabičku *Body Mass Index*, která počítá tuto veličinu na základě dvou vstupních parametrů, výšky a váhy pacienta. Krabičku lze dále použít pro konstrukci úloh.

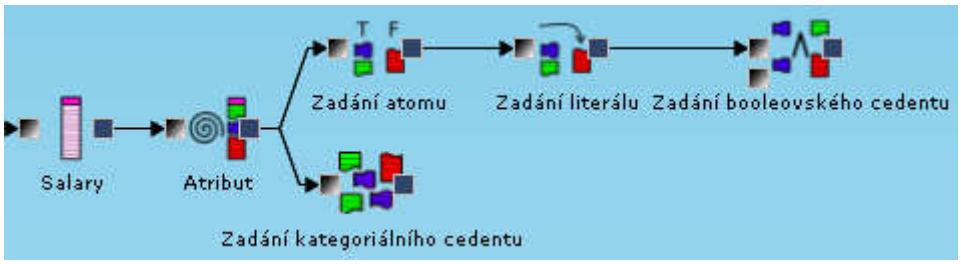
2.3. GUHA procedury implementované ve Ferdovi

V současné době je v prostředí Ferda implementovaných šest GUHA procedur: 4FT, KL, CF, SD4FT, SDKL a SDCF.

K samotnému generování hypotéz se používají součásti systému LISp-Miner, do budoucna se počítá s vlastní implementací jednotlivých GUHA procedur s možným rozšířením do relačních verzí. Je třeba také upřesnit terminologický rozdíl mezi pojmem *4ft-Miner* a 4FT. Pro účely této práce je 4FT GUHA procedura hledající asociační pravidla daného tvaru (jak je popsáno v podkapitole 2.3.1). Oproti tomu *4ft-Miner* je komponenta systému LISp-Miner, která implementuje proceduru 4FT. Vzhledem k tomu, že nezáleží na konkrétní implementaci procedury 4FT v systému Ferda[1], budeme dále používat pojem 4FT. Podobnou terminologií se práce řídí i v případě zbývajících GUHA procedur.

Z šesti implementovaných procedur se pro tuto práci nejvíce hodí uvažovat o procedurách 4FT, KL a CF. Zbývající tři procedury sice správně fungují, avšak jejich použití pro analytiky, kteří nejsou datovými experty, je prozatím problematické. Procedury SD4FT, SDCF a SDKL vznikly proto, že byly softwarové i jiné prostředky na jejich implementaci a předpokládalo se, že postupně vzniknou příslušné intuitivní kvantifikátory a typové úlohy, které využijí zatím neobjevené možnosti těchto procedur. Soustředíme se tedy na první tři procedury, u nichž existují osvědčené praktické možnosti využití a dlouhodobé zkušenosti s používáním.

Než popíšeme jednotlivé procedury, je důležité vysvětlit konstrukci úlohy ve Ferdovi, neboť je jiná než u systému LISp-Miner. Zajímá nás hlavně konstrukce „řetízku“ (zapojení posloupnosti krabiček různých typů do sebe) od sloupce až po zadání cedentu, protože tuto konstrukci můžeme použít při zadávání všech GUHA procedur. Obrázek 3 zobrazuje tuto konstrukci.



Obrázek 3: Zadání cedentů ze sloupce v prostředí Ferda

Celá konstrukce začíná u krabičky sloupec, která reprezentuje sloupec v tabulce (na obrázku 3). Dále uživatel na obrázku vytvořil atribut, což je výsledek kategorizace hodnot domény sloupce[2]. Zde se uživatel rozhodne, jestli vytvořit kategoriální cedent, který používá všechny kategorie, nebo booleovský cedent, který používá jenom některé kategorie atributu. Ve spodní větvi na obrázku uživatel zkonstruoval kategoriální cedent a může u něho dále nastavit minimální a maximální délku. V horní větvi uživatel pokračovat ve tvorbě booleovského cedentu[3]. Zatímco v systému LISp-Miner slouží dialog *Literal* k zadání typu koeficientu, znaménka a typu literálu, v systému Ferda je postup rozdělen do dvou krabiček. V první krabičce *Zadání atomu* se nastavuje typ

[1] Ať už se jedná o implementaci *4ft-Miner*, nebo plánovanou nezávislou implementaci ve Ferdovi popsanou v [17].

[2] K dispozici jsou krabičky dynamických ekvidistančních, ekvifrekvenčních a *each value one category* atributů a statický atribut, pro náš příklad jsme použili ekvifrekvenční atribut. Rozdíly a způsoby použití jsou dobře popsány ve [15].

[3] Správně bychom měli říkat zadání booleovského cedentu místo booleovský cedent a podobně používat slovo zadání i u jiných krabiček. Rozdíl je v tom, že zadání cedentu představuje „vzorec“ pro konstrukci cedentu, kdežto samotný cedent je konkrétní cedent použitý při generování úlohy.

koeficientu a délka atomu. V druhé krabičce *Zadání literálu* se nastavují věci týkající se samotného literálu: jeho typ a znaménko[1]. Dále si uživatel vytvořil krabičku *Zadání booleovského cedentu*, u které se dá nastavit minimální a maximální délka.

Čtenář může pro bližší seznámení s teorií za konstrukcí úloh použít [26] nebo [24] či se o něm dočíst v dodatku B.

2.3.1. Procedura 4FT

Procedura 4FT hledá asociační pravidla ve tvaru **A ~ S/P**, kde **A**, **S** i **P** jsou booleovské cedenty a ~ je 4FT kvantifikátor. Systém sestrojí čtyřpolní kontingenční tabulku antecedentu a sukcedentu za dané podmínky a testuje ji vůči kvantifikátoru, což je funkce (často statistická) definovaná nad čtyřpolní tabulkou. V současné době je pro proceduru 4FT definovaných 16 různých kvantifikátorů. [26] a [24] obsahují rozsáhlé teoretické poznatky o proceduře 4FT, popis sestavení 4FT úlohy ve Ferdovi je k dispozici v systému jako součást nápovědy.

2.3.2. Procedura KL

Procedura hledá vztahy typu **R ~ C/γ** (které už nejsou formálně považovány za asociační pravidla), kde **R** a **C** jsou kategoriální atributy představující řádkové a sloupcové atributy a **γ** je booleovský atribut značící podmínku. Konečně **~** je KL kvantifikátor. Tento vztah se nazývá *KL pattern*, nebo také *KL hypotéza*. V systému Ferda je implementovaných pět agregačních kvantifikátorů a šest funkčních kvantifikátorů, ze kterých je pro tuto práci nejzajímavější Kendallův kvantifikátor. Více o tomto kvantifikátoru i o proceduře KL se dozvíte v [25] a [28].

2.3.3. Procedura CF

Procedura CF hledá vztahy typu **~ R/γ**, kde **R** je kategoriální atribut a **γ** je booleovský atribut představující podmínku. Lze říct, že procedura hledá rozložení frekvencí atributu za jisté podmínky, které je nějakým způsobem zajímavé (to je vyjádřeno zadaným kvantifikátorem a jeho parametry). Hypotézy procedury CF také nejsou asociační pravidla, nazýváme je *CF hypotéza*, nebo *CF pattern*. Ferda nabízí pět agregačních kvantifikátorů (stejné jako u KL) a devět funkčních většinou statistických kvantifikátorů.

2.4. Systém Ever-Miner

Ever-Miner by měl být následníkem systému LISp-Miner a měl by podstatně rozšířit možnosti stávajícího systému. Již několik let se o systému uvažuje a vznikají stále přesnější představy o tom, jak by měl fungovat. Funkční požadavky i nástiny implementačních řešení jsou uvedeny [14], [27], [26], [29], [30] a [31]. Ever-Miner je prozatím hypotetický prostředek, který komunikuje s uživatelem, zkoumá data z různých pohledů a snaží se nalézt na základě všech vstupních údajů dosud nepoznané souvislosti. Součástí systému je i soustava typových úloh, která má pomoci uživateli řešit nejčastější problémy. Důraz je kladen hlavně na použití expertních systémů, které používají různě typy znalostí pro sestavování úloh[2], a opakovanému běhu různých procedur k dosažení optimálních výsledků. Tyto nástroje by si měl uživatel nastavovat pomocí „wizardů“, což jsou uživatelsky přívětivé komponenty, s jejichž pomocí systém postupně získává informace o záměrech uživatele.

---

[1] Systém LISp-Miner například neumožňuje tvorbu podmnožin délky 1 a 3. V budoucí implementaci procedur nezávislé na LISp-Miner genech bude možné zapojit do krabičky *Zadání literálu* více různých krabiček *Zadání atomu*, z nichž každá může tvořit podmnožiny jiné délky.

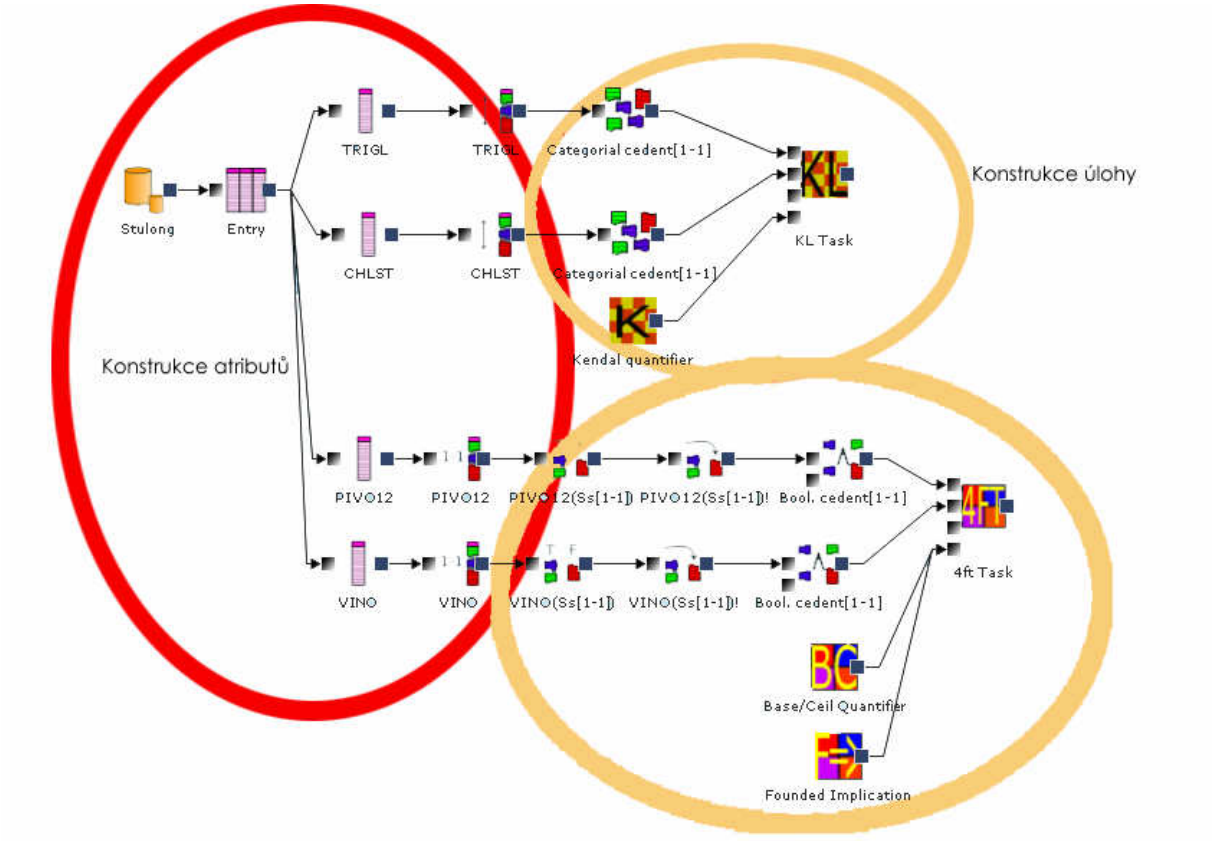
[2] Jedná se o popis dat, *background knowledge* nebo analytické zprávy, vše ve strojově čitelné formě.

Je na místě zmínit (nebylo tak učiněno v [15]), že systém Ferda již splnil některé požadavky na Ever-Miner (jako jednotné úložiště všech úloh a možnost uchování typových úloh). Protože však v době vzniku konceptu Ever-Miner Ferda neexistoval, stále považujeme Ever-Miner za nástupce systému LISp-Miner, nikoliv systému Ferda. Zároveň se ukazuje, že Ferda díky své architektuře, modularitě a rozšiřitelnosti by mohl být vhodným kandidátem pro implementaci funkčních požadavků Ever-Miner.

2.4.1. Implementace Ever-Miner v prostředí Ferda

Pro účely této práce i prací na ní navazujících byl vymyšlen návrh architektury systému Ever-Miner postavené na systému Ferda. I když je návrh výsledkem této práce a tudíž se příliš nehodí do kapitoly popisující prostudované metody a literaturu, byl zde zahrnut, neboť se to hodí z důvodu navazování jednotlivých témat do sebe a lepšího pochopení pro čtenáře.

Následující odstavce popisují základní vrstvy nové architektury a návaznost na systém Ferda. Architektura systému Ferda, zejména jeho *krabičkový model*, se jeví jako vhodná (a také jediná existující) platforma pro vývoj systému Ever-Miner. Výchozí situace je zobrazena na obrázku 4.



Obrázek 4: Znázornění Ever-Miner basic úloh

Na obrázku vidíme konstrukci dvou úloh v prostředí Ferda. Obě úlohy zjišťují závislosti v databázi STULONG[8][1] zkoumající výskyt ateroskleózy. První úloha

[1] Studie STULONG byla realizovaná na II. Interní klinice První lékařské fakulty Karlovy Univerzity a Všeobecné Fakultní nemocnice Univerzity Karlovy, U nemocnice 2, Praha 2 (vedoucí Prof. M. Aschermann, MD, SDr, FESC), pod dohledem Prof. F. Boudíka MD, ScD, ve spolupráci s M. Tomečkovou, MD, PhD, a Ass Prof J. Bultasem, MD, PhD. Data byly převedeny do elektronické podoby pracovištěm EuroMISE Karlovy Univerzity a Akademie věd (vedoucí Prof. J. Zvárová, DrSc). Zdroj dat je na webových stránkách http://euromise.vse.cz/challenge2004. V současné době je analýza dat podporovaná grantem Ministerstva školství, mládeže a tělovýchovy České republiky č. LN 00B 107.

ověřuje závislost hladiny cholesterolu a triglicerid pomocí procedury KL, druhá úloha závislost konzumace piva a vína u zkoumaných pacientů pomocí procedury 4FT.

Základem celé budoucí architektury je Ferdův *krabičkový model*. Tuto vrstvu pojmenujeme *Vrstva 0 – Ferdův krabičkový model*. Pomocí krabiček Ferdy (ať už stávajících, implementovaných či navržených v této práci, nebo dalších) a pomocí mechanizmů nabízených ve Ferdovi (zejména možnost nastavování vlastností krabiček pomocí zásuvek) lze sestrojit jakoukoli úlohu dobývání znalostí pomocí GUHA procedur programově, tedy pouhým vytvořením, zapojením a nastavením krabiček. To je hlavní důvod pro použití Ferdy k implementaci funkčních požadavků Ever-Miner.

Druhá vrstva systému Ever-Miner je prozatím pracovně nazvaná *Vrstva 1 – EverMiner basics*. Tato vrstva bude plnit pouze 2 úkoly, které jsou však nezbytné pro veškeré další úlohy systému Ever-Miner: automatická konstrukce atributů a automatická konstrukce úlohy na dobývání znalostí. Na obrázku 3 jsou tyto úkoly vyznačené červeným a žlutým oválem. Červený ovál zahrnuje část řetízku Ferdy od definice datového zdroje až po konstrukci atributů. Dva žluté ovály reprezentují dvě výše popsané úlohy na dobývání znalostí, tedy tvorbu cedentů a jejich zapojení do krabičky GUHA procedury. Důvodem k oddělení konstrukce atributů od konstrukce jednotlivých úloh je rozdílná konstrukce úlohy pro různé GUHA procedury při použití stejných atributů. Konstrukce atributů (kategorizace hodnot domény sloupců zkoumané datové matice) bývá většinou pokaždé stejná a vychází z povahy dat[1]. Systém tedy může atributy vytvořit pouze jednou a dále je používat nezměněné. Naproti tomu konstrukce úlohy probíhá pokaždé jinak. Rozdíl je patrný například z typů cedentů, které slouží jako vstup pro jednotlivé procedury (může se jednat o kategoriální cedenty, booleovské cedenty, nebo jejich kombinace).

Předpokládáme, že *Vrstva 1 – EverMiner basics* bude pomocí dalších znalostí o datech (zejména pomocí vhodných ontologií) konstruovat atributy i úlohy bez zásahu uživatele. Některé návrhy na možné použití ontologií jsou uvedeny v 6. kapitole tato práce. Automatickou konstrukci atributů a úloh můžeme dále použít k řešení složitějších problémů. Jedná se například o potvrzení platných vztahů v datech, hledání výjimek v datech nebo nalezení optimálního množství nových vztahů v datech. Více o typových úlohách Ever-Miner pojednává [31]. Těmito „nejobecnějšími“ úkoly se bude zabývat poslední vrstva systému: *Vrstva 2- EverMiner úlohy*. Vrstva bude využívat opakovaných zadání a běhů procedur (z předchozí vrstvy) a pomocí expertních systémů povede k nalezení nejlepších výsledků. Bohužel fungování této vrstvy je stále v konceptuálním stádiu a s implementací se v nejbližší budoucnosti nepočítá[2].

Veškeré implementační návrhy krabiček nebo soustavy krabiček této práce budou zkoumány i v závislosti na zmíněnou architekturu systému Ever-Miner s jasným zařazením a zdůvodněním, do které vrstvy daný návrh patří.

[1] Více o tomto problému pojednává část 6.3.

[2] Jestliže chceme použít vizuální prostředí Ferdy i pro systém Ever-Miner, je potřeba udělat některé změny v programu. Za hlavní považujeme možnosti tvoření „nadkrabiček“, tedy krabiček, které v sobě budou obsahovat řetízky základních krabiček (z vrstvy 0). Například tvorba atributů datového zdroje, na obrázku 4 naznačená červeným oválem, by mohla být jedna z těchto krabiček. V současné době sice existuje mechanizmus zabalování a rozbalování zásuvek krabiček, tento mechanizmus však pouze skrývá na pracovní ploše některé krabičky před uživatelem, ale uživatel musí zapojovat a nastavovat tyto krabičky před tím, než je skryje. U „nadkrabičky“ by měl uživatel nastavovat vlastnosti pro celou skupinu krabiček a podle nastavení vlastností se vytvoří konkrétní řetízek krabiček vrstvy 0. [16] má být prací na toto téma.

3. Background knowledge

Při dobývání znalostí pomocí GUHA procedur se jeví výhodné (i z hlediska CRISP-DM metodologie [3]) mít co nejvíce znalostí o struktuře a charakteru zkoumaných dat. Informace o doméně, ze které data vychází, nám mohou v tomto úkolu značně pomoci. Právě doménové znalosti shromažďují informace o určité doméně. V současné době se uvažuje zejména o použití dvou typů doménových znalostí: ontologie a *background knowledge*.

Doménové ontologie jsou nejznámější a nejrozšířenější druh doménových znalostí. Pro zachycení ontologií existuje několik jazyků a mnoho nástrojů, které s nimi umí pracovat. V neposlední řadě existuje velké množství ontologií popisujících jednotlivé domény, například medicínská ontologie UMLS [39]. Velmi zajímavé doménové ontologie lze nalézt také v [6]. V minulosti byly provedeny pokusy o využití ontologií při dobývání znalostí, o kterých podrobněji hovoří 4. kapitola. Proto nelze v této práci ontologie ignorovat a budeme se snažit vypracovat či zlepšit způsoby jejich využití.

Background knowledge je typ znalostí jen zřídka zmiňovaný v literatuře. Jedná se však o perspektivní typ znalostí, neboť jsou to znalosti vycházející přímo od doménového experta, který nemusí používat složité prostředky pro jeho formulaci.

Tato kapitola se věnuje primárně *background knowledge*, neboť si myslíme, že tento typ znalostí je velmi dobře použitelný při dobývání znalostí pomocí GUHA procedur. Čtenář je nejdříve obeznámen s pojmem *background knowledge.* Protože neexistuje žádná vhodná formalizace pravidel *background knowledge* pro GUHA procedury, diskutujeme dále v kapitole o použití stávající formalizace pravidel *background knowledge* pomocí kvalitativních modelů. Ukáže se, že tato formalizace není vhodná pro GUHA procedury a proto je vymyšlena nová formalizace, která lépe využívá vlastnosti dobývání znalostí pomocí GUHA procedur. V závěru kapitoly jsou po krátkém představení ontologií srovnány ontologie a *background knowledge* a navrženy způsoby reprezentace *background knowledge* pomocí ontologií. Jsou zde také uvedeny argumenty, proč je dobré posuzovat *background knowledge* jako samostatný typ doménových znalostí.

3.1. Úvod do background knowledge

Background knowledge nebo také *field knowledge*, česky nejpřesněji znalosti z oboru, můžeme definovat jako souhrn různých slovních pravidel, které platí v určitém oboru a jsou tam přijímány jako obecná znalost. Pravidla mohou definovat funkční závislost mezi veličinami, mohou vyjadřovat souvislosti mezi entitami v doméně, nebo říkat něco o jejich chování.

Dobrý příklad sady pravidel *background knowledge* se nachází v projektu STULONG[8], který se zabývá zkoumáním aterosklerózy. Pomocí expertů na danou doménu (v tomto případě lékařů) byla vypracovaná sada 36 pravidel popisujících známé a obecně přijímané souvislosti mezi jednotlivými entitami jako *„Jestliže roste krevní tlak, tak roste riziko cévní mozkové příhody“*, nebo *„Pokud roste úroveň cholesterolu, pak roste i úroveň triglyceridů.“* Dodatek A obsahuje tyto a další příklady pravidel *background knowledge*, všechna pravidla na které narazíme při vypracovávání diplomové práce, budou k dispozici v tomto dodatku.

Za účelem získání jiných možných pravidel *background knowledge* byl kontaktován expert z domény maloobchodní prodej piva Vratislav Beneš. Podařilo se sestavit zajímavé pravidla jiné od těch, které známe ze STULONGu a jsou také uvedeny

v dodatku A. Bohužel k těmto pravidlům nejsou k dispozici data, na kterých bychom je mohli ověřovat.

Hlavní výhodou *background knowledge* je možnost pomocí pravidel snadno (a doufáme, že v budoucnu automaticky) sestavit úlohy, které zajímají experty dané oblasti. Každé pravidlo může být transformováno do úlohy, která se posléze spouští na zkoumaných datech. Výsledek jednoho nebo několika běhu vhodných procedur potom může potvrdit nebo vyvrátit dané pravidlo na konkrétních datech. Zajímavou úlohou je také nalezení výjimek daného pravidla v datech. Vše může být nakonec zahrnuto do výsledné vygenerované analytické zprávy. Všechny výše uvedené možnosti použití pravidel jsou možné pouze za předpokladu úspěšné formalizace pravidel.

3.2. Formalizace background knowledge

Jak již bylo výše zmíněno, *background knowledge* obsahuje různorodá vyjádření vztahů v dané doméně tak, jak je vnímají experti této domény. Závažnost a platnost těchto vztahů se může lišit i v závislosti na doméně: ve fyzice jsou vztahy vyjádřeny exaktně pomocí matematických rovnic fyzikálních zákonů, kdežto například v sociologii znamenají pouze předpokládané chování či smýšlení určité skupiny lidí. Z této heterogenity vyplývá nelehký úkol formalizace pravidel (motivace v předchozím odstavci však jasně ukazuje nutnost formalizace a strojového zpracování pravidel).

3.2.1. Kvalitativní modely

Pro dobývání znalostí pomocí GUHA procedur představuje tato práce první pokus o formalizaci pravidel *background knowledge*. Avšak například u rozhodovacích stromů a klasifikačních úloh existují formalizace pomocí kvalitativních modelů. Tyto modely jsou založené na funkčních vztazích mezi veličinami a podrobněji se o nich dočtete v [1] a [21]. Učení podle těchto modelů funguje tak, že se vybírají pouze rozhodovací pravidla konzistentní s určitým spojením v kvalitativním modelu. Model sestává z uzlů reprezentujících (měřitelné) entity v doméně a spojeními mezi uzly, mající nějaké vlastnosti. Například zápis X (I+) → Y znamená, že derivace Y monotónně funkčně závisí na X. Poté se mohou vybírat jenom rozhodnutí konzistentní s rozhodovacím pravidlem **If (X > k_1) then Y roste**. Kompletní popis metody může čtenář najít v [21].

Prvním přirozeným kandidátem na formalizaci pravidel pro GUHA procedury je použití existující formalizace pomocí kvalitativních modelů. Pro převedení z dobývání znalostí pomocí klasifikačních úloh na GUHA procedury stačí jenom malá modifikace algoritmu: místo pravidel se použijí hypotézy vygenerované jednotlivými procedurami. Avšak při podrobnějším promyšlení narazíme na několik problémů. První problém se týká použitelnosti formalizace na různé domény. Kvalitativní model totiž předpokládá existenci vztahů mezi veličinami, které se dají zapsat do matematických funkcí (například aby šla spočítat derivace). Takové vztahy ve všech doménách nenajdeme. Další velký problém je fakt, že kvalitativní modely jsou založené na měřitelných hodnotách veličin (kardinálních v našem smyslu) a jejich porovnávání. S nominálními hodnotou si model neumí poradit, neboť ji neumí relací větší či menší porovnat s žádnou jinou hodnotou. Posledním problémem je expresívnost, tedy vyjadřovací schopnost modelu. Ta se omezuje jenom na zachycení některých jevů (které vyplývají z omezení kvalitativního modelu) a například ke vztahu *„Jestliže roste krevní tlak, tak*

roste riziko cévní mozkové příhody" uvedenému na začátku této kapitoly není možné pomocí této formalizace přiřadit odpovídající rozhodovací pravidlo[1].

Možná by se tyto problémy daly řešit vhodným rozšířením modelu, v následující části si však představíme jinou formalizaci, „šitou na míru" dobývání znalostí pomocí GUHA procedur, která tyto problémy téměř postrádá.

3.2.2. Atributy, validační literály a abstraktní kvantifikátory

Jedná se o formalizaci pomocí atributů, validačních literálů a abstraktních kvantifikátorů[2], která byla vymyšlena v této práci. Správná tvorba atributu, tedy kategorizace domény, je zásadní pro úspěšnost dalších kroků při konstrukci úlohy i při samotné interpretaci výsledků. Například pro medicínské veličiny jsou definované určité významné hodnoty: vysoký či nízký krevní tlak, hraniční hodnoty *body mass index*. Pro booleovský cedent platí, že jestliže nevytvoříme věcně správně atribut podle těchto hodnot, žádný další krok (konstrukce zadání atomů, literálů či cedentů) úlohu většinou neopraví. U kategoriálního cedentu se dá zadat jenom délka, což také nepomůže opravit chyby vniklé při nesprávné kategorizaci atributu. Z toho vyplývá, že formalizace pravidel *background knowledge* (u dobývání znalostí pomocí GUHA procedur) by měla být založena na atributech. Dodatek B seznamuje čtenáře o podrobnostech tvorby atributů.

Pomocí GUHA procedur ve Ferdovi lze zkoumat buď vztahy k celé doméně (pomocí kategoriálního cedentu) anebo k určité podmnožině domény (pomocí vhodně zadaného atomu a literálu). Dále se dá nastavit znaménko literálu. V hypotézách se však vždy vyskytují buď celé domény v případě kategoriálního cedentu, anebo určité kategorie domény se znaménkem v případě booleovského cedentu.

Začněme tedy od pravidla *background knowledge „Pokud roste úroveň cholesterolu, pak roste i úroveň triglyceridů."* a zkusme vymyslet formalizaci na základě předchozích poznatků o atributech. V pravidle figurují dvě veličiny, úroveň cholesterolu a triglyceridů a existuje mezi nimi nějaký vztah. Můžeme se na ně dívat jako na celou doménu, označíme **chlst** a **trigl**, kde obojí jsou atributy vzniklé z domén. Anebo můžeme použít kategorizaci pro vysokou hladinu cholesterolu i triglyceridů, označme **chlst(HIGH)** a **trigl(HIGH)**. Výběr kategorie atributu (s možností znaménka[3]) bez ohledu na další zapojení budeme nazývat *validační literál*[4]. První případ přirozeně povede na zkoumání pravidla pomocí procedury KL a druhý pomocí procedury 4FT. Zbývá ještě formalizovat část pravidla *„Pokud roste … pak roste…"* K tomu slouží *abstraktní kvantifikátory*.

Abstraktní kvantifikátor si lze představit jako zobecnění určitého kvantifikátoru určité procedury. Kvantifikátory procedur mají své vlastnosti a podle těchto vlastností lze kvantifikátory dělit do tříd kvantifikátorů. Nejlépe je v tomto ohledu prozkoumaná procedura 4FT, kde jsou definované třídy kvantifikátorů a každý kvantifikátor patří do určité třídy. Pro tuto proceduru budou zřejmě *abstraktní kvantifikátory* odpovídat třídám

---

[1] Pro jednoduchost, i protože se budeme podrobněji zabývat jinou vhodnější formalizací, zde neuvádíme důkazy tvrzení o problémech formalizace založené na použití kvalitativních modelů. Po přečtení [31] se čtenář dozví, že některé tvrzení jsou tam uvedeny přímo, jiné bezprostředně vyplývají z definic datových struktur a algoritmu.

[2] Inspirované z velké části Václavem Línem a jeho prací o analytických zprávách[6].

[3] Viz tvorba literálů v dodatku B.

[4] Terminologicky je *validační literál* literál, ve kterém se vyskytuje jenom jedna kategorie. Tedy každý *validační literál* je literál, avšak ne každý literál je *validační literál*. Literály, které nejsou *validační literály*, jsou například literály vytvořené z řezů, či literály z podmnožin větší délky než 1. Literál obsahuje *validační literál*, pokud mají oba stejné znaménko a množina kategorií literálu obsahuje kategorii *validačního literálu*.

kvantifikátorů. U jiných procedur definice tříd zatím nebyly prozkoumány a tak si musíme vystačit s použitím významných kvantifikátorů a jejích hodnot. Jestliže budeme pokračovat ve formalizaci našeho pravidla, můžeme zapsat pravidlo chystané pro proceduru KL jako **chlst** ↑ **trigl**, kde ↑ je vhodný kvantifikátor vyjadřující „rostoucí závislost“. Pro KL se tuto závislost vyjadřuje Kendallův kvantifikátor. Formalizované pravidlo pro proceduru 4FT bude vypadat třeba **chlst(HIGH)** ⇒ **trigl(HIGH)**, kde ⇒ je jeden z implikačních kvantifikátorů[1].

Z výše uvedeného vyplývá, že jak pro 4FT tak pro KL lze definovat „typ“ formalizace slovního pravidla *background knowledge*. Je to dáno tím, že každá procedura se hodí na něco jiného a její výstupy mají jinou sílu. Jestliže analytikovi nabídneme při dobývání znalostí několik těchto typů formalizací, je už na něm, jaký typ se mu bude zdát nejvhodnější (například by mohl postupovat reverzně: „Kterou proceduru bych použil, aby mi vyšlo tohle pravidlo?“).

Pokračováním našich snah o lepší formalizaci slovních pravidel (ať už 4FT nebo KL) je zprostředkování podmínky, se kterou všechny procedury umí zacházet. Analytik si může stanovit, za jaké podmínky chce, aby pravidlo bylo platné. Pokusme se shrnout dosavadní výsledky a vytvořit šablonu na slovní pravidla, ze kterých se dají formalizovat pravidla *background knowledge* pro GUHA procedury:

Za jisté podmínky, vyjádřitelné validačním literálem, jsou spolu dvě veličiny či stavy, vyjádřitelné jedním nebo více atributy či validačními literály, ve vztahu. Tento vztah je definován abstraktním kvantifikátorem.

Šablona zní na první pohled složitě, a mohou se vyskytnout obavy, že jí doménový expert nebude rozumět. Existuje přitom logický požadavek, aby expert mohl zadávat pravidla bez bližší znalosti GUHA procedur. Tento požadavek lze splnit například asistencí datového analytika, který zná možnosti formalizace a bude ze slovních pravidel tvořit pravidla formalizované, nebo vyvinutím vhodných nástrojů, které před expertem skryjí detaily formalizace. Tyto nástroje by se mohli stát součástí Ever-Miner, zejména druhé vrstvy architektury navržené v této práci.

Pro formalizaci byly použity jen dvě ze šesti GUHA procedur implementovaných ve Ferdovi. Prozatím neuvažujeme o rozšíření formalizace na další procedury. Důvodem je malá praktická zkušenost s procedurami a na to navazující neexistence typových úloh. Jestliže nastane po implementaci a kladných zkušenostech s *backgroung knowledge* pro 4FT a KL potřeba rozšířit formalizaci o nové procedury, nic nebrání tomu, aby se tak stalo.

3.3. Ontologie

Dalším typem doménových znalostí jsou ontologie, konkrétně doménové ontologie[2]. Abychom mohli srovnat *background knowledge* a ontologie, musíme nejdříve vysvětlit tento pojem. Ontologie je prostředek, do kterého se snažíme zachytit sdílené znalosti o určité oblasti tak, abychom mohli tyto znalosti (ve formě ontologie) dále rozvíjet, spravovat a používat. Doménové ontologie shromažďují informace o specifických doménách jako je například lékařství, inženýrství či jiné. Čtenář, který nezná pojem ontologie, se o něm může dočíst v [10] nebo [11]. Výchozí literatura pro tuto práci je [9] a českou terminologii pro ontologie jsme převzali z [36]. Pro

[1] Nutno podotknout, že formalizované pravidlo pro proceduru 4FT neodpovídá přesně původnímu slovnímu pravidlu, neboť tvrdí, že jestliže je vysoká jedna strana pravidla, tak potom s určitou pravděpodobností je vysoká i druhá. Protože je procedura 4FT nečastěji používaná a nejlépe prozkoumaná z hlediska kvantifikátorů, je nutné uvést formalizaci i pro ni.

[2] Existují také například ontologie generické, úlohové a aplikační. Velmi pěkně je rozdělení ontologií popsané v [5] v kapitole o ontologiích.

porozumění dalšímu textu je nutné základní porozumění problematice ontologií. Velmi hezký úvod do ontologií nabízí dodatek C, což je část práce [5]. K terminologii je nutné podotknout, že termín atribut je vyhrazený pro dobývání znalostí a pro „vlastnosti tříd" budeme používat termín slot. Bližší vysvětlení ontologií není účelem této práce, tudíž budeme dále předpokládat, že čtenář je s termínem seznámen. S rozšířeným používáním WWW v komerční sféře a vznikem tzv. *sémantického webu* [36] se ontologie těší velkému nárůstu zájmu a jsou v posledních letech jednou z klíčových oblastí výzkumu znalostního inženýrství. V současné době je k dispozici mnoho nástrojů na tvorbu ontologií, existují taktéž nástroje, které ontologie používají. Existuje několik jazyků, ve kterých lze ontologie zapsat a v neposlední řadě existuje množství obsáhlých ontologií popisujících jednotlivé domény.

3.4. Srovnání background knowledge a ontologií

Background knowledge se oproti ontologiím zabývá pouze určitými pravidly mezi entitami a nesnaží se popisovat celou komplexní strukturu domény a vztahy v ní. V ontologii existují nástroje na zachycení pravidel *background knowledge* ať už pomocí relací, či formálních axiomů. Při hlubším promyšlení však zjistíme, že můžou nastat některé případy, kdy není použití ontologie elegantní[1]. Můžeme například definovat relaci *rostouciZavislost* a spojit pomocí ní třídu *hladinaCholesterolu* s třídou *hladinaTrigliceridu*, nebo také instance *hladinaCholesterolu* a *hladinaTrigliceridu* třídy *laboratorní měření*. Problém však může být v tom, že tyto veličiny nevystupují jako samostatné třídy, ale spíše jako sloty jedné nebo různých tříd a ontologie neumožňují definovat relace mezi sloty tříd[2]. Můžeme z nich samozřejmě třídy udělat a v původních třídách změnit typ slotu na nově vytvořenou třídu, ale tímto krokem se vzdalujeme optimálnímu návrhu ontologie.

Mohli bychom také uvažovat o použití formálních axiomů ontologií pro zápis *background knowledge*. Tam ale narazíme na dva hlavní problémy, formální problém s interpretací pravidel a implementační problém. Axiom je pravda, která by měla platit vždy. Naproti tomu pravidlo *background knowledge* je jenom obecná znalost, konsenzus, který platí pouze většinou[3]. Implementační problém tkví ve faktu, že ne všechny ontologické jazyky poskytují plnou podporu pro práci s logickými formulemi[4].

Z předchozích důvodů si myslím, že je lepší oddělit *background knowledge* od ontologie a klasifikovat ji jako samostatnou doménovou znalost (další důvody jsou uvedeny v souvislosti s předchozí literaturou v části 4.2.). Dále chceme poskytnout expertům v oboru jednoduchý systém jak zapisovat pravidla, které následně mohou datoví analytici převést na abstrakce a dále používat. Aby mohl doménový expert zapisovat pravidlo do ontologie, musí nejdříve rozumět tomu co to ontologie je, a poté ještě rozumět konkrétní ontologii, do které pravidlo zapisuje. Ani jeden z těchto úkolů není jednoduchý.

[1] A možná také nesprávné ve smyslu korektního designu ontologií.
[2] Zde velmi závisí na tom, jaký jazyk používáme. V „nových" XML jazycích se používají pouze sloty, takže by s tím neměl být problém.
[3] Například velmi vhodná úloha je hledání výjimek od obecných pravidel v hledaných datech. To jasně ukazuje, že pravidlo vždy neplatí.
[4] Ontologické jazyky se dělí na „odlehčené" a „silné", kde pouze „silné" jazyky podporují plnou práci s logikou prvního řádu, tudíž i s formálními axiomy

4. Práce zabývající se tématem

4.1. Využití ontologií

Myšlenka využití ontologií pro odbývání znalostí pomocí systému LISp-Miner[37] (a tedy pro GUHA procedury) je známá na VŠE již delší dobu. Vznikla diplomová práce, několik článků a prezentací na toto téma – literatura [5], [4] a [35]. V [5] a [4] definují autoři možné využití ontologií v jednotlivých fázích metodologie CRISP-DM[3] a poskytují konkrétnější aplikace na systém LISp-Miner. Například ve [4] navrhují (mimo jiné) identifikaci chybějících atributů pomocí ontologie. Tyto návrhy, jakým způsobem aplikovat ontologie do dobývání znalostí budeme dále nazývat postupy využití ontologií nebo zkráceně PVO. Byly provedeny experimenty na databázi STULONG[8] a jako ontologie byla použita medicínská ontologie UMLS[39] (později modifikovaná pro potřeby STULONG). V [35] se navíc zkoumané PVO aplikují na sociologické data i ontologii.

Všechny uvedené práce identifikují PVO, kde by mohly ontologie zlepšit porozumění dat, zadání úlohy i interpretace výsledků ať už v obecném rámci CRISP-DM metodologie, nebo ve specifickém rámci dobývání znalostí pomocí systému LISp-Miner. Je v nich také uvedeno několik zajímavých myšlenek pro převod pravidel *background knowledge* na ontologii či opačně[1]. Podrobně se těmito PVO bude zabývat kapitola 6. Bohužel všechny kladou důraz zejména na uživatele, který využívá ontologií pro dobývání znalostí a chybí zde návrhy na automatizované (algoritmizovatelné) strojové využití. Nutno podotknout, že v době vzniku prací existoval jen systém LISp-Miner a Ferda[15] byl ve vývoji. Bylo tedy velmi obtížné podporu pro ontologie (či jiné doménové znalosti) integrovat do stávajících modulů systému LISp-Miner[2]. Ačkoli se v této práci nebudou žádné PVO implementovat, 6. kapitola nabídne konkrétní návrhy krabiček pro využití ontologií a tím podstatně rozšíří práce [5], [4] a [35].

4.2. Využití Background knowledge

Používání *background knowledge* ve hledání a potvrzování pravidel je idea, která byla navržena v [26] v souvislosti konceptem Ever-Miner. *Background knowledge* zde má sloužit jako jeden z mnoha vstupů pro Ever-Miner. Jak už bylo zmíněno v předešlé kapitole, první náznak formalizace *background knowledge* byl proveden v [6]. Ačkoli chtěli autoři použít formalizaci pravidel pomocí abstraktních atributů k něčemu jinému (získávání analytických zpráv na základě obsahu) a také byla jejich formalizace poněkud jiná a zjednodušená, myšlenka se ukazuje jako velmi vhodná i pro naše téma.

Background knowledge pravidla nad databází STULONG[8] existovala již za doby tvorby prací [5], [4] a [35]. Zatímco se [5] o pravidlech nezmiňuje, [4] a [35] je už vnímá a zahrnuje do ontologie UMLS. Otázkou zůstává, zda byla pravidla skutečně přidána do ontologie jako relace, neboť ve fázi *Result evaluation* se podle nich dál zhodnocují data a pomocí čtyřpolních tabulek se rozhoduje, jestli data podporují dané pravidlo, či nikoli. Ačkoli byla pravidla na začátku považovaná za součást ontologie,

[1] Například je velmi zajímavé použití *explanation templates*, nebo také ontologické interpretace silných 4ft hypotéz v [35].

[2] V systému LISp-Miner je možné naprogramovat nový modul, který komunikuje s ostatními moduly pomocí sdílené metabáze. Ontologii bychom však nejlépe využili integrací do stávajících modulů, například tvorba dílčích cedentů v modulech *Task. Ferda svým krabičkovým modelem nabízí větší modularitu.

v průběhu procesu dobývání znalostí byla z ontologie „vyjmuta“ a posuzována samostatně. Tedy nemá velký význam zabudovat pravidla jako „obyčejné relace“ do ontologie, když jsou posléze (například ve fázi *Result evaluation*) považované za „nadřazené“ relace, tedy relace kterým je věnována mnohem větší pozornost[1].

Toto je další důvod, proč si myslím, že je potřebné mít *background knowledge* oddělené od ontologie. Lékaři a výzkumníci, kteří plnili STULONG daty vymysleli sadu těchto pravidel, protože se jim pravidla zdála důležitá. Tito lidé požadují od datových horníků analyzujících STULONG, že se nějakým způsobem k těmto pravidlům vyjádří v souvislosti s daty. Jestliže pravidla zahrneme do externí ontologie, může se stát, že ontologie pravidla zcela „absorbuje“ (například tím že je postaví na úroveň „obyčejných“ relací) a lékaři se nedočkají svých odpovědí. Velké ontologie mohou mít stovky až tisíce relací. Je zřejmé, že u těchto ontologií není zajímavé hledat závislosti v datech podle všech relací a jestliže nevíme, které relace byli předtím pravidla *background knowledge*, nemůžeme podle nich zkoumat data.

Tato práce na rozdíl od předchozích chápe *background knowledge* jako samostatný (a velmi perspektivní) typ doménových znalostí a navrhne a vyvine nástroje v prostředí Ferda, které budou pravidla *background knowledge* podporovat.

[1] „Obyčejné relace“ jsou třeba relace *partOf* nebo *locatedIn*. Je zřejmé, že oproti těmto relacím relace vzniklé z pravidel *background knowledge* přitahují pozornost analytiků mnohem více.

5. Background knowledge – implementace validace pravidel

5.1. Důvody pro implementaci background knowledge oproti ontologiím

Součástí diplomové práce má být i implementace nástrojů využívajících výsledků dosažených v práci. Implementace v této práci bude provedena pomocí nových krabiček v prostředí Ferda, které se začlení do stávajícího postupu dobývání znalostí a obohatí ho jistým způsobem o doménové znalosti. Po dohodě s vedoucím práce bylo rozhodnuto, že práce nebude obsahovat pokus o implementaci krabiček používajících ontologie, nýbrž implementaci validaci pravidel *background knowledge*. Níže jsou uvedeny hlavní důvody pro toto rozhodnutí.

- Dostupnost jednotlivých typů doménových znalostí
- Využitelnost implementace *background knowledge* oproti ontologiím
- Obtížnost implementace ontologií

Existuje velký rozdíl mezi dostupností ontologií a *background knowledge*. V současné době je k dispozici jedno i druhé pouze pro databázi STULONG [8][1]. Pro další potencionálně zkoumané domény tyto znalosti nemáme. Je však nepoměrně lehčí vytvořit si sadu pravidel *background knowledge* než vytvořit si ontologii. K vytvoření sady pravidel stačí znát pouze přibližnou formu pravidel a pro zápis může sloužit tužka a papír. K vytvoření ontologie je třeba znát teorii kolem ontologií, dále některé zásady designu ontologií a také softwarový nástroj, který umí ontologie konstruovat. Domníváme se proto, že *background knowledge* má díky své jednoduchosti lepší předpoklady k vytvoření a používání pro různé domény než ontologie. Proto i případný nástroj používající *background knowledge* bude více využíván.

Lepší využitelnost implementace *background knowledge* oproti ontologiím je další důvod, proč upřednostňujeme implementaci *background knowledge*. Každé pravidlo *background knowledge* v sobě nese jisté zadání úlohy na dobývání znalostí. Tato úloha poté umožní ověřit či vyvrátit platnost pravidla na zkoumaných datech. Na druhou stranu využití ontologií v sobě nese jen částečnou pomoc uživateli, jedná většinou o usnadnění nebo zpřehlednění části úlohy, kterou už uživatel musí mít vymyšlenou (kapitola 6. pojednává podrobněji o plánovaných postupech využití ontologií). Proto si myslíme, že je užitečnější implementování validace pravidel *background knowledge*, než některých postupů využití ontologií.

Posledním důvodem pro implementaci validace *background knowledge* je její snazší naprogramovaní oproti programování jakéhokoli použití ontologií. Pro ontologie ve Ferdovi jsou zásadní krabičky *Ontologie* a *Mapování*, tak jak jsou popsány v kapitole 6, jinými slovy načtení struktury ontologie do Ferdy a mapování pojmů ontologie na pojmy datového zdroje. Už první úkol, načtení struktury ontologie, představuje problém. Jestliže například předpokládáme použití jazyka OWL, musíme nejdříve zajistit parsování tohoto jazyka. Pro tento úkol neexistuje komponenta na .NET Framework, což je hlavní vývojové prostředí pro systém Ferda. Musela by se použít některá existující komponenta v Javě. Ferda sice umožňuje pomocí middleware vrstvy ICE [40] mít moduly (taktéž krabičky) zapsané pomocí různých programovacích jazyků na různých platformách, ale nikdo to ještě nezkoušel a lze předpokládat problémy při

[1] Zde existuje sada 36 pravidel *background knowledge*, které popisují základní vědomosti o vztazích mezi doménami. Dále byla vytvořena v rámci práce [5] ontologie pro STULONG na bázi ontologie UMLS.

první implementaci. Dále se musí vytvořit komplikovaný SLICE návrh pro předání struktury ontologie dalším modulům systému Ferda[1]. Více o detailech implementace systému Ferda lze najít v [15].

Naproti tomu implementace validace pravidel *background knowledge*, tak jak je navržena v této kapitole, představuje jednoduchý, ale přesto silný nástroj pro validaci, který využívá pouze stávajících krabiček ve Ferdovi. Implementační problémy ontologií popsané výše se zde nevyskytují. Vyskytuje se tudíž méně problémů, které by mohli úspěšnou implementaci zpozdit, zkomplikovat, či jinak narušit.

Po zvážení výše popsaných argumentů jsme se s vedoucím diplomové práce rozhodli implementovat validaci pravidel *background knowledge* jako implementační část diplomové práce. Tato kapitola popisuje implementaci validace. Kapitola 6 dále představuje návrhy pro implementaci ontologií v prostředí Ferda.

5.2. Validace pravidel background knowledge

Naším úkolem je tedy udělat nástroj, který využívá pravidla *background knowledge* a obohacuje jimi dobývání znalostí pomocí GUHA procedur. Na jedné straně máme pravidla *background knowledge* tak, jak je sestavil například doménový expert. Na druhé straně máme prostředí Ferda a GUHA procedury implementované v něm. Průnikem možností obou je postup, který pomocí krabiček Ferdy a vhodné formalizace pravidel *background knowledge* poskytuje uživateli možnost sestavit si pravidlo pomocí konstrukce, nastavení a zapojení krabiček a validovat toto pravidlo oproti hypotézám vzešlým z jistého běhu GUHA procedury ve Ferdovi. Tomuto postupu budeme říkat **validace pravidel *background knowledge* pomocí GUHA procedur** a o jeho implementaci v prostředí Ferda pojednávají následující části kapitoly 5.

5.3. Implementované krabičky

V části 2.2. a 2.3. jsme se dočetli o Ferdově *krabičkovém modelu* a o GUHA procedurách implementovaných v systému. Dle *krabičkového modelu* je principem validace pravidel *background knowledge* vymyšlení a implementace vhodných krabiček a jejich zapojení do stávající úlohy. Jako základ pro nové krabičky poslouží formalizace pravidel pomocí atributů, validačních literálů a abstraktních kvantifikátorů představená v části 3.2.2.

Základem této formalizace je atribut. Současná implementace Ferdy obsahuje čtyři podobné krabičky, které poskytují funkčnost atributu: dynamické ekvidistanční, ekvifrekvenční a *each value one category* atributy a statický atribut. Více se o nich dozvíte v [15]. Tyto krabičky se používají v zadání úloh a poskytují ostatním krabičkám informace o kategorizaci hodnot sloupce. Není proto třeba pro účely formalizace pravidel *background knowledge* vymýšlet pro práci s atributy krabičky nové.

5.3.1. Validační literál

Dalším pojmem ve formalizaci pravidel *background knowledge* vymyšleným pro tuto práci je *validační literál*. Tento literál se používá při formalizaci pravidel vhodnou pro proceduru 4FT a znamená literál s jednou vyznačenou kategorií a

[1] SLICE je jazyk middleware vrstvy ICE, který definuje rozhraní mezi moduly systému Ferda a který se překládá do jiných jazyků, které ICE podporuje.

znaménkem, s jakým bude vystupovat v hypotézách[1]. Ferda takovou krabičku neobsahuje, a proto musela být implementována. Tato jednoduchá krabička obsahuje jedinou zásuvku *Atribut* a dvě vlastnosti.

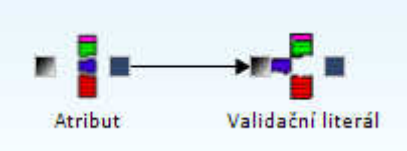
Zásuvky krabičky *Validační literál*:

- Atribut

Vlastnosti krabičky *Validační literál*:

- Jméno kategorie
- Znaménko kategorie

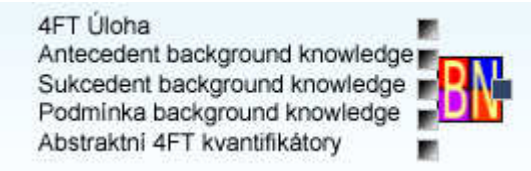
Při práci s krabičkou uživatel nejdříve zapojí do zásuvky *Atribut* libovolný atribut vzniklý při zadání GUHA úlohy. Pro to, aby mohla být krabička použita při pozdější validaci pravidel, musí uživatel dále nastavit obě vlastnosti krabičky. Vlastností *Jméno kategorie* uživatel nastaví vlastní kategorizaci zapojeného atributu, tedy zvolí ze seznamu všech kategorií atributu tu, která je zajímavá. Dále může nastavit pomocí vlastnosti *Znaménko kategorie* znaménko, s jakým bude literál vystupovat v hypotézách. Výchozí znaménko je *pozitivní*, dá se zvolit ještě *negativní* a *obě* znaménka, jestliže na znaménku nezáleží. Obrázek 5. zobrazuje zapojení atributu do krabičky *Validační literál*.



Obrázek 5: Zapojení atributu do validačního literálu

5.3.2. 4FT Background knowledge validátor

Ještě než postoupíme k poslednímu pojmu formalizace, *abstraktnímu kvantifikátoru*, je třeba popsat nejdůležitější krabičku validace pravidel *background knowledge* – krabičku, která validuje hypotézy. Krabička pro validaci pravidel pomocí procedury 4FT se nazývá *4FT Background knowledge validátor*. Zapojením krabiček pro *validační literály* a *abstraktní kvantifikátory* do této krabičky uživatel definuje jednotlivá pravidla. Obrázek 6. zobrazuje krabičku a její zásuvky.



Obrázek 6: Popis zásuvek krabičky *4FT Background knowledge validátor*

Krabička má k dispozici pět zásuvek a následující vlastnosti, akce a moduly pro interakci:

Zásuvky krabičky *4FT Background knowledge validátor*:

- 4FT Úloha
- Antecedent background knowledge
- Sukcedent background knowledge

[1] Zde je důležité zmínit, že literál a zadání literálu jsou rozdílné pojmy. Zatímco zadání literálu znamená „vzorec“ na literály se zadáním koeficientu, typu a znaménka, literál je konkrétní hodnota vzorce, tedy množina kategorií se znaménkem.

- Podmínka background knowledge
- Abstraktní 4FT kvantifikátory

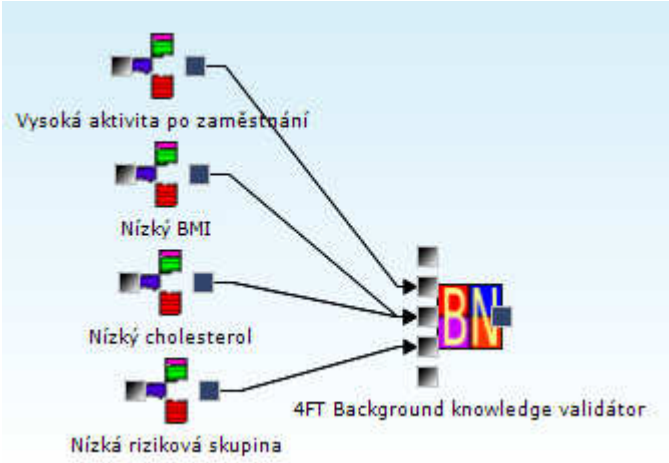
Vlastnosti krabičky *4FT Background knowledge validátor*:

- Stav
- Validované hypotézy

Akce a moduly pro interakci krabičky *4FT Background knowledge validátor*:

- Validovat Background knowledge
- Modul pro prohlížení výsledků

Pro pochopení funkčnosti krabičky je nutné znát význam zapojení jednotlivých zásuvek. Začněme tedy zásuvkou *4FT Úloha*. Do této zásuvky uživatel zapojí jednu krabičku typu *4FT Úloha* poskytující informace o hypotézách vygenerovaných jedním během GUHA procedury 4FT[1]. Pomocí dalších tří zásuvek uživatel tvoří pravidlo *background knowledge* za pomoci validačních literálů (viz část 3.2.2). Do první zásuvky *Antecedent background knowledge* uživatel zapojí jeden nebo více validačních literálů popisující antecedent, do druhé zásuvky *Sukcedent background knowledge* zapojí validační literály popisující sukcedent a konečně do zásuvky *Podmínka background knowledge* zapojí validační literály pro podmínku. Obrázek 7 ukazuje příklad zapojení do těchto zásuvek.



Obrázek 7: Zapojení validačních literálů do krabičky *4FT Background knowledge validátor*

Pravidlo z obrázku 7. se vztahuje k datovému zdroji STULONG a jeho přepis do slovní podoby může být například: *„Pro pacienty v nízké rizikové skupině je ve vztahu vysoká aktivita po zaměstnání s nízkým BMI a nízkým cholesterolem.*[2]*“*

Validační literál odpovídající vysoké aktivitě po zaměstnání je zapojen do zásuvky *Antecedent background knowledge*, validační literály odpovídající nízkému BMI a nízkému cholesterolu jsou zapojeny do zásuvky *Sukcedent background knowledge* a konečně nízká riziková skupina je zapojená do podmínkové zásuvky.

[1] Bylo by také možné zapojit do zásuvky více 4FT úloh. Pro vstup do algoritmu validace by potom místo množiny hypotéz z jedné úlohy vstupovalo sjednocení množin hypotéz z více úloh. Praktické využití tohoto zlepšení je však sporné, neboť jedno pravidlo *background knowledge* logicky vede na konstrukci jedné úlohy, jejíž hypotézy se posléze budou validovat.

[2] Pozorný čtenář si všimne, že sada pravidel *background knowledge* u datového zdroje STULONG pravidlo neobsahuje. Pravidlo vzniklo spojením dvou existujících pravidel s přidáním podmínky, neboť chceme v příkladu poukázat na možnost použití podmínky a zapojení dvou validačních literálů do stejné zásuvky.

Když si uživatel správně zkonstruuje pravidlo pomocí zapojení validačních literálů do krabičky, může potom pomocí akce *Validovat Background knowledge* spustit algoritmus validace. O jeho průběhu informují vlastnosti krabičky: vlastnost *Stav* informuje o stavu validace a může nabývat hodnot *Nezahájeno*, *Přerušeno* a *Dokončeno*. Vlastnost *Validované hypotézy* udává počet validovaných hypotéz 4FT úlohy. Výstupem krabičky je množina validovaných hypotéz – podmnožina vstupních hypotéz ze zásuvky *4FT Úloha*. Protože krabička poskytuje stejný výstup jako krabička *4FT Úloha*, lze prohlížet výsledné hypotézy pomocí modulu pro interakci *Modul pro prohlížení výsledků*, který už je ve Ferdovi implementovaný.

5.3.3. 4FT abstraktní kvantifikátory

Do poslední zásuvky krabičky *4FT Background Knowledge validátor* se zapojují abstraktní atributy pro proceduru 4FT. Je to také poslední část formalizace pravidel (pro proceduru 4FT), o které ještě nebyla zmínka. V části 3.2.2. se mluví o abstraktním kvantifikátoru jako o zobecnění určitého kvantifikátoru určité procedury. Vzhledem k faktu, že pro proceduru 4FT jsou definované třídy kvantifikátorů, přirozenými kandidáty pro abstraktní kvantifikátory pro 4FT jsou právě tyto třídy. Tabulka 1 obsahuje přehled tříd kvantifikátorů tak, jak je sestavil vedoucí práce.

Pro každou třídu kvantifikátorů byla sestrojena jedna krabička, která ji reprezentuje. Každá krabička obsahuje skryté vlastnosti pro nastavení obvyklých hodnot parametrů. Tyto vlastnosti může uživatel nastavit v konfiguračních XML souborech krabičky. Původně se uvažovalo o úplné parametrizovatelnosti krabiček abstraktních kvantifikátorů, tj. že by uživatel mohl přesouvat i kvantifikátory z jedné třídy do druhé. Protože však určení náležení kvantifikátoru do třídy je záležitostí matematického důkazu, nepředpokládá se pohyb kvantifikátorů mezi třídami[1]. Dále potom kvantifikátory nemusí mít stejné parametry a tudíž je úplná parametrizovatelnost pro uživatele značně nepřehledná.

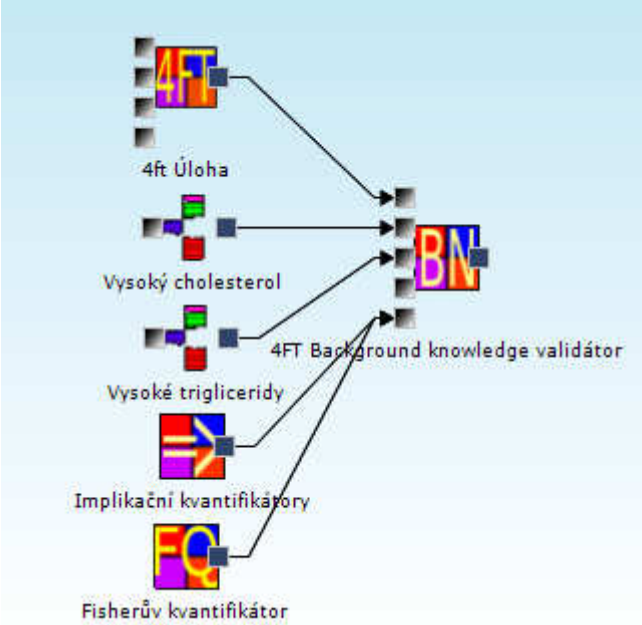
Obrázek 8. ukazuje konečné zapojení krabičky *4FT Background knowledge validátor*. Jedná se o validaci pravidla „*Pokud roste úroveň cholesterolu, pak roste i úroveň triglyceridů.*“ z části 3.2.2. za pomoci procedury 4FT (tedy použitím validačních literálů). Oproti obrázku 7. je zde zapojená 4FT úloha a do zásuvky *Abstraktní 4FT kvantifikátory* jsou zapojeny dva kvantifikátory. První krabička *Implikační kvantifikátory* zastupuje třídu Implikačních kvantifikátorů definovanou v tabulce 1. Dále je do zásuvky zapojen „obyčejný“ 4FT *Fischerův kvantifikátor*. Možnost zapojit do zásuvky pro abstraktní kvantifikátory i „obyčejné“ kvantifikátory byla ponechána pro zkušené uživatele znající přesně význam 4FT kvantifikátorů. V tomto případě uživatele zajímá, jestli hypotézy z krabičky *4FT Úloha* jsou validní vzhledem k implikačním kvantifikátorům a navíc splňující Fischerův kvantifikátor. Část 5.4. přesně popisuje algoritmus validace a úlohu abstraktních i „obyčejných“ kvantifikátorů v ní.

[1] Mohlo by se to stát pouze v případě nových kvantifikátorů, nebo přesunem kvantifikátoru ze třídy *Ostatní* do nějaké jiné.

Implikační kvantifikátory		
symbol	**název**	**obvyklé hodnoty parametrů**
$\Rightarrow_{p, Base}$	fundovaná implikace	P = 0.95, Base = 5% z počtu řádků matice
$\Rightarrow^{!}_{p, \alpha, Base}$	dolní kritická implikace	P = 0.95, α = 0.05, Base = 5% z počtu řádků matice
$\Rightarrow^{?}_{p, \alpha, Base}$	horní kritická implikace	P = 0.95, α = 0.05, Base = 5% z počtu řádků matice
$\sum$ - dvojitě implikační kvantifikátory		
$\Leftrightarrow_{p, Base}$	dvojitá fundovaná implikace	P = 0.95, Base = 5% z počtu řádků matice
$\Leftrightarrow^{!}_{p, \alpha, Base}$	dvojitá dolní kritická implikace	P = 0.95, α = 0.05, Base = 5% z počtu řádků matice
$\Leftrightarrow^{?}_{p, \alpha, Base}$	dvojitá horní kritická implikace	P = 0.95, α = 0.05, Base = 5% z počtu řádků matice
$\sum$ - ekvivalenční kvantifikátory		
$\equiv_{p, Base}$	fundovaná ekvivalence	P = 0.95, α = 0.05, Base = 20% z počtu řádků matice
$\equiv^{!}_{p, \alpha, Base}$	dolní kritická ekvivalence	P = 0.95, α = 0.05, Base = 20% z počtu řádků matice
$\equiv^{?}_{p, \alpha, Base}$	horní kritická ekvivalence	P = 0.95, α = 0.05, Base = 20% z počtu řádků matice
Fisherovské kvantifikátory		
$\sim_{\delta, Base}$	jednoduché vychýlení	δ = 0, Base = 5% z počtu řádků matice
$\approx_{\alpha, Base}$	Fischerův kvantifikátor	α = 0.05, Base = 5% z počtu řádků matice
$\approx^{\chi}_{\alpha, Base}$	χ^2 kvantifikátor	α = 0.05, Base = 5% z počtu řádků matice
Ostatní kvantifikátory		
$\approx^{+}_{p, Base}$	Above average	P = 0.2, Base = 5% z počtu řádků matice
$\approx^{-}_{\alpha, Base}$	Below average	P = 0.2, Base = 5% z počtu řádků matice
$\approx^{E}_{\delta, Base}$	E-kvantifikátor	δ = 0.05, Base = 5% z počtu řádků matice

Tabulka 1: Třídy 4FT kvantifikátorů[1]

[1] Obvyklé hodnoty parametrů závisí na situaci, kdy je kvantifikátor použit. Zde uvedené hodnoty odpovídají prvnímu spuštění procedury 4FT, když o datech uživatel nic neví. V jiných situacích se můžou hodnoty značně lišit.
Above average patří (velmi pravděpodobně) také mezi Fischerovské kvantifikátory. Pro účely této práce není případná definice třídy Fischerovských kvantifikátorů ani důkaz tohoto tvrzení podstatný.



Obrázek 8: Konečné zapojení krabičky *4FT Background knowledge validátor*

5.3.4. KL Background knowledge validátor

Kromě validace pomocí procedury 4FT může uživatel použít validaci pravidel pomocí procedury KL. Validace je založena na zkoumání závislosti dvou atributů vyjádřené pomocí abstraktního kvantifikátoru pro KL. K tomuto účelu slouží krabička *KL Background knowledge validátor*. Krabička je na první pohled podobná krabičce pro validaci 4FT pravidel a má následující zásuvky, vlastnosti, akce a moduly pro interakci:

Zásuvky krabičky *KL Background knowledge validátor*:

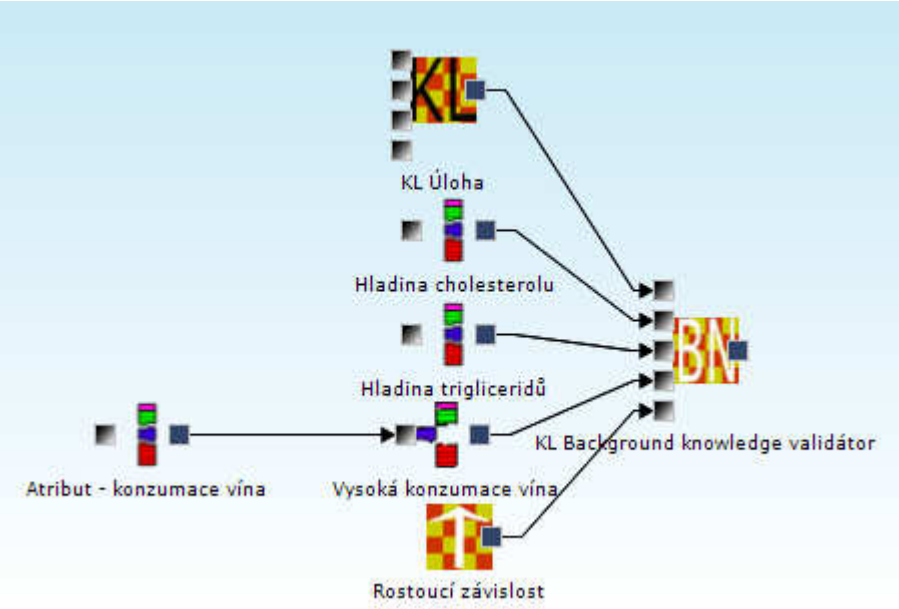
- KL Úloha
- Řádkové atributy
- Sloupcové atributy
- Podmínka background knowledge
- Abstraktní KL kvantifikátory

Vlastnosti krabičky *KL Background knowledge validátor*:

- Stav
- Validované hypotézy

Akce a moduly pro interakci krabičky *KL Background knowledge validátor*:

- Validovat Background knowledge
- Modul pro prohlížení výsledků



Obrázek 9: Použití krabičky *KL Background knowledge validátor*

Pravidla *background knowledge* se konstruují stejně jako u 4FT zapojením vhodných krabiček do zásuvek krabičky *KL Background knowledge validátor*. Na obrázku 9. si vysvětlíme toto zapojení. Opět použijeme pravidlo *„Pokud roste úroveň cholesterolu, pak roste i úroveň triglyceridů.“*, s tím že si dále stanovíme podmínku vymezující pacienty pouze s vysokou konzumací vína[1]. Slovní pravidlo by mohlo znít například: *„Pro pacienty s vysokou konzumací vína: pokud roste úroveň cholesterolu, pak roste i úroveň triglyceridů.“*

Uživatel nejdříve zapojí krabičku typu *KL Úloha*, poskytující KL hypotézy, do stejnojmenné zásuvky. Poté musí definovat pravidlo *background knowledge*. Má k dispozici dvě zásuvky, *Řádkové atributy* a *Sloupcové atributy* pro atributy, které mají být spolu ve vztahu. Na obrázku 9. zapojil uživatel do těchto zásuvek atributy vyjadřující hladinu cholesterolu a triglyceridů[2]. Dále má uživatel k dispozici podmínku vyjádřenou validačním literálem (podobně jako u krabičky pro validaci pomocí 4FT). V našem příkladu si uživatel vytvořil atribut konzumace vína a zajímaly ho vysoké hodnoty. Posledním krokem ke správnému zapojení krabičky *KL Background knowledge validátor* je zapojení abstraktního kvantifikátoru. O těchto kvantifikátorech pojednává následující část práce, o algoritmu validace pravidel pomocí procedury KL pak část 5.4.

5.3.5. KL abstraktní kvantifikátory

Ačkoli je pro proceduru KL definováno pět agregačních a šest funkčních kvantifikátorů, nejsou jejich vlastnosti ani použití zdaleka tak dobře prozkoumány jako u procedury 4FT[3]. Důvodem je zřejmě to, že procedura KL je méně pochopitelná pro začátečníka[4] a také se méně používá například při výuce nebo výzkumu. Prozatím nejsou definované žádné třídy kvantifikátorů, které bychom mohli použít pro abstraktní kvantifikátory.

---

[1] Definice podmínky je zde opět zejména pro to, aby se ukázaly všechny možnosti zapojení krabičky.
[2] Pro přesnost je třeba uvést, že oproti validaci pomocí KL může uživatel do zásuvek *Řádkové* a *sloupcové atributy* zapojit jenom jeden atribut. Zapojení více atributů by nemělo pro pravidlo smysl. Podrobnější vysvětlení je v části 5.4.2.
[3] Kompletní popis KL kvantifikátorů se nachází v [20] a [28].
[4] Kromě pochopení toho, co samotná procedura dělá, je problém pochopit jednotlivé kvantifikátory, které často vyžadují hlubší znalost statistiky.

Protože teorie KL kvantifikátorů není zatím prozkoumaná, musíme se dívat po dalších možnostech zadání abstraktních kvantifikátorů. Nabízí se zvolit přístup „co uživatele zajímá“, tedy vymyslet takové kvantifikátory, na které se uživatelé nejvíce ptají. Jestliže si prohlédneme sadu pravidel pro STULONG v dodatku A, zjistíme, že všechny závislosti mezi atributy v pravidlech by se dali shrnout do vět *„jestliže roste A roste B“* a *„jestliže roste A klesá B“*. Mohli bychom tedy sestrojit abstraktní kvantifikátory pro tyto případy s tím, že jejich použití by bylo široké.

Vytvoříme tedy abstraktní kvantifikátory *Rostoucí závislost* a *Klesající závislost*, které budou ověřovat tyto pravidla. Jak už bylo řečeno v části 3.2.2., použijeme k tomu Kendallův kvantifikátor pro KL[1]. Kvantifikátor počítá *pozitivní* či *negativní ordinální závislost* atributů. Tuto závislost definuje parametr τ_b, který může nabývat hodnot z intervalu <-1, 1>. Hodnoty τ_b menší nule znamenají *negativní ordinální závislost*, tedy jestliže roste řádkový atribut, tak potom klesá sloupcový. Hodnoty τ_b větší než nula znamenají *pozitivní ordinální závislost*, tedy jestliže roste sloupcový atribut, tak roste také řádkový. Jestliže $|\tau_b| = 1$, potom sloupcový atribut je funkcí řádkového.

Největším problémem u těchto abstraktních kvantifikátorů zůstává určení významné hranice τ_b, tedy hodnoty, od které už je závislost mezi atributy významná a není to jenom výsledek náhodného rozložení hodnot v kontingenční tabulce. Prozatím byla tato hodnota stanovena na 0,7 a pozdější pokusy na datech ukážou, jestli je tato hodnota relevantní.

U validace pomocí procedury 4FT se dal každý „obyčejný“ kvantifikátor zapojit jako abstraktní kvantifikátor. Protože bylo implementovaných celkem pět různých abstraktních kvantifikátorů, předpokládá se, že tyto abstraktní kvantifikátory budou uživateli stačit a že si jenom občas propůjčí „obyčejný“ kvantifikátor pro svoji validaci. U procedury KL lze také každý „obyčejný“ kvantifikátor zapojit jako abstraktní. Protože však existují pouze dva abstraktní kvantifikátory, může být použití „obyčejných“ kvantifikátorů vítanou volbou pro zlepšení vyjádření vztahu mezi atributy.

5.4. Algoritmy pro validaci pravidel

V následujících odstavcích budou představeny algoritmy pro validaci pravidel pomocí procedur 4FT a KL. Tyto algoritmy nejsou nijak zvlášť těžké či převratné. Obsahují však několik důležitých kroků, které je potřeba uvést pro lepší pochopení problematiky validace pomocí GUHA procedur. Dále se mezi validací pomocí 4FT a KL vyskytují některé na první pohled ne příliš patrné rozdíly, které je také třeba zmínit.

```
IList<HypothesisStruct> ValidateBackgroundKnowledge(IList<HypothesisStruct> hypotheses)
{
      hypotheses = ValidateAttributes(hypotheses);
      hypotheses = RestrictToNormalQuantifiers(hypotheses);
      hypotheses = RestrictToAbstractQuantifiers(hypotheses);

      return hypotheses;
}
```

Kód 1: Kostra algoritmu validace

Kód 1 představuje kostru algoritmu validace (pro přehlednost velmi zjednodušenou). Do funkce vstupují hypotézy z úlohy, které se mají validovat. Validace probíhá ve třech krocích – validace vůči atributům, validace vůči „obyčejným“ kvantifikátorům a validace vůči abstraktním kvantifikátorům. Tyto dílčí validace jsou

[1] Kendallův kvantifikátor tak, jak je použitý v této práci, neodpovídá úplně přesně kvantifikátoru definovanému v [20]. V tomto dokumentu je se počítá pouze s absolutní hodnotou τ_b a znaménko závislosti je nezajímavé. Z poznámek v dokumentu je však zřejmé, že autoři původně se znaménkem počítali a neimplementovali ho zřejmě z důvodu nízké priority.

na sobě nezávislé a mají za úkol vyřadit ty hypotézy ze seznamu, které nesplňují jisté kritérium. Pořadí validací je dané předpokládanou náročností a „vyřazovací schopností“ jednotlivých validací.

První validace vůči atributům[1] má největší možnost vyřadit hypotézy ze seznamu a přitom není časově náročná, neboť v ní probíhají pouze porovnávání textových řetězců (bude popsáno dále). Například výstupem 4FT procedury jsou hypotézy, které obsahují v cedentech nejrůznější literály, tedy i kategorie z atributu. Pravidlo *background knowledge* obsahuje pouze několik validačních literálů, tudíž všechny hypotézy, které tyto kategorie neobsahují, se musí vyřadit.

Druhá probíhá validace vůči „obyčejným“ kvantifikátorům. Tato validace bude taktéž přesněji popsána níže. Třetí, časově zdaleka nejnáročnější, je validace vůči abstraktním kvantifikátorům. Při ní se vždy počítají hodnoty jednoho nebo i více kvantifikátorů což může být častokrát výpočet složitých statistických vzorců. Proto je pro optimalizaci výhodné mít tuto validaci až poslední, kdy do ní vstupuje co nejmenší množství hypotéz.

5.4.1. Validace vůči kvantifikátorům

Začněme popisem validace vůči abstraktním kvantifikátorům, která je zjednodušeně popsaná v kódu 2. Tato validace je pro 4FT i KL stejná.

```
IList<HypothesisStruct>            RestrictToAbstractQuantifiers(IList<HypothesisStruct>
hypotheses)
{
      bool validInAllQuantifiers;
      List<HypothesisStruct> result = new List<HypothesisStruct>();

      foreach (HypothesisStruct hypothesis in hypotheses)
      {
            validInAllQuantifiers = true;

            foreach (AbstractQuantifier quant in AbstractQuantifiers)
            {
                  if (!quant.Validity(hypothesis))
                  {
                        validInAllQuantifiers = false;
                        break;
                  }
            }

            if (validInAllQuantifiers)
            {
               result.Add(hypothesis);
            }
      }

      return result;
}
```

Kód 2: Validace vůči abstraktním kvantifikátorům

Kód není třeba dále rozebírat, snad jen poznámku ke funkci *bool Validity(HypothesisStruct)*, kterou každý abstraktní kvantifikátor poskytuje. Pro abstraktní kvantifikátory pro KL se vypočítá vždy jenom hodnota parametru (τ_b) a poté se porovná s hraniční hodnotou definovanou v konfiguračních souborech. U abstraktního kvantifikátoru pro 4FT se počítají vždy hodnoty pro všechny kvantifikátory v dané třídě, a jestliže hypotéza splní alespoň jeden z nich, vrátí funkce *true*.

[1] Pro validaci pomocí procedury 4FT by se mělo přesněji říkat validace vůči literálům, neboť se používají validační literály. Protože literály vycházejí z atributů a z důvodu jednoduchosti budeme první fázi validace nazývat nadále validace vůči atributům.

Pro „obyčejné“ kvantifikátory procedura vypadá téměř shodně s rozdílem některých datových typů. Proto ji zde nebudeme uvádět.

5.4.2. Validace vůči atributům

Tato část validace je různá pro procedury 4FT a KL. Vyplývá to z principu fungování těchto procedur. Při zapojení více dílčích (booleovských) cedentů do procedury 4FT se tyto cedenty v hypotéze skládají a mohou vystupovat současně. Jestliže naopak zapojíme vícero kategoriálních cedentů do procedury KL, cedenty se do hypotézy neskládají a pouze jeden z nich se stane řádkovým či sloupcovým atributem, nad kterým se potom počítá kontingenční tabulka. Kódy 3 a 4 zobrazují procedury pro validaci vůči atributům pro procedury 4FT a KL.

```
IList<HypothesisStruct> ValidateAttributes(IList<HypothesisStruct> hypotheses)
{
      //používá se struktura CategoryNameAndSign, která obsahuje jméno kategorie a její
      //znaménko, máme naplněné proměnné (ze zásuvek)
      IList<CategoryNameAndSign> antecedents;
      IList<CategoryNameAndSign> succedents;
      IList<CategoryNameAndSign> conditions;

      List<HypothesisStruct> result = new List<HypothesisStruct>();
      bool containsAllAttributes;

      foreach (HypothesisStruct hypothesis in hypotheses)
      {
            containsAllAttributes = true;

            foreach(CategoryNameAndSign ant in antecedents)
            {
                  foreach (BooleanLiteralStruct literal in hypothesis.booleanLiterals)
                  {
                        if (LiteralIsAntecedent(literal))
                        {
                                containsAllAttributes = CheckCategorySigns(...);
                                containsAllAttributes =
                                       CheckIfLiteralContainsCategory(ant.name);
                        }
                  }
            }

            //To samé se děje pro sukcedent a podmínku

            if (containsAllAttributes)
            {
                  result.Add(hypothesis);
            }
      }

      return result;
}
```

Kód 3: Validace atributů pro proceduru 4FT

Kód 3 je na první pohled složitější než kódy před ním. Princip je stejný jako u validace kvantifikátorů: iteruje se podle hypotéz a hypotézy splňující podmínky ve funkci jsou přidávány do výsledku. Souhrnně lze říct, že pro každou hypotézu pro každý kategorizovaný atribut z antecedentu, sukcedentu i podmínky se musí v hypotéze vyskytovat (booleovský) literál, který se shoduje znaménkem a obsahuje kategorii vyjádřenou validačním literálem. Protože je tato věta značně nepřehledná, dáváme k dispozici i jasnější vyjádření pseudokódem.

```
IList<HypothesisStruct> ValidateAttributes(IList<HypothesisStruct> hypotheses)

{
      //ze zásuvek krabičky máme k dispozici jména sloupcových a řádkových atributů, tak
      //jak vystupují v literálu
      string rowAttributeNameInLiterals;
      string columnAttributeNameInLiterals;

      List<HypothesisStruct> result = new List<HypothesisStruct>();
      bool containsEverything;

      foreach (HypothesisStruct hypothesis in hypotheses)
      {
            containsEverything = true;

            if (hypothesis.literals[0].literalName != rowAttributeNameInLiterals)
            {
                  containsEverything = false;
            }

            if (hypothesis.literals[1].literalName != columnAttributeNameInLiterals)
            {
                  containsEverything = false;
            }

            if (!ContainsCondition(…))
            {
                  containsEverything = false;
            }

            if (containsEverything)
            {
                  result.Add(hypothesis);
            }
      }

      return result;
}
```

Kód 4: Validace atributů pro proceduru KL

Kód 4 zobrazuje validaci atributů pro proceduru KL. Pro stručnost v kódu předpokládáme, že řádkový atribut bude vždy v seznamu literálů na nultém místě a sloupcový na místě prvním. Dále není detailně rozepsaná procedura *ContainsConditions*, která podobně jako u validace pro 4FT zkouší přítomnost názvů a znamének všech validačních literálů v podmínce hypotézy. Je vidět, že validace atributů pro KL je jednodušší, neboť stačí, když se shodují jména literálů atributů[1] .

Můžeme si všimnout, že při všech porovnáváních literálů se porovnávají pouze jejich jména. Mohl by tedy hypoteticky nastat případ, kdy by se například nesprávně vyhodnotilo porovnání dvou literálů z různých atributů, které se shodují ve jméně[2]. V tomto případě by algoritmus chybně validoval pravidlo. Tento problém jsem se však rozhodl neřešit. Lze totiž předpokládat, že k němu nebude docházet. Můžeme počítat s přirozenou inteligencí uživatele, který chce něco vyzkoumat místo toho, aby se pokoušel rozbíjet si pracně sestavenou úlohu na validaci. Při plánovaném programovém sestavování celých řetízků (např. systémem Ever-Miner) k problému za předpokladu korektního algoritmu docházet nebude. Při řešení problému (pomocí zpětného dohledávání zapojených krabiček) by mohlo dojít k dalším problémům, jejichž vysvětlení je pro tuto práci nepodstatné.

5.5. Užitečnost a použitelnost validace pravidel

V předchozích odstavcích byl představen a podrobně popsán nástroj na validaci pravidel *background knowledge* v prostředí Ferda. Byla provedena implementace

[1] Je to způsobeno odlišným principem fungování procedur, což je vysvětleno na začátku části 5.4.2.
[2] Vzhledem k tomu, že všechny krabičky pro atributy mají vlastnost *Jméno v literálech*, může se tento případ stát celkem snadno.

navrhnutých krabiček a první testování funkčnosti krabiček i interakce se stávajícími řetízky krabiček systému Ferda. Po testování lze konstatovat, že nástroj je funkční. Zároveň si musíme položit otázku, zda je nástroj (kromě funkčnosti) také užitečný a vhodný pro širší použití. Tato část práce se snaží nalézt odpovědi na položené otázky. Jsou zde také uvedeny některé možnosti dalšího využití nástroje, zejména v souvislosti s projektem Ever-Miner.

5.3.1. „Není to jen další filtr?“ aneb užitečnost nástroje

Kritický čtenář by mohl namítnout, že nástroj na validaci pravidel *background knowledge* je jenom další filtr nad hypotézami. Tento čtenář má pravdu, neboť striktně řečeno, nástroj vezme množinu hypotéz na vstupu a vydá na výstupu její podmnožinu, tedy hypotézy skutečně filtruje. Myslím si však, že nástroj představuje zcela nový druh filtru, co se týče způsobu i užitečnosti filtrování.

V současné době existuje v systému Ferda třídění podle různých statistik a podle hodnot aplikovaných kvantifikátorů. Existuje také filtrování podle literálů[1]. Avšak toto filtrování je dost omezené, neboť se zabývá pouze výskytem literálů v hypotéze. Například jestliže se sukcedent 4FT úlohy skládá ze dvou literálů, tak potom můžeme pomocí filtru zobrazit jenom ty hypotézy, ve kterých se vyskytuje jeden z literálů. Jestliže chceme zkoumat pouze tento literál, není důvod zahrnovat druhý literál do sukcedentu.

Validace pravidel *background knowledge* jde o krok dál, tedy nezabývá se jenom o výskyt literálu v hypotéze, nýbrž o výskyt atomů, potažmo kategorií atributu. Zvláště významné je použití validace v případě procedury 4FT. Při komplikovanějších zadáních 4FT úloh často vychází velké množství dlouhých hypotéz. Pro zmenšení počtu hypotéz se postupuje většinou tak, že zadání cedentů ponechá a zpřísní se hodnoty kvantifikátorů. Při dalším běhu se sice zmenší počet hypotéz, ale může se stát, že zajímavé ale slabší hypotézy novým nastavením kvantifikátoru neprojdou. Nástroj tedy umožňuje co nejpřesněji popsat hypotézy, které uživatele zajímají. V tomto ohledu nástroj významně posouvá stávající možnosti dobývání znalostí pomocí procedury 4FT.

V systému LISp-Miner sice existovaly filtry na hypotézy, avšak možnosti filtrování byly skryty v nepřehledném dialogu a uživatel často ani nevěděl, že filtrovat může. V modulu *Modul pro prohlížení výsledků* systému Ferda je situace poněkud lepší, ale stále jediným výstupem filtru je vizuální reprezentace hypotéz, která už se nedá dále použít. Dalším přínosem validace pravidel *background knowledge* je krabičková implementace, tedy „vytažení“ filtru do krabičky. Uživatel si je potom vědom toho, že filtr existuje a může s ním pracovat jako s krabičkou, tedy může používat výsledky validace jako vstup pro další krabičky[2].

Poslední agrument pro užitečnost validace (jak pro 4FT tak pro KL) je existence abstraktních kvantifikátorů. Zatímco před implementací validace uživatel musel znát význam kvantifikátorů a jejich parametrů při zadávání úlohy, nyní už může jenom zapojit do krabičky validace určitý abstraktní kvantifikátor a nemusí se zajímat o jeho nastavení. Abstraktní kvantifikátor tedy „odstíní“ od uživatele teorii kvantifikátorů.

Na druhou stranu je třeba přiznat, že pro validaci pravidel *background knowledge* pomocí procedury KL nástroj nabízí jen trochu více než dva nové kvantifikátory: přesné zadání podmínky pomocí validačního literálu. U procedury KL

[1] Ve starším systému LISp-Miner také existuje filtrování podle délky cedentu.
[2] Můžeme například řetězit několik validačních krabiček za sebou a tedy filtrovat několikrát podle různých kritérií.

nástroj není tak silný, protože KL většinou negeneruje tak velké množství hypotéz a hypotézy jsou snadněji čitelné[1].

5.3.2. Použitelnost nástroje

Z výše uvedeného vyplývá, že implementovaný nástroj může být užitečný, zejména pro proceduru 4FT. Otázkou zůstává, jestli je nástroj vhodný pro širší použití. S nástrojem zatím nejsou žádné obsáhlejší zkušenosti při použití, které by odhalily případné chyby návrhu krabiček či nevhodné uživatelské postupy. Proto jsme se rozhodli vypracovat rozsáhlejší experimenty, které poskytnou první praktické zkušenosti s nástrojem. Jako úkol byla vybrána verifikace pravidel datového zdroje STULONG. S průběhem a výsledky experimentů seznamuje 7. kapitola.

5.3.3. Souvislost s projektem Ever-Miner a možnosti dalšího rozvoje

Nástroj je implementovaný pomocí krabiček ve Ferdovi, tedy na nulté vrstvě plánovaného systému Ever-Miner (viz část 2.4.1). Při jeho používání uživatel ještě stále musí znát teorii GUHA procedur a způsob sestavování úloh v prostředí Ferda. Je tedy určen pro datové analytiky spíše než pro doménové experty. Optimální způsob použití je spolupráce mezi oběma, tedy doménový expert poskytne datovému analytikovi data a zadá sadu pravidel *background knowledge*, která se má zkoumat. Analytik sestrojí pomocí nástroje úlohy odpovídající těmto pravidlům, validuje pravidla a výsledky předá zpět expertovi.

Nástroj je však důležitý i pro systém Ever-Miner. Poskytuje vrstvu 0 pro řešení typových úloh jako *„Potvrzení platných vztahů v datech“* nebo *„Hledání výjimek v datech“*. Pokud bude zkonstruovaná vrstva 1 systému Ever-Miner (viz část 2.4.1), mohou krabičky sloužit jako základ druhé vrstvy systému pro řešení těchto typových úloh. Výsledkem by mohl být systém, který (už bez datového analytika) řeší doménovému expertovi pomocí přívětivého uživatelského prostředí zajímavé úlohy.

Prohloubení znalostí a zkušeností s validací pravidel *background knowledge* představuje další směr možného zkoumání. Je nutno prozkoumat používání abstraktních kvantifikátorů a navrhnout další. Také se nabízí možnost zlepšit formalizaci pravidel[2]. V neposlední řadě můžeme vypracovat sady příkladů pravidel a postupů validace, které budou sloužit pro lepší pochopení nástroje.

[1] Velkého množství verifikací a hypotéz u KL se lze dočkat jenom při použití komplikované podmínky.
[2] Ve stávajícím modelu má uživatel například pouze možnost výskytu všech kategorizovaných atributů, které jsou zapojeny do jedné zásuvky, současně. Postačovat by mohl výskyt pouze jednoho z kategorizovaných atributů. Toto zlepšení by příliš nezkomplikovalo formalizaci, která by se stala silnější co do vyjadřovací schopnosti.

6. Využití ontologií při aplikacích GUHA procedur

Jak jsme se dozvěděli ve 4 kapitole, v minulosti byly navrženy postupy jak využít ontologie pro dobývání znalostí pomocí procedur GUHA (budeme je nazývat postupy využití ontologií, aneb PVO). Bohužel tyto PVO byly vymyšleny zejména pro ruční použití a o možné pomoci počítače se spíše neuvažovalo[1]. Nové prostředí Ferda se svým *krabičkovým modelem* umožňuje velmi snadno včlenit tyto PVO do procesu zadávání úlohy. Proto můžeme na rozdíl od předchozích prací uvažovat o strojové pomoci uživateli. Tato kapitola podrobně rozebere všechny PVO, které přicházejí v úvahu. Jsou to nejenom PVO z literatury [5], [4] a [35], ale i některé nové nápady autora a vedoucího diplomové práce. Všechny tyto PVO jsou podrobně rozpracovány, u většiny je sestrojen krabičkový návrh v prostředí Ferda a diskutovány možnosti dalšího rozvoje PVO. Práce tímto poskytuje značné rozšíření dosavadních znalostí o využití ontologií ve směru použití v aplikacích. V rámci práce se nebude žádný z PVO implementován, nebo pro implementační část byla vybrána validace pravidel *background knowledge* (důvody jsou uvedeny v části 5.1). Kapitola se snaží připravit půdu pro další vývoj v tomto směru tím, že z existujících PVO vybere pro budoucí implementaci nejvhodnější. „Vhodnost“ PVO bude posuzována zejména podle tří kritérií:

- **Obecnost**
- **Implementovatelnost**
- **Ulehčení práce analytika**

Obecnost říká, že jestliže daný PVO implementujeme, tak bude obecný a bude fungovat nezávisle na vstupních datech. Například jestliže chceme pracovat s ontologiemi, řešení postupu bude muset umět pracovat se všemi ontologiemi daného jazyka (jestliže vstup bude jakákoli ontologie).

Implementovatelnost znamená, že lze v rozumném čase a rozsahu implementovat daný postup. Například Ever-Miner je sice nadějný koncept, ale z důvodu neznalosti, časové a algoritmické náročnosti prozatím implementovatelný není. V každém PVO bude navrženo nejdříve nejjednodušší řešení, které má k implementovatelnosti nejblíže. Jestliže se řešení dá dále rozvíjet, tak budou navrženy postupy a nápady, jakým směrem by se mělo ubírat k případnému rozpracování (či budoucí implementaci).

Ulehčení práce analytika je kritérium důležité zejména z praktického hlediska. Navrhneme například PVO, řešící zadání některé úlohy pomocí ontologie, která se musí vytvořit. Tímto sice ubude práce na zadání úlohy, tato práce je však transformovaná do tvorby ontologie, která zadání vytvoří. Analytik, který zvládal zadávání úlohy, se nyní ještě musí naučit teorii ontologií, aby věděl jak má ontologii tvořit. Na druhou stranu poté co je ontologie vytvořena, je použitelná pro podobná data a podobné úlohy. Zatímco o kritériích obecnosti a implementovatelnosti můžu rozhodnout sám, z našeho (i když hypotetického) příkladu je zřejmé, že rozhodnout, zda PVO práci skutečně ulehčí, není triviální. Proto bude splnění tohoto kritéria konzultováno se zkušenými datovými analytiky (zejména s vedoucím diplomové práce Janem Rauchem). Rozhodnutí i argumentace bude zaznamenána v této práci.

[1] Neuvažovalo se o tom zejména z důvodu nedostatku lidí, kteří by mohli něco naprogramovat.

6.1. Krabička Mapování

6.1.1. Problém mapování ontologie a datového zdroje

Dříve než postoupíme k jednotlivým PVO, musíme vyřešit problém týkající se možnosti použití ontologií ve Ferdovi obecně. V každém z PVO nám budou jako vstupy figurovat dva „zdroje informací“, ontologie a datová matice, z nichž chceme strojově zpracovatelnou cestou získat nové znalosti, popřípadě zatím neznámé skutečnosti. Čtenář si jistě představí situaci, kdy máme k dispozici obecnou ontologii o doméně, a někdo nám dá data z určitého místa. V ontologiích jsou veškeré objekty, ať už třídy nebo sloty identifikované svými názvy. Podobně je tomu i u datových zdrojů. Lehce se může stát, že tyto názvy budou v některých případech odlišné, nebo úplně jiné. Aby jakýkoli postup pro využití ontologií mohl být úspěšný, musíme nejdříve zajistit mapování názvů ontologie na názvy datového zdroje – spojit sémanticky si odpovídající pojmy obou.

V praxi se však ukazuje, ve většině případů toto mapování nemůžeme ad hoc provést. Většinou k analyzovaným datům neexistuje ontologie, takže o mapování nemůže být řeč. Jestliže přece ontologie existuje, tak názvy reálných entit jsou různé v ontologii a datové matici, kde se většinou jedná o zkratky či první písmena názvů entit. Výjimkou je např. ontologie UMLS_2 v [5], zde byla však ontologie vytvořena uměle s ohledem na datovou matici (STULONG).

K nalezení co nejobecnějšího způsobu řešení mapování si stanovíme některé požadavky, které by tento způsob měl splňovat. Tyto jsou zahrnuty v následujících bodech:

- Nezávislost na použité technologii (hlavně formát připojení datového zdroje)
- Názvy v ontologii i v datovém zdroji se nesmí modifikovat
- Znovupoužitelnost mapování

Požadavek *nezávislosti na použité technologii* platí zejména pro jména z datového zdroje. Ferda i LISp-Miner používají pro připojení k datovým zdrojům rozhraní ODBC. Přes rozhraní lze připojit datové zdroje různých formátů, přes databázové připojení na server, soubory Microsoft Access™, až po zdroje v textových souborech. Naproti tomu pro ontologie se prozatím počítá s využitím jazyka OWL [22], což je jazyk na bázi XML. V XML mohou být názvy tříd ontologie libovolně dlouhé a za použití unicode kódování odpadá problém s diakritikou. Totéž se ale nedá říct o ODBC připojeních. Ačkoli moderní databáze podporují například názvy sloupců s mezerou, může se stát, že některé z ODBC formátů je podporovat nebudou[1]. Podobně to je s diakritikou a kódováním.

Požadavek na *zákaz modifikace ontologie či datového zdroje* je přirozený a plyne z povahy úloh dobývání znalostí. Například by nás mohlo napadnout přejmenovávat názvy v ontologii podle názvů datového zdroje (či opačně). Ontologie i datový zdroj používáme jako vstupní data, výsledky dobývání znalostí do nich nezapisujeme[2]. Může se stát, že nemáme právo na zápis do databáze (o to víc nemůžeme přejmenovávat sloupce a měnit tím schéma), nemáme právo na zápis do ontologie anebo ontologie může mít systém požadavků na změnu (pokud ji využívá více vědeckých teamů). Z těchto důvodů nemá smysl uvažovat o přejmenování a je nutné

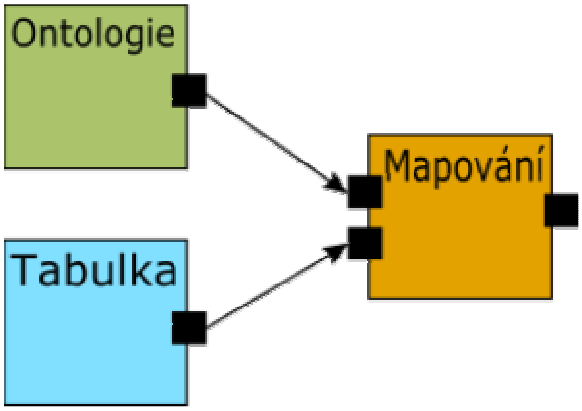
[1] Také patří k dobrým praktikám databázového designu nepoužívat mezeru v názvech sloupců.
[2] Tvrzení platí obecně pouze u datového zdroje, v kapitole bude uveden PVO, kde chceme propagovat nalezenou znalost zpět do ontologie, tedy do ní zapisujeme.

najít jiné řešení. Je důležité podotknout, že přejmenování nemusí naplňovat ani první požadavek nezávislosti na technologii.

Třetí požadavek je praktický a sledujeme jím hlavně ušetření práce experta provádějící mapování. Jak je zmíněno v [4], prozatím nepočítáme se strojovým prováděním mapování. Je tedy důležité provedené mapování expertem někde zaznamenat, aby se už příště nemuselo dělat, popřípadě při lehkých změnách vstupů (ontologie či zdroje dat) snadno modifikovat zaznamenané mapování.

6.1.2. Návrh krabičky

V této části bude navržena krabička *Mapování*, která splňuje všechny požadavky, které byly definované v předchozím textu. Základní schéma zapojení krabiček je zobrazeno na obrázku 10.



Obrázek 10: Návrh zapojení krabičky Mapování

Návrh vychází z krabičky *Tabulka*, která již je ve Ferdovi implementovaná. Jejím výstupem jsou informace o jedné tabulce datového zdroje. Další krabička, která zatím implementovaná není, je krabička *Ontologie*. Měla by mít podobnou funkcionalitu jako krabička *Databáze*, pomocí níž se dá napojit k datovému zdroji. Tedy nesloužila by k tvorbě ontologií, nýbrž by měl uživatel mít možnost připojit se ať už k ontologii vytvořené na místním počítači pomocí některého z programů pro tvorbu ontologií[1], anebo k ontologii na webovém serveru. Jednotná krabička *Ontologie* by se mohla použit ve všech postupech na využití ontologií ve Ferdovi. Výstupem této krabičky by měla být data o ontologii uložené ve vhodné datové struktuře[2]. I když je funkčnost krabičky *Ontologie* na první pohled jednoduchá, dá se předpokládat, že bude její implementace značně složitá. Důvody jsou shrnuty v části 5.1. Je však nezbytné tuto krabičku sestrojit pro další použití ontologií.

Přejděme ke krabičce *Mapování*. Tato krabička bude mít dvě zásuvky, jednu pro krabičku typu *Ontologie* a druhou pro krabičku typu *Tabulka*. Do zásuvky pro tabulky bude možno zapojit právě jednu krabičku tohoto typu, do zásuvky pro ontologie můžeme uvažovat o zapojení více ontologií. Krabička bude mít následující vlastnosti a akce:

Vlastnosti krabičky *Mapování:*

- Soubor mapování
- Editace mapování

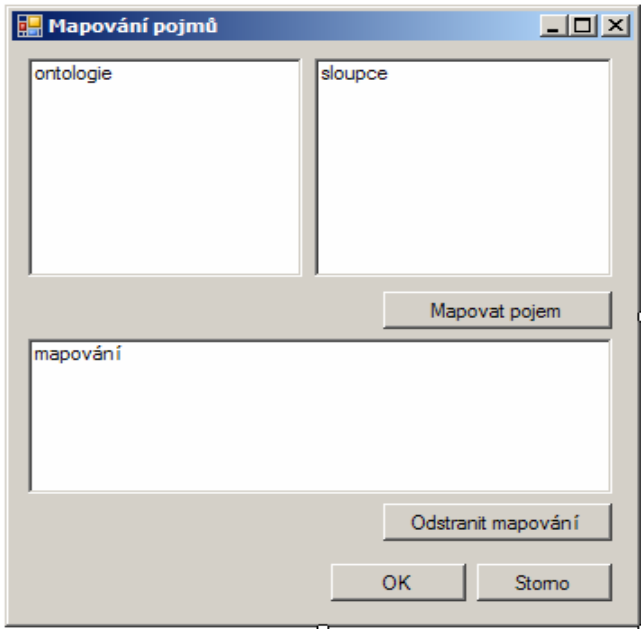
Akce krabičky *Mapování*:

[1] Například nástroj Protégé [38]
[2] Struktura musí být vhodná zejména pro middleware ICE, který Ferda používá.

- Uložit do souboru
- Nahrát ze souboru

Do vlastnosti *Soubor mapování* uživatel zadává cestu k souboru, do kterého se bude ukládat mapování. Mohli bychom také uvažovat o uložení mapování do souboru typu *xfp* (XML formát do kterého si Ferda ukládá svoje projekty). Chceme však, aby se dalo mapování použít pro více projektů, proto je vhodný samostatný soubor.

Vlastnost *Editace mapování* se bude nastavovat pomocí modulu pro nastavení, který je nezbytným doplňkem ke krabičce *Mapování*. Informace o tom, co je to modul pro nastavení v prostředí Ferda, je v [15], nebo dokumentaci k systému na [34]. Jestliže uživatel klikne na vlastnost *Editace mapování*, zobrazí se mu dialog modulu pro nastavení, kde uživatel spojuje jednotlivé názvy. Návrh jednoduchého dialogu je na obrázku 11.



Obrázek 11: Dialog pro mapování pojmů ontologie a sloupců datového zdroje

V horní části se nachází dva seznamy, jeden obsahuje názvy z ontologie, druhý obsahuje názvy sloupců. Uživatel označí termín z ontologie nalevo a sloupec napravo a poté klikne na tlačítko *Mapovat pojem*[1]. V této chvíli systém provede mapování pojmů a zobrazí je do třetího seznamu s názvem *Mapování*. Seznam obsahuje všechny již namapované pojmy. Tlačítko *Odstranit mapování* odstraní mapování vyznačené v seznamu *Mapování*. Dále dialog obsahuje klasické tlačítka *Ok* a *Storno*.

Akce krabičky slouží ke serializaci uživatelem provedeného mapování. Akce *Uložit do souboru* uloží mapování ve vhodném formátu na místo určené vlastností *Soubor mapování* a akce *Nahrát ze souboru* nahraje mapování. Předpokládáme dále, že při nahrávání mapování se bude muset provést nějaká kontrola konzistence.

6.1.3. Možnosti dalšího rozvoje

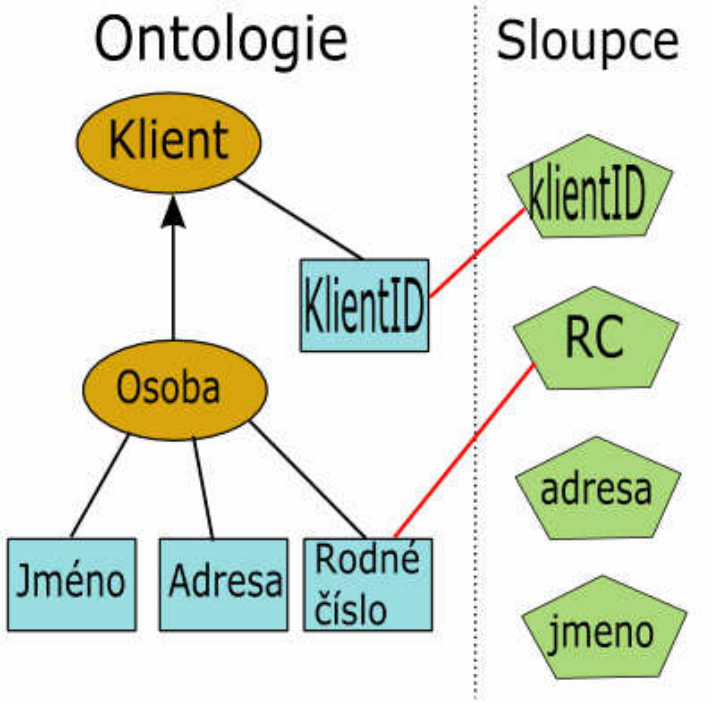
Výše uvedený návrh splňuje požadavky na něj kladené v předchozím textu. Je nezávislý na použité (databázové) technologii, nemodifikuje názvy vstupů a pomáhá analytikovi tím, že ukládá mapování do souboru. Existují však potenciální problémy, které krabička neřeší. Předpokládali jsme, že budeme mapovat pouze vztahy 1:1, což by odpovídalo korektnímu designu ontologie i datového zdroje. V reálném světe se však

[1] Pro jednoduchost předpokládáme, že pojmy se budou mapovat poměrem 1:1.

mohou vyskytnout případy mapování 1:N, například při existenci redundantních atributů, o kterých budeme mluvit později. Mohl by také nastat extrémní případ mapování M:N. V tomto případě by se muselo přepracovat uživatelské rozhraní dialogu, aby se v něm dalo mapování vhodným způsobem řešit.

Návrh má další problém – škálovatelnost. Pro ontologie s tisíce třídami nebo pro data se stovkami sloupců se stává ruční postup mapování neúnosný. Můžeme tedy uvažovat o použití vhodných algoritmů provádějících mapování automaticky, například pracujících na principu porovnání názvů. Analytik by měl mít možnost v případě nesrovnalostí výsledné mapování editovat.

Zlepšení mapování by pro analytika jistě přinesla vizualizace postupu. Navrhovaný dialog je jednoduchý a nezachycuje vztahy v ontologii. Uživatel by mohl prvky ontologie (vzájemně propojené podle vztahů) i názvy sloupců jako vizuální prvky, které propojuje přetahováním myší. Obrázek 12. ukazuje vizualizaci postupu mapování.



Obrázek 12: Vizualizace mapování. Ovály reprezentují třídy, obdélníky sloty, pětiúhelníky sloupce, šipka je dědičnost, červená čára značí již namapované vztahy.

6.2. Využití ontologie pro identifikaci chybějících atributů

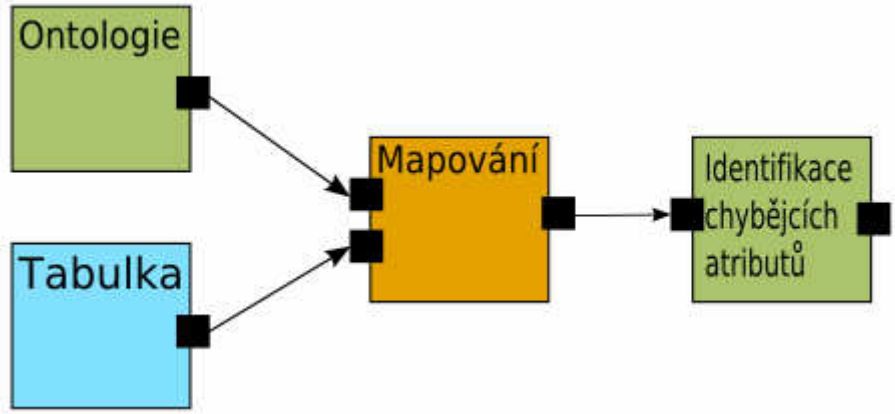
V [4] se píše bez větších detailů o využití ontologie pro identifikaci chybějících atributů za pomocí ontologií. Autoři tento PVO správně zařadili do fáze *Data understanding* metodologie CRISP-DM. Následující odstavce rozvádějí tento nápad, uvádí možnosti i problémy řešení a nabízí návrh implementace pomocí krabiček v prostředí Ferda.

Základní idea spočívá v tom, že program z (jakékoli) ontologie vytáhne informace o vztazích mezi třídami a sloty dané ontologie a ty potom porovná se sloupci v datovém zdroji. Výsledkem by měla být zpráva, která analyzuje datovou matici, vhodným způsobem seskupuje sloupce a případně informuje o chybějících atributech[1].

[1] Zde narážíme na problém s terminologií. Název identifikace chybějících atributů má historický důvod, byl tak pojmenován v předchozích pracích o využití ontologií. Postup zde uvedený pracuje pouze se sloupci, ačkoli je jasné, že se ze sloupců atributy později stanou. Proto ponecháváme původní název.

6.2.1. Návrh krabičky

Základní návrh pro krabičkovou implementaci v prostředí Ferda vidíme na obrázku 13.



Obrázek 13 – Návrh krabiček pro identifikaci chybějících atributů

Za krabičku *Mapování* se dále zapojí nová krabička *Identifikace chybějících atributů*. Tato krabička bude mít jednu zásuvku pro krabičky typu *Mapování*. Krabička bude mít následující akce a moduly pro interakci[1]:

Akce krabičky *Identifikace chybějících atributů:*

- Identifikovat chybějící atributy

Moduly pro interakci krabičky *Identifikace chybějících atributů*:

- Zobrazení chybějících atributů

Krabička se bude svým chováním velmi podobná krabičkám jednotlivých dobývacích procedur. Akce *Identifikovat chybějící atributy* spustí algoritmus identifikace chybějících atributů, který bude popsaný dále. Na zobrazení výsledků slouží modul pro interakci *Zobrazení chybějících atributů*, který poskytuje řazení atributů (sloupců) na základě zkoumané ontologie, popřípadě označí některé atributy za chybějící. Jako u krabiček procedur budou k dispozici *read-only* vlastnosti seznamující uživatele s výsledky běhu algoritmu. Konkrétní vlastnosti krabičky a uživatelská podoba modulu pro interakci jsou předmětem případné implementace krabičky. Konečně výstupem krabičky by měli být chybějící atributy ve vhodné datové reprezentaci.

6.2.2. Návrh algoritmu

Pro úspěšné fungování algoritmu na identifikaci předpokládáme, že všechny sloupcové entity budou v ontologii zastoupeny primárně jako sloty tříd (dále to mohou být například také třídy či instance tříd). Prvním krokem algoritmu je identifikace všech relevantních tříd, tj. tříd, které obsahují alespoň jeden slot shodný se sloupcem. Zde velmi záleží na správném mapování pojmů. Krok bude zřejmě probíhat procházením všech tříd v ontologii, přičemž každá se bude porovnávat s množinou sloupců[2]. Po tomto kroku pracujeme jen s množinou relevantních tříd. Algoritmus vhodně seskupí sloupce do skupin podle toho, jak vystupují korespondující sloty ve třídách v ontologii.

[1] Opět přesné informace o tom, co je modul pro interakci je v [15], nebo dokumentaci k systému Ferda na [34].

[2] Tento postup volíme, neboť přístup k ontologii bude zřejmě pomalejší než přístup k databázi, ontologie může být také rozsáhlá. Situaci také komplikuje taxonomie („dědičnost“) tříd – jestliže narazíme na potomka nějaké třídy, musíme zkoumat jeho sloty i sloty předka.

Jestliže je ve třídě slot, který se nevyskytuje v podobě sloupce, označí algoritmus tento slot za chybějící atribut.

Jestliže budeme zapojovat obecnou ontologii, předpoklad o slotech tříd však nemusí platit. Může se například stát, že některé sloupce budou v ontologii vyjádřeny pouze pomocí třídy, nebo to také mohou být instance některých tříd. V tomto případě algoritmus nebude dobře fungovat a nevrátí nám žádné údaje (i když bychom zajímavé informace dostat mohli). Mohli bychom algoritmus ještě vylepšit a obohatit ho o některé výjimečné postupy, anebo na druhou stranu se omezit jenom na ontologie, které obsahují entity odpovídající sloupcům jako sloty tříd. Vzhledem k tomu, že pro většinu zkoumaných datových matic ontologie neexistují a musí se vytvořit, není problém, aby se při tvorbě ontologií myslelo na toto omezení.

Další výkonnostní problém by mohl nastat s velkými ontologiemi, které jsou umístněny na webovém serveru přes pomalé připojení. První fáze algoritmu, tedy získání množiny relevantních tříd by potom mohla trvat neúměrně dlouho. I když se to se předpokládá, že vstupem pro krabičku budou většinou malé ontologie umístněné na lokálním disku, na tento problém by měla případná implementace brát zřetel. Zde nutno poznamenat, že tento problém přímo souvisí se škálovatelností krabičky *Mapování* a protože ta prozatím není vyřešená, vstupní ontologie by se ani nenamapovala (tudíž by algoritmus neproběhl).

6.2.3. Příklad použití

Následující příklad přiblíží praktické využití postupu identifikace chybějících atributů. Mějme databází Barbora[7] o údajích z fiktivní banky. Tato obsahuje tabulku *Loans* se sloupci: *birth_number, District, duration, loan_id, payments, Salary* a *status.* Nad ní nechť máme postavenou jednoduchou ontologii. Ontologie obsahuje pouze dvě třídy a to *zákazník* a *úvěr*. Třída úvěr má sloty: *číslo úvěru, částka, doba trvání, výše splátky* a *status*. Třída zákazník má sloty *rodné číslo, jméno, ulice, město, okres, plat* a *rodinný stav*.

Uživatel ve Ferdovi zapojí krabičku tabulky *Loans* a krabičku ontologie do nové krabičky *Mapování* a vytvoří mapování pomocí dialogu z obrázku 5. Anglické a české názvy byly zvoleny úmyslně, aby byla zdůrazněna potřeba mapování. Dále do mapování zapojí krabičku *Identifikace chybějících atributů*, jako na obrázku 7. Poté spustí akci *Identifikovat chybějící atributy* této krabičky. Proběhne algoritmus a uživatel se na výsledek podívá modulem pro interakci *Zobrazení chybějících atributů.* Modul mu seskupí sloupce do skupin podle tříd v ontologii – v první skupině bude mít sloupce *birth_number, district* a *Salary*. Do druhé skupiny budou patřit sloupce *amount, duration, loan_id, payment* a *status.* Dále budou identifikované chybějící atributy: *jméno, ulice, město* a *rodinný stav* u třídy *zákazník*.

6.2.4. Diskuse o „vhodnosti“

Z příkladu je vidět, že PVO rozhodně ulehčí prácí analytika ve fázi *Data understanding*. Nejenom že identifikuje atributy, které se nevyskytují v datech (například atribut *rodinný stav* z příkladu by mohl být pro analýzu zajímavý), PVO také seskupí sloupce do skupin podle tříd. Toto může být významná informace pro někoho, kdo o datech vůbec nic neví[1]. Důležitou informací může být už jenom samostatné mapování pojmů, v ontologiích často existuje vedle názvu i popis slotu nebo třídy (tato

[1] Pro účely výuky systému LISp-Miner se používá databáze STULONG[8]. Ta obsahuje tabulku *entry* nad kterou se nejčastěji dobývá s více než šedesáti sloupci z lékařské domény. Pro začínajícího studenta jsou názvy sloupců velkou neznámou (názvy jako ICTL nebo SUBSC) a často neví, kde najít vysvětlující informace. Mapování názvů i seskupování atributů do tříd mu může významně pomoci porozumět datům.

výhoda však pramení spíše z existence krabičky *Mapování* než *Identifikace redundantních atributů*).

Také zbývající dvě kritéria PVO dostatečně splňuje. Počítá se s zapojením obecné ontologie a PVO je implementovatelný. Problémem je zejména fakt, že kvalita tohoto postupu přímo závisí na kvalitě vstupních dat. Jestliže budeme mít kvalitní vstupní ontologii, datovou matici a bude možné vytvořit rozumné mapování, tak potom výsledky budou uspokojivé. Když však nemáme dobrou ontologii, anebo není možné vytvořit mapování názvů, zřejmě algoritmus žádné chybějící atributy neprozradí ani neseskupí sloupce do skupin podle tříd. Pro rozhodnutí o případné implementaci postupu je tedy nutné zvážit i dostupnost a relevanci vstupních dat.

6.2.5. Možnosti dalšího rozvoje

V současné době pracují všechny GUHA procedury implementované ve Ferdovi nad jednou tabulkou v databázi. Plánuje se však rozšířit Ferdu o relační dobývání znalostí, tj. dobývání znalostí pomocí již existujících procedur nad tabulkami ve schématu hvězdy. O implementaci procedury 4FT se můžete dočíst v [13]. Bylo by tedy vhodné hledat chybějící atributy nejenom ze sloupců jedné tabulky, nýbrž z celého datového zdroje. Musel by se zřejmě modifikovat algoritmus, abychom v něm zohlednili možnosti databázového schématu.

6.3. Tvorba atributů pomocí ontologií

6.3.1. Tvorba atributů ve Ferdovi

Tvorba atributů je jedním ze základních kroků fáze *Data preparation* metodologie CRISP-DM. Jak už bylo v práci několikrát zmíněno, vhodná kategorizace domény je nezbytná k úspěšnému dobývání znalostí. Ve Ferdovi (i v systému LISp-Miner) jsou k dispozici tři typy automatické kategorizace: ekvidistanční, ekvifrekvenční a *each value one category*. Uživatel si může také vytvořit atribut ručně, tj. vytvořit si kategorie a do nich vkládat jednotlivé hodnoty. Nabízí se tedy relativně široké možnosti tvorby atributu, avšak v některých případech tyto možnosti nestačí.

Pro číselné veličiny, které mají jisté významné hodnoty nerovnoměrně distribuované (podle nichž by se mělo kategorizovat), systém nabízí jen malou podporu. Příkladem je veličina *Body mass index*, měřící „tloušťku“ člověka, která se používá u medicínských studií[1]. Obecně se dá říct, že člověk s BMI pod 20(kg/m^2) je podvyživený, nad 25 má nadváhu a nad 30 už trpí obezitou. Samozřejmě je výhodné použít toto rozdělení pro tvorbu atributů. Ve stávajícím systému by však analytik musel atribut tvořit ručně, což je pracné. Analytik také nemusí přesně znát významné hodnoty veličiny, aby podle nich atribut vytvořil. V tomto případě, by to mělo dopad na kvalitu dobývání znalostí.

6.3.2. Využití ontologií při tvorbě atributů

Nabízí se tedy využití znalostí v ontologiích pro tvorbu atributů. Ontologie může obsahovat informace o datových typech, maximálních a minimálních hodnotách, které by mohli uživateli pomoci při tvorbě atributu, popřípadě pro něj atribut vytvořit. Tento PVO nebyl zmíněn v žádné z literatury [5], [4] nebo [35]. Jedná se o velmi perspektivní PVO, jehož plné možnosti přesahují rámec této práce. Jeho plná implementace by významně umožnila implementaci části *Konstrukce atributů* první vrstvy systému Ever-

[1] Krabička *Body mass index* je také ukázková krabička implementovaná v prostředí Ferda. Uživatel si může do ní zapojít sloupce výška a váha a krabička sama spočítá BMI. Dá se použít jako sloupec i jako atribut.

Miner (*Vrstva 1 – EverMiner basics*), který je popsán v části 2.4.1[1]. V následujících odstavcích bych chtěl upozornit na některé problémy a navrhnout jejich možné řešení, ukázat možný krabičkový návrh PVO a zmínit možnosti dalšího rozvoje.

6.3.3. Ontologie pro tvorbu atributů

Na začátku kapitoly jsme si definovali obecnost jako jedno z kritérií na postupy využití ontologií. Zkusme tedy hypoteticky vyčíst informace z jakékoli doménové ontologie. O ontologii nic nevíme, známe jenom její strukturu. Tedy známe to, jak jsou sloty seskupené do tříd, známe taxonomii mezi třídami, můžeme znát některé instance tříd. Nechť máme dále ontologii zapojenou pomocí krabičky *Ontologie* do krabičky *Mapování*, která identifikuje entity v ontologii, které se vyskytují v tabulce. Víme už trochu více, ale jen to, kterými entitami se máme zabývat (ostatní nelze namapovat na sloupec, tudíž z nich nejde vytvořit atribut). Stále je ale obtížné strojově z těchto informací vytvořit kategorizaci atributů. Projekce mapování do ontologie může být svým typem slot, třída nebo dokonce instance. Je-li to slot, například typu celé číslo, víme, že z něj lze tvořit ekvidistanční nebo ekvifrekvenční atributy a možná budou v ontologii informace o minimálních a maximálních hodnotách. Je li to třída, musíme předpokládat nominální hodnoty (instance třídy) o kterých se nic víc neví.

Z tohoto rozboru vyplývá, že jakákoli ontologie nám vůbec nemusí dodat potřebné informace pro tvorbu atributů. Pro použití ontologie by musel být k dispozici také velmi chytrý algoritmus, nebo dedukční mechanizmus, který by získával informace z ontologie. Nyní navrhnu řešení, které poněkud slevují z kritéria obecnosti pro ontologie, zato však poskytují více možnosti pro automatickou tvorbu atributů. Na tomto místě je nutno upozornit, že se v průběhu práce ukazuje výhodné používat místo obecných specifičtější ontologie – například v algoritmu pro identifikaci chybějících atributů jsme také dali omezení na vystupování mapovaných pojmů jako slotů tříd v ontologii[2].

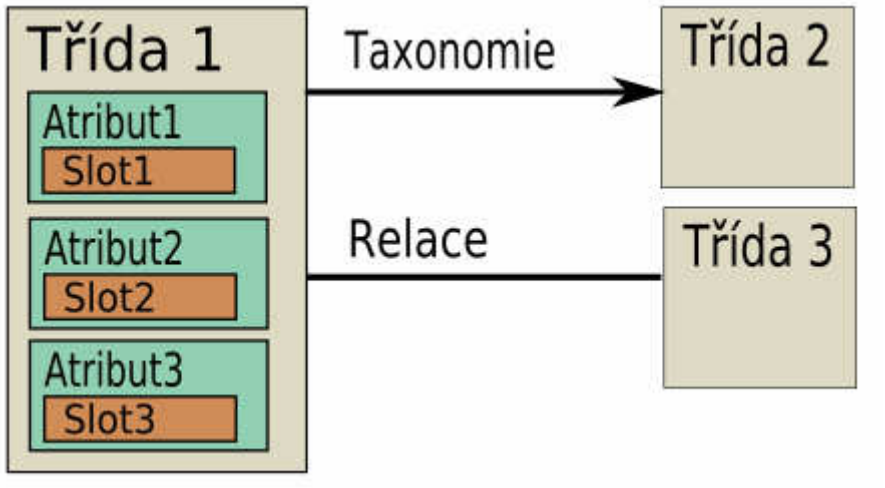
V současné době uvažujeme dvě možná řešení tohoto zásadního problému. První řešení používá „obalení“ slotů speciální třídou v ontologii. Ontologie to umožňují za pomocí meta-tříd (třída jejíž instance jsou třídy), nebo také *class fields* (slot jehož hodnota definuje novou třídu). Poté lze vytvořit třídu, která bude u každého slotu jiná (nemůžeme totiž použít instance tříd jako typy slotů) a bude obsahovat nějaké informace o slotu. Tato třída by obsahovala informace o atributu. Problém tohoto řešení je v použitém jazyku – je možný jenom v dialektu OWL Full.

Obrázek 14 ukazuje tvorbu ontologií s tím, že všechny sloty budou jistým potomkem *atributu*. Ostatní prvky struktury ontologií (relace, taxonomie) zůstávají.

Druhé řešení využívá trochu jiného typu ontologií. Tento typ se nazývá *prezentační ontologie* a podrobně se jimi zabývá [18]. *Prezentační ontologie* popisují třídy a jejich atributy a nemusí být nutně zapsány v OWL. Zásadní problém s doménovými ontologiemi v OWL byla právě syntaxe OWL, neboť neumožňuje vytvořit konstrukce, které se nejvíce hodí pro popis tříd. *Prezentační ontologie* by tento problém mohli řešit, avšak konkrétní řešení není prozatím rozmyšleno.

[1] PVO by zajistil pomocí ontologie tvorbu jedné krabičky *Atribut*. Ke konstrukci všech atributů zatím systém Ferda není uzpůsoben, neboť neumí sestrojovat „nadkrabičky“, jak se o nich píše taktéž v části 2.4.1.

[2] Jak už bylo zmíněno, vzhledem k tomu, že ontologie pro dané domény většinou neexistují, není problém tato omezení v úvahu. Například pro akademické účely je výhodné vytvořit kvalitní ontologii, kterou budou žáci používat.



Obrázek 14: Zabalení slotů v nových třídách pro atributy

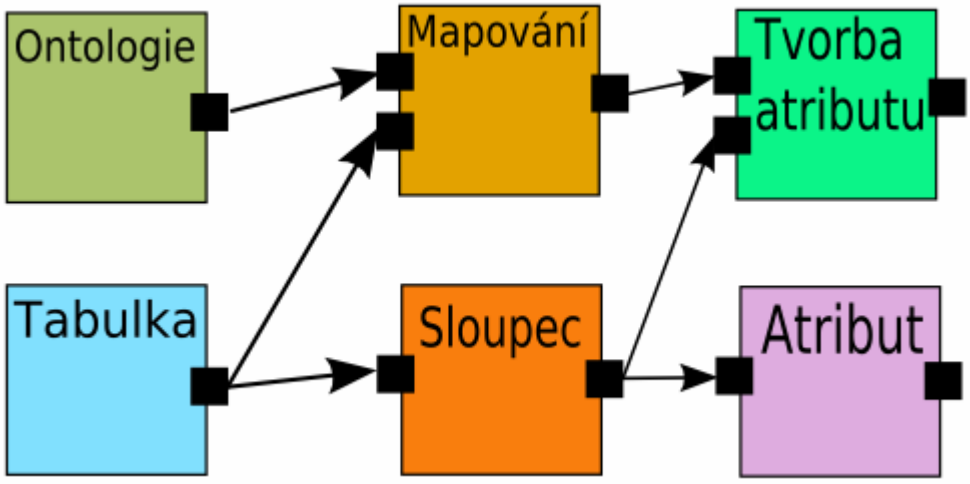
Vraťme se k diskusi o informacích, které by měly atributy nést. Následující seznam uvádí některé informace, které by se mohli nejvíc hodit:

- Datový typ
- Minimální a maximální hodnoty
- Extrémní hodnoty
- Významné hodnoty rozdělující doménu (pro ordinální hodnoty)
- Typické hodnoty (pro nominální hodnoty)

Za pomocí těchto informací můžeme pomocí vhodných pravidel konstruovat atributy. Například podle pravidla „Jestliže existují významné hodnoty rozdělující doménu, potom postav kategorie na základě těchto hodnot.“ by se rozdělil na vhodné kategorie sloupec BMI. Musíme však dodat, že ani podoba pravidel není rozmyšlena do té míry, aby mohl být PVO implementován.

6.3.4. Návrh krabičky

Návrh a zapojení krabičky *Tvorba atributu* bude také používat krabičku *Mapování* a schéma zapojení znázorněno na obrázku 15.



Obrázek 15: Zapojení krabičky pro tvorbu atributů

Dolní řada krabiček jsou krabičky implementované v systému Ferda. Z horní řady byly už popsány krabičky *Ontologie* a *Mapování*. Nová krabička *Tvorba atributu* má dvě zásuvky, jednu pro krabičky typu *Mapování* a druhou pro krabičky typu *Sloupec*. Krabička bude vytvářet novou krabičku typu *Atribut* pomocí *krabiček nabízených na vytvoření* (mechanizmus popsaný v [15], nebo dokumentaci k systému Ferda na [34]). Uživatel tedy klikne pravým tlačítkem na krabičku *Tvorba atributu*, vybere z kontextového podmenu *Krabičky nabízené na vytvoření* položku jmenující se stejně jako zapojený sloupec. Systém potom vytvoří krabičku (statický) *Atribut*, zapojí do ní sloupec a vytvoří se kategorie podle pravidel v krabičce *Tvorba atributu*.

Krabička bude zřejmě obsahovat vlastnosti, které budou modifikovat pravidla, nebo jejich prioritu při tvorbě atributu. Mohli bychom také místo několika jednoduchých pravidel uvažovat o využití složitějšího expertního systému a vlastnosti krabičky by mohli sloužit jako parametry pro expertní systém.

6.3.5. Diskuse o „vhodnosti“

Kritérium vhodnosti, které PVO jednoznačně splňuje je ulehčení práce analytika. Analytik, který o datech neví mnoho, si může pomocí PVO vytvořit atributy dobře odpovídající realitě. I pro analytika znalého dat se může PVO hodit – například pro BMI sloupec analytik nemusí tvořit kategorie ručně, postup mu je vytvoří automaticky.

Kritérium obecnosti jsme probrali diskusí o zapojení obecných ontologií. Výsledkem této diskuse je rozhodnutí o nepoužívání obecných ontologií a návrh modifikovaných ontologií, které by hodily. Tedy kritérium není úplně splněno.

Největší problém však stále zůstává v implementovatelnosti. Většina nápadů pro automatickou tvorbu ontologií není rozvinuta do té míry, že by se podle nich mohlo začít programovat. Musel by se přesně definovat formát ontologií, které s krabičkou spolupracují, hlavně definice zásadní meta-třídy *Atribut*. Z této definice by potom vycházel seznam pravidel pro případný expertní systém.

6.3.6. Možnosti dalšího rozvoje

PVO tvorby atributů nabízí velké možnosti rozvoje. Zejména se jedná o neustálé vylepšování množiny pravidel, které určují tvorbu atributů. Po případné implementaci by se mělo kriticky diskutovat o relevanci atributů, které byly krabičkou vytvořeny. Důležitá je také podoba a zlepšování expertního systému.

Dalším logickým rozšířením je automatizovaná tvorba dalších krabiček zadání úlohy pomocí ontologie. Mohli bychom například konstruovat krabičku *Zadání atomu* (vybírání typu koeficientu), nebo i další krabičky až po zadání cedentů. Možnost tvorby atributů i atomů pomocí heuristik byla zmíněna v [14], průkopnickou prací na toto téma je [32] a také [33]. V těchto pracích se však neuvažovalo o použití ontologií. Všechny tyto rozšíření spadají do části *Konstrukce úlohy* první vrstvy systému Ever-Miner definované v této práci.

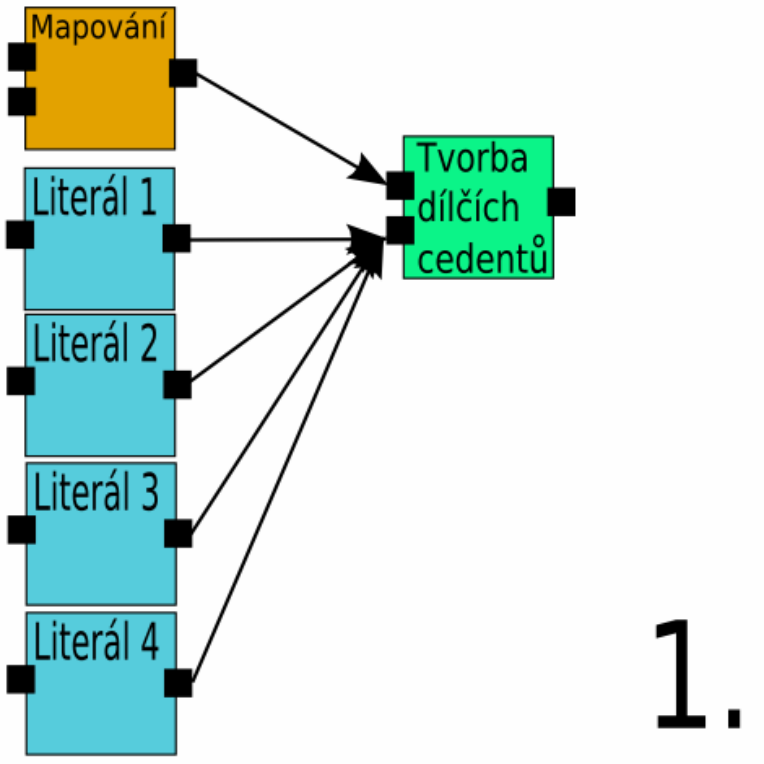
6.4. Konstrukce dílčích cedentů za pomoci ontologie

Myšlenka konstrukce dílčích cedentů se objevila v pracích [5] a [4], dokonce se autoři v [4] zmiňují o možné automatizaci PVO. Idea je stejná jako u identifikaci chybějících atributů: namapují se jednotlivé zadání cedentů na entity z ontologie a ty potom podle vztahů třída-slot vytvoří dílčí cedenty. Přirozenými kandidáty pro jeden dílčí cedent jsou zadání cedentů, které reprezentují sloty stejné třídy. Algoritmus pro nalezení dílčích cedentů je tudíž shodný s algoritmem v části 5.2.2. (s tím že onen algoritmus navíc identifikuje redundantní atributy). Rozdíl je hlavně ve fázi dobývání

znalostí z hlediska CRISP-DM, identifikace chybějících atributů spadá do fáze *Data understanding* a tudíž jenom informuje uživatele. Konstrukce dílčích cedentů už patří do fáze *Data preparation* a PVO se užívá pro konstrukce úloh. V následujících odstavcích bude stručně představeno krabičkové řešení v systému Ferda a provedena diskuse na téma „vhodnost“.

6.4.1. Návrh krabičky Konstrukce dílčích cedentů ve Ferdovi

Navrhované použití v krabičkovém modelu začíná výchozí situací, která je zachycená na obrázku 16.

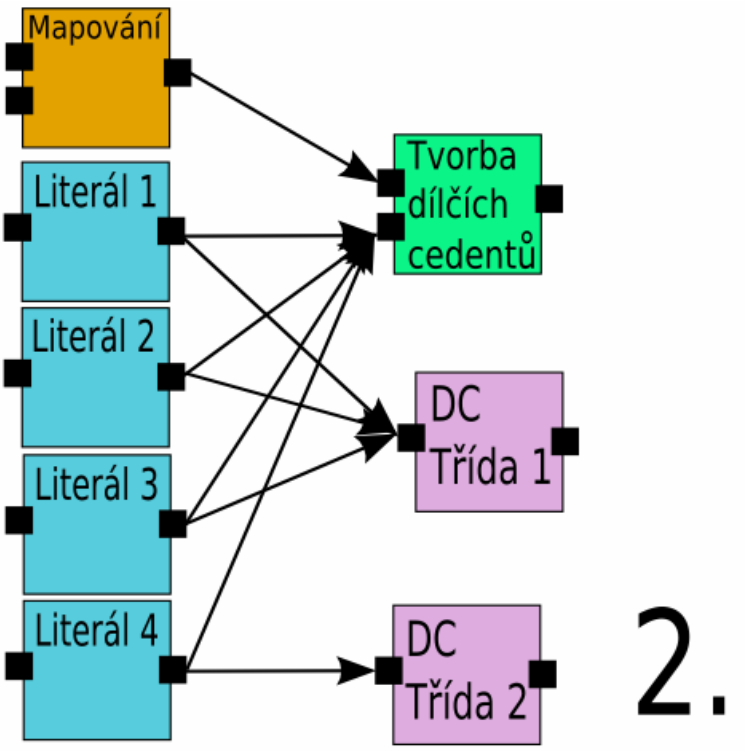


Obrázek 16: Situace před použitím krabičky *Tvorba dílčích cedentů*

Opět se zde vyskytuje krabička *Mapování*, která mapuje pojmy ontologie a datové tabulky. Tato krabička je zapojena do nové krabičky *Tvorba dílčích cedentů*, která má mimo dvě zásuvky. Druhá zásuvka je pro krabičky typu *Literál*, které jsou ve Ferdovi implementované[1]. Do ní zapojíme všechny krabičky pro literály, ze kterých chceme vytvořit dílčí cedenty. Krabička *Tvorba dílčích atributů* bude nabízet na vytvoření krabičky dílčích cedentů, které odpovídají tvorbě dílčích cedentů podle tříd ontologie. Nechť například podle obrázku 10 budou v ontologii informace o dvou třídách, kde *třída1* bude obsahovat sloty odpovídající literálům 1, 2 a 3, a *třída2* bude mít laterál 4. Obrázek 17 ukazuje situaci, kdy uživatel vytvoří všechny dílčí cedenty pomocí mechanizmu *Krabičky nabízené na vytvoření*[2].

[1] Toto zadání platí pouze pro tvorbu booleovských dílčích cedentů. Mohli bychom modifikovat krabičku *Tvorba dílčích cedentů* tak, aby zásuvka, která přijímá literály, přijímala i atributy (pro konstrukci kategoriálních dílčích cedentů). Zde by algoritmus seskupování atributů do skupin nefungoval tak elegantně, protože bychom nemohli tvořit cedenty po skupinách podle ontologie, ale museli bychom ještě brát ohled na druh cedentu.

[2] Uživatel si bude moci vybrat, jestli vytvoří jen některé cedenty, nebo všechny cedenty najednou kliknutím na odpovídající položku v menu.



Obrázek 17: Krabička *Tvorba dílčích cedentů* vytvořila dva nové dílčí cedenty

Na ploše přibyly nové krabičky typu *Booleovský cedent*, do kterého jsou zapojené literály tak, že dílčí cedenty odpovídají třídám z ontologie. Uživatel s nimi dále může pokračovat při konstrukci úlohy.

6.4.2. Diskuse o „vhodnosti“

Jako u postupu identifikace chybějících atributů splňuje PVO všechna kritéria, která jsme na začátku kapitoly stanovili. Opět musíme upozornit, že kvalita výstupu přímo závisí jak na kvalitě a dostupnosti ontologie, tak i na existenci korektního mapování. Z pohledu analytika by také mohla být zajímavá spolupráce mezi identifikací chybějících atributů a konstrukcí dílčích cedentů. Analytik se nejdříve z identifikace dozví něco o datech (ve fázi *Data understanding*) a poté si nechá systémem vytvořit dílčí cedenty (ve fázi *Data preparation*), což může zejména pro velké množství atributů ušetřit práci.

6.5. PVO nevhodné pro implementaci v prostředí Ferda

Všechny předchozí PVO se ukázaly jako vhodné pro implementaci v prostředí Ferda. Literatura [5], [4] a [35] však obsahuje i další PVO, které vhodné pro implementaci nejsou. Pro úplnost jsou tyto PVO uvedeny v následující části a je u nich napsané, z jakých důvodů není vhodné o implementaci uvažovat.

6.5.1. Využití ontologie pro identifikaci redundantních atributů

Autoři v [4] zmiňují, že ve fázi *Data understanding* by mohla být ontologie využita i pro identifikaci redundantních atributů. Znamená to, že PVO by pomocí ontologie určil atributy (sloupce datové matice), které reprezentují stejnou reálnou entitu. V [4] například atributy *age on entrance to STULONG study* a kombinace atributů *birth year* a *year of entrance to STULONG study* vyjadřují shodnou entitu *Age* v ontologii. PVO by měl na základě ontologie a datové matice algoritmicky (popřípadě pomocí expertního systému) identifikovat jeden z těchto atributů jako redundantní.

Kdybychom se rozhodli identifikaci redundantních atributů implementovat, krabičkové řešení a detaily návrhu by byly velmi podobné jako u *Identifikace chybějících atributů*. Dokonce bychom mohli sloučit krabičky *Identifikace chybějících atributů* a *Identifikace redundantních atributů* do jedné. Přesto si však myslím, že není vhodné krabičku tvořit. Můžeme předpokládat přirozenou „racionalitu" tvůrců datového zdroje (návrh databázového schématu a tabulek), tedy že nikdo nevytváří tabulku tak, aby tam měl dva sloupce vyjadřující to samé. Myslím si, že k redundanci atributů bude docházet zřídka a když už k ní dojde, tak to bude třeba jako v [4] – jedná se o skrytou redundanci kombinací sloupců. V těchto komplikovaných případech potřebujeme jednak složitý algoritmus, ale také spolehlivé mapování názvů. K mapování je zapotřebí expert, který rozumí pojmům z ontologie i významu sloupců datové matice. Předpokládám, že při provádění mapování není těžké redundance odhalit a expert na ně narazí, popřípadě opraví v datové matici. Není na to tedy třeba krabička.

6.5.2. Tvorba úloh pomocí ontologií

Nápad na tvorbu úloh podle ontologie se objevil v [5] a počítá s vytvořením úlohy pro některou z procedur podle relace nebo relací mezi třídami v ontologii. Realizace tohoto nápadu by měla vytvořit konkrétní úlohu konkrétní procedury, přesunuli jsme se tedy z fáze *Data preparation* do fáze *Modelling* v terminologii CRISP-DM. Z textu v [5] není jasné, zda autorka předpokládá ruční nebo automatickou tvorbu úloh. V této práci se budeme zabývat automatickou tvorbou úloh. Protože problém úzce souvisí s analýzou první vrstvy systému Ever-Miner *Konstrukce úlohy*, uvedeme v této části práce podrobnější úvahy o problému. Protože vycházíme z literatury [5], budeme se v této části zabývat konstrukcí úlohy pouze pomocí z vhodné ontologie.

Pro naši úvahu předpokládejme, že máme obecnou ontologii, která obsahuje vztahy mezi třídami v podobě relací a mapování ontologie na datovou matici[1]. Chceme tedy vymyslet PVO, který automaticky využívá relací ontologie k tvorbě úloh. Princip je ten, že jednotlivé úlohy ověřují platnost relace či složení relací v datech. Předpokládejme dále, že máme automaticky vytvořené nějaké atributy, například pomocí postupu v kapitole 6.3. Pokud by nebyl k dispozici prostředek na automatickou tvorbu atributů, musel by uživatel tvořit atributy ručně. Musel by si tedy z množiny atributů vybrat a vytvořit ty, které jsou zajímavé pro analýzu (a tudíž by automatická tvorba úlohy neměla smysl). Za krátko však narazíme na problémy, které se musí vyřešit, aby se dal postup používat. Mezi nejvýznamnější patří:

- Která relace ontologie se má pro úlohu použít?
- Která procedura se má použít?
- Jak se má „sestavit" úloha a který kvantifikátor se má použít?

První problém výběru relace, podle které se má konstruovat úloha, je nejzásadnější. Pro malé ontologie obsahující desítky relací můžeme zkoušet konstruovat úlohy ze všech relací, přičemž je zřejmé, že některé z relací nemusí být pro experta v doméně zajímavé. Pro větší ontologie však nevíme, podle které relace máme data zkoumat. Diskuse na toto téma již byla provedena v kapitole 4.2. Také v [5] autorka zvažuje které relace použít a dochází k závěru, že pro nalezení zajímavých úloh bychom

[1] Například pomocí krabičky *Mapování*.

museli porovnávat každou třídu s každou[1]. To je ovšem neúnosné z hlediska časové náročnosti.

Dalším stále závažným problémem je výběr procedury, která se má použít na cedenty odpovídající třídám, které jsou spojené zkoumanou relací. V [5] se počítá jen s využitím procedury 4FT, ale mohli bychom stejně dobře použít proceduru KL[2]. Máme možnost buď sestrojit úlohu pro obě procedury, nechat uživatele proceduru zvolit, anebo implicitně použít 4FT jakožto nejpoužívanější proceduru. Jestliže implicitně použijeme 4FT, tak potom ztrácíme možnosti procedury KL. Jestliže necháme uživatele rozhodnout se, musí uživatel znát procedury, aby mohl mezi nimi vybírat, tudíž si stejně dobře může úlohu sestrojit sám.

Když už byla vybrána i relace i procedura, zbývá „sestavit" úlohu a vybrat kvantifikátor. „Sestavením" úlohy rozumíme konstrukci a zapojení krabiček od atributů ke krabičce procedury tak, aby bylo posléze možno proceduru spustit. Pro proceduru KL sestává dokončení úlohy z konstrukce kategoriálních cedentů z atributů, volby jejich délky (můžeme použít 1, 1) a zapojení do krabičky *KL Úloha*. Pro proceduru 4FT je však situace komplikovanější, neboť se musí konstruovat atomy, literály a dílčí cedenty s množstvím parametrů. Zde opět můžeme použít implicitní nastavení vlastností krabiček pro atomy, literály i dílčí cedenty, avšak oslabujeme tím sílu procedury. Daly by se také vymyslet heuristiky, které by z povahy a struktury dat vymyslely vhodné nastavení atomů, literálů a cedentů. Tyto heuristiky však prozatím nejsou vymyšlené.

Posledním problémem je výběr kvantifikátoru pro procedury. Můžeme ho vyřešit tak, že vymyslíme seznam nejpoužívanějších kvantifikátoru a ty používáme v procedurách[3]. Opět však přicházíme o možnosti, které ostatní kvantifikátory nabízejí.

Shrneme-li dosavadní poznatky, přicházíme k závěru, že pouze pomocí doménové ontologie (tedy ontologie obsahující znalosti o zkoumané doméně) lze sestrojit úlohu velmi špatně, nebo vůbec. Ať už jsou k dispozici jakkoli dobré znalosti o povaze a významu dat, stále z nich nelze vyčíst, co na nich bude analytik zkoumat. Proto je důležité se analytika, potažmo doménového experta zeptat co si přeje vyzkoumat a tedy jaká úloha se má sestrojit. Systém Ever-Miner na tento fakt pamatuje soustavou typových úloh, které se uživateli předvedou. Stav, ve kterém uživatel řekne systému, co chce zkoumat a potom systém sestrojí pomocí vhodných doménových i jiných znalostí úlohu (tedy obstará se o konstrukci atomů, literálů i cedentů) se jeví prozatím jako dostatečný. K zadání úlohy by třeba mohli sloužit pravidla *background knowledge*.

6.5.3. Dekompozice 4FT úloh v závislosti na ontologii

Ve textech [5], [4] a [35] autoři diskutují o možnosti rozdělení komplexních 4FT úloh s velkým množstvím obtížně interpretovatelných hypotéz na menší úlohy. Pokusíme se přiblížit situaci, kdy by mohlo dojít k dekompozici úlohy a vysvětlit důvody, proč je zbytečné uvažovat o automatizovaném nástroji pro tento PVO.

Častý postup, jak dobývat pomocí GUHA procedur, je sestrojit si nejdříve co největší možnou úlohu, tu potom spustit a posléze se pokusit interpretovat nejsilnější hypotézy a v případě dlouhých hypotéz (co se počtu literálů týče) úlohu rozdělit na

[1] Autorka však nepředpokládá jenom konstrukce úloh dle relací, tedy vztah jedné a druhé třídy, ale i úlohy vztahu více tříd k jedné třídě (tedy více dílčích cedentů v cedentu v řeči zadání) či vztahy vzniklé skládáním relací.

[2] Procedura CF se nehodí, protože zkoumá pouze jeden kategoriální atribut. Z důvodu jednoduchosti také neuvažujeme použití procedur SD4FT a SDKL.

[3] U procedury 4FT bude nejvhodnějším kandidátem na první použití kvantifikátor *fundované implikace* či *above average*.

několik menších. V tomto případě by se mohla hodit ontologie, neboť by pomohla rozdělit úlohu na několik sémanticky správných podcelků. V [35] je princip vysvětlen na příkladu: nad databází STULONG zkoumáme závislost mezi *aktivitou* a *nemocí*. V případě velkého množství hypotéz je možné rozdělit úlohu na části podle částí třídy *nemoc*: *aktivita ~ kardiovaskulární nemoc*, *aktivita ~ trávicí nebo oběhová nemoc*, *aktivita ~ hypertenze* atd.

Představme si však konstrukci této velké úlohy v prostředí Ferda. Uživatel musí nejdříve vhodně kategorizovat všechny sloupce, nad kterýma chce dobývat. Poté tvoří atomy a literály. Z literálů tvoří dílčí cedenty tak, aby jeden dílčí cedent zhruba odpovídal zkoumané entitě (například *kardiovaskulární nemoc*). K tomu mu může pomoci například PVO Konstrukce dílčích cedentů. Tyto cedenty zapojí do úlohy a úlohu spustí. Jestliže vyjde velké množství hypotéz a chce úlohu zjednodušit, potom jednoduše vypojí některé z dílčích cedentů z úlohy a úlohu spustí znovu. Nepotřebuje k tomu tedy žádnou ontologii, neboť všechny informace má už schované v zadání úlohy, tedy v zapojení krabiček.

6.5.4. Propagace zjištěných znalostí do ontologie

Posledním probíraným PVO v rámci této práce je propagace zjištěných znalostí zpět do ontologie. Ačkoli může nápad znít na první pohled velmi slibně, automatizované použití naráží na řadu problému. Prvním problémem je (automatizovaná) interpretace hypotézy vzešlé z běhu GUHA procedury. Jak bylo naznačeno v části 3.4, mohli bychom použít pro zapsání pravidla relaci nebo axiom v ontologii. Axiom však musí platit vždy, totéž se předpokládá o relaci mezi třídami. Avšak hypotéza znamená pouze potvrzení určitého statistického kritéria na určitých datech, tedy nehodí se ji hned psát automatizovaným nástrojem zpět do ontologie. Její relevanci by měl potvrdit doménový expert a až pak se může nějakým způsobem propagovat do ontologie. Není tedy důvod PVO automatizovat[1].

[1] Mnohem lepší použití výsledků dobývání než propagace znalosti zpět do ontologie se zdá být vytvoření slovní analytické zprávy pro doménového experta. Tým studentů MFF UK pracuje na tomto úkolu v projektu nazvaném *LM – Report Asistent* [23].

7. Validace background knowledge na sadě pravidel STULONG

V 5. kapitole jsme se dozvěděli o implementaci validace pravidel *background knowledge* v prostředí Ferda. Byl představen krabičkový návrh a algoritmy implementovaného nástroje. Po prvotním testování byl nástroj shledán jako funkční. Abychom však mohli podrobněji vyzkoušet užitečnost nástroje, je třeba nástroj dále testovat na obsáhlejším úkolu, který by se mohl vyskytnout ve skutečném nasazení. Jako úkol byla vybrána validace pravidel STULONG[8]. V minulosti proběhly pokusy o testování jednoho či dvou pravidel například v [4] nebo [35] s ručním vyhodnocením platnosti pravidla. My jsme se rozhodli validovat část pravidel sady STULONG, která se týká vzdělání. Tabulka 2. zobrazuje pravidla, která se budou validovat.

Pravidlo	
Pokud roste vzdělání, roste i	aktivita po zaměstnání
	odpovědnost v zaměstnání
	spotřeba vína
	počet kontrolních návštěv
Pokud roste vzdělání, klesá i	kouření
	aktivita v zaměstnání
	BMI
	spotřeba piva

Tabulka 2: Pravidla *background knowledge* určená k validaci

Cílem této kapitoly je prozkoumat možnosti nástroje na validaci pravidel *background knowledge* na reálném úkolu. Jako vstup bude sloužit databáze STULONG, ze které budeme používat tabulku *Entry* obsahující údaje ze vstupní prohlídky 1417 mužů, kteří se zúčastnili projektu. Další vstup představují pravidla *background knowledge* definované datovými experty projektu STULONG (Tabulka 2). Naším záměrem není diskutovat o kvalitě dat či pravidel, nýbrž pokusit se je co nejlépe ověřit za pomoci nástroje na validaci. Prostředí Ferda ještě nebylo použito pro výukové účely, a proto je řešení úkolu validace pravidel zřejmě první větší test funkčnosti celého systému na úloze přinášející konkrétní výsledky[1]. Proto jsou do výsledků testování zahrnuty i poznatky o práci s prostředím.

Vypracování úkolu probíhalo ve čtyřech fázích:

1. Tvorba atributů
2. Tvorba úloh
3. Běh úloh a interpretace výsledků
4. Závěrečné zhodnocení

7.1. Tvorba atributů

Projektu STULONG se zúčastnilo 1417 mužů, kteří se podrobili vstupní prohlídce. V průběhu prohlídky bylo zaznamenáno 244 atributů a nejzajímavější atributy (měřitelné či klasifikovatelné veličiny) jsou shrnuty v tabulce *Entry*. Tato tabulka obsahuje 64 sloupců obsahující hodnoty zajímavých atributů. Rozdělení atributů do tématických skupin zobrazuje tabulka 3.

[1] Ačkoli byl systém podroben před obhajobou rozsáhlému testování, toto testování bylo spíše nahodilé zkoušení všech funkcí systému, které nemělo za úkol něco praktického vyzkoumat.

Skupina atributů	Počet atributů
Identifikační data	2
Sociální charakteristiky	6
Fyzická aktivita	4
Kouření	3
Spotřeba alkoholu	9
Cukr, káva, čaj	3
Osobní anamnéza	18
Dotazník A_2	3
Fyzikální vyšetření	8
Biochemické vyšetření	3
Rizikové faktory	5

Tabulka 3: Rozdělení atributů tabulky *Entry* do skupin

V první fázi úkolu bylo nutné ze sloupců tabulky vytvořit atributy ve významu GUHA procedur, tedy kategorizovat domény sloupců. Jak již bylo řečeno v částech 2.4.1., 3.2.2. a 6.3. tvorba atributů je jednou z nejdůležitějších částí úlohy na dobývání znalostí a proto ji je nutné pečlivě provést.

7.1.1. Výběr vhodných sloupců

Prvním krokem bylo studium pravidel a výběr použitelných sloupců tabulky *Entry*. Oproti šabloně na pravidla *background knowledge* definované v části 3.2.2. pravidla sady STULONG neobsahují podmínku a vztah je vyjádřen rostoucí či klesající závislostí mezi pravou a levou stranou pravidla. Například pro pravidlo *„Pokud roste vzdělání, roste i kouření“* se levou a pravou stranou pravidla rozumí *„rostoucí vzdělání“* a *„rostoucí kouření“*. Je tedy potřeba identifikovat sloupce, které by se mohly použít k tvorbě atributů vyjadřujících strany pravidel.

Strana pravidla	Použité sloupce
Aktivita po zaměstnání	AKTPOZAM
Odpovědnost v zaměstnání	ZODPOV
Vzdělání	VZDELANI
Spotřeba vína	VINO, VINOMN (pro KL)
Kouření	KOURENI
BMI	Kombinace sloupců VYSKA a VAHA
Spotřeba piva	PIVOMN
Aktivita v zaměstnání	TELAKTZA

Tabulka 4: Výběr vhodných sloupců tabulky pro konstrukci atributů

Tabulka 4 udává přehled stran pravidel a sloupců, které použijeme ke kategorizaci. Z tabulky je patrné, že pro některé strany pravidel bylo potřeba vzít několik sloupců, například pro hodnoty BMI je třeba vzít sloupce VYSKA a VAHA. Pro jednu stranu pravidla, *počet kontrolních návštěv* se nepodařilo najít sloupec v tabulce, neboť takové informace tabulka *Entry* (nad níž zkoumáme) neobsahuje. Z původních osmi pravidel se počet pravidel, která lze nad tabulkou *Entry* validovat, snížil na sedm.

7.1.2. Tvorba atributů

Dalším krokem byla konstrukce atributů pro validaci pomocí KL a validačních literálů pro validaci pomocí 4FT. Použity byly pouze statické atributy a dynamické *each value one category* atributy. Důvodem je povaha dat, většinou převládají sloupce s několika různými číselnými hodnotami, přičemž každá číselná hodnota odpovídá má jistý význam (dle vysvětlivek na webových stránkách projektu[1]). Příkladem tohoto

[1] http://euromise.vse.cz/challenge2004/data/index.html

sloupce je tabulka 5, která ukazuje možné hodnoty, význam a frekvence hodnot sloupce KOURENI.

Hodnota	Význam	Počet pacientů
1	Nekuřák	385
2	Kuřák cigaret 1–4 cig./den	45
3	Kuřák cigaret 5–14 cig./den	206
4	Kuřák cigaret 15–20 cig./den	391
5	Kuřák cigaret 21 a více cig./den	348
6	Kuřák doutníků nebo dýmky	29
13	Neudáno	17

Tabulka 5: Hodnoty, význam a frekvence sloupce KOURENI

V pravidlech se vyskytují pouze dva vztahy, rostoucí a klesající závislost a pro tyto vztahy byly implementovány nové KL abstraktní kvantifikátory (viz část 5.3.5). Proto se původně předpokládalo, že pravidla se budou validovat zejména pomocí procedury KL. Povaha dat však ukázala nutnost validace i pomocí procedury 4FT. Validovat pomocí KL se hodí tehdy, jestliže sloupec obsahuje kardinální hodnoty pro vytvoření smysluplné kontingenční tabulky. Bohužel v případě databáze STULONG jsou u většiny sloupců hodnoty sice číselné, ale nejsou kardinální, neboť jsou to jenom číselníky s předdefinovaným významem. Validace pomocí 4FT tedy má svůj význam i pro úlohy na první pohled určené pro proceduru KL.

Po úvaze byly pro některé sloupce vytvořeny rozdílné atributy pro proceduru 4FT a KL. Důvodem je hlavně rozdílný pohled na kategorizaci. Rozdíl si ukážeme na sloupci KOURENI, jehož hodnoty jsou zobrazeny v tabulce 5. Jestliže chceme zkoumat rostoucí trend kouření, je určitě dobré použít atribut *each value one category*, neboť počet vykouřených cigaret stoupá s hodnotou sloupce[1]. Tento atribut je velmi výhodný do KL kontingenční tabulky. Naproti tomu jestliže zkoumáme pomocí procedury 4FT, zajímá nás rozdělení domény spíše na „kuřák vs. nekuřák“ než to, kdo kolik vykouří. Pro tento případ se hodí statický atribut s ručním přiřazením hodnot do kategorií.

Bylo vytvořeno osm statických atributů a pět atributů *each value one category*. Z atributů jsme dále snadno vytvořili osm validačních literálů, všechny s pozitivním znaménkem. Projekt *atributy.xfp* v adresáři *StulongValidace* na přiloženém CD obsahuje tyto atributy a validační literály.

V průběhu fáze tvorby atributů se ověřila funkčnost krabičky *Validační literál* a dalších krabiček pro tvorbu atributů v prostředí Ferda. Tvorba proběhla bez problému, jediné co by se dalo prostředí vytknout je poněkud nestandardní práce s modulem pro nastavení *Modul pro úpravu kategorií*, avšak i tento modul byl plně funkční.

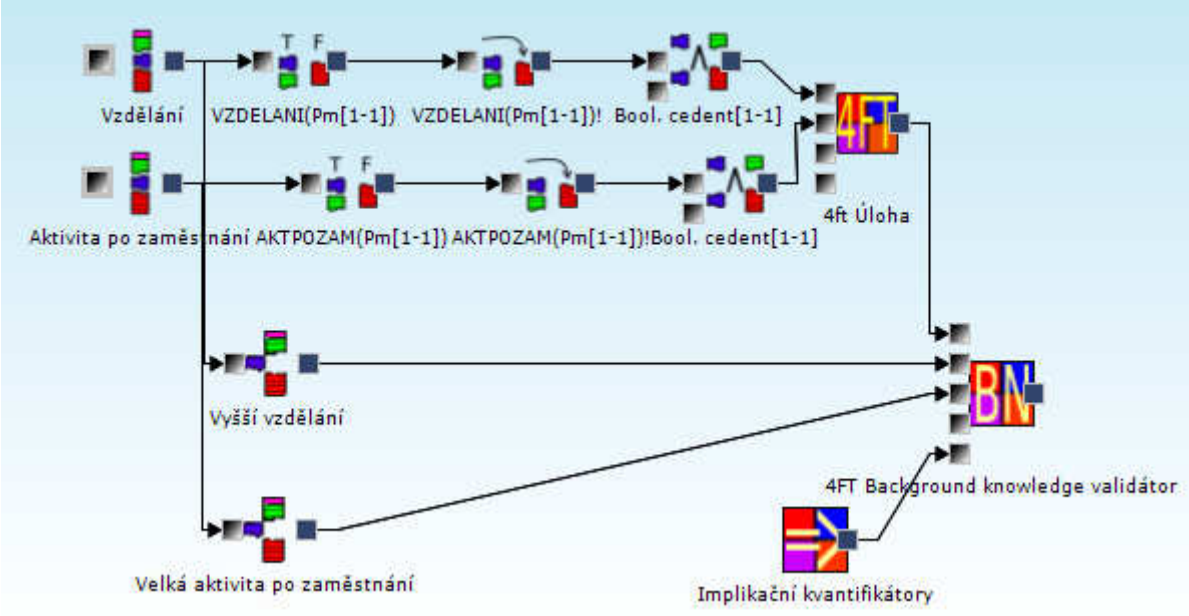
7.2. Tvorba úloh

Po konstrukci atributů jsme začali sestavovat úlohy. Vytvořili jsme tedy cedenty (kategoriální pro proceduru KL a booleovské pro proceduru 4FT) s tím, že byly ponechány všechny výchozí nastavení krabiček[2]. Dále jsme sestrojili KL a 4FT úlohy konstrukcí příslušných krabiček a zapojením cedentů do zásuvek krabiček. Abychom nemuseli používat několik krabiček KL pro validaci různých pravidel, zapojili jsme do zásuvky *Sloupcové atributy* krabičky *KL Úloha* více kategoriálních cedentů (s tím že v hypotéze bude vystupovat vždy jenom jeden). Pro krabičky úloh jsme nepoužili žádné

[1] Je to pravda za předpokladu, že kuřáci doutníků a dýmek si ničí své zdraví více než nejtěžší kuřáci cigaret a také hodnoty 13 jako X kategorie.

[2] Kategoriální cedent délky [1-1] pro KL, zadání atomů podmnožin délky 1, zadaní literálů kladného znaménka *basic* a booleovský cedent délky [1-1] pro 4FT.

kvantifikátory, neboť jsme chtěli ověřit platnost hypotéz pomocí abstraktních kvantifikátorů[1]. Jednu z úloh pro 4FT znázorňuje obrázek 18.



Obrázek 18: Úloha na validaci pomocí procedury 4FT

Úloha na obrázku zkoumá pravidlo *„Pokud roste vzdělání, roste i aktivita po zaměstnání.“* V horní části vidíme konstrukci zadání atomů, literálů a booleovských cedentů z atributů *Vzdělání* a *Aktivita po zaměstnání*. Tyto cedenty jsou zapojeny do krabičky 4FT úlohy bez kvantifikátoru. V dolní částí vidíme validační literály pro *Vyšší vzdělání* a *Velká aktivita po zaměstnání*, které jsou vytvořeny z atributů *Vzdělání* a *Aktivita po zaměstnání*. Úloha a dva validační literály jsou zapojené do krabičky validátoru spolu s abstraktním kvantifikátorem pro *Implikační kvantifikátory*. Tato a všechny další úlohy jsou uloženy v projektu *Ulohy.xfp* adresáře *StulongValidace* na přiloženém CD.

V průběhu tvorby úloh se vyskytla jedna chyba v krabičce *Validační literál*, která byla opravena. Práce s prostředím Ferda probíhala bez obtíží. Využila se zejména komponenta archivu krabiček (více v [15]), ze které byly na pracovní plochu přetahovány krabičky pro atributy a validační literály. Vertikální prohlížení archivu (filtrování krabiček na základě výběru jejich kategorie a typu) se osvědčilo, možnost horizontálního prohlížení archivu zůstala téměř nepoužita a tím se potvrdil předpoklad, že hlavní práce s krabičkami se bude dít na pracovní ploše. Zároveň by se pro projekty s větším počtem krabiček hodila možnost vyhledávání v archivu podle jména krabičky (či podle regulárního výrazu), která zatím ve Ferdovi schází. Projekt obsahoval po vytvoření úloh více než 80 krabiček a práce s prostředím se z pohledu uživatele nezpomalila, což znamená, že pro projekty podobného rozsahu není třeba větší optimalizace prostředí.

7.3. Běh úloh a interpretace výsledků

Po sestavení úloh jsme spustili jednotlivé GUHA procedury a posléze validovali pomocí validačních krabiček. Tabulka 6 ukazuje výsledky validace jednotlivých pravidel pomocí procedur 4FT a KL.

[1] A využít tak jednu z předností nástroje na validaci, tj. že uživatel nemusí příliš rozumět normálním kvantifikátorům, když používá abstraktní (popsáno v části 5.3.1).

Pravidlo		Platné pomocí KL	Platné pomocí 4FT
Pokud roste vzdělání, roste i	aktivita po zaměstnání	ANO	NE
	odpovědnost v zaměstnání	NE	NE
	spotřeba vína	NE	NE
	počet kontrolních návštěv	NE	NE
Pokud roste vzdělání, klesá i	Kouření	NE	NE
	aktivita v zaměstnání	NE	NE
	BMI	chyba	Chyba
	spotřeba piva	NE	Chyba

Tabulka 6: Výsledky validace pravidel

Výsledky lze shrnout tak, že se platnost pravidel v datech nepotvrdila. Rozeberme nyní podrobněji jednotlivé možnosti, které nastaly. Zajímavý výsledek je potvrzení pravidla *„Pokud roste vzdělání, roste i aktivita po zaměstnání“* pomocí procedury KL, ale vyvrácení pomocí procedury 4FT. Neboť jsme použili pro obě úlohy stejné atributy, lze vyloučit příčinu tohoto jevu ve nesprávné kategorizaci. Vysvětlení musíme hledat v použití abstraktních kvantifikátorů. Pro *implikační kvantifikátory* je výchozí hodnota pro parametr *base* 0,05[1], což je v tomto případě splněno (hodnota kvantifikátoru *base* je 0,5). Problém je tedy v příliš vysoké hodnotě parametru *p* (ten je dle tabulky pro všechny *implikační kvantifikátory* 0,95). Lze tedy říci, že *implikační kvantifikátory* hledají jenom „extrémně“ silné závislosti mezi antecedentem a sukcedentem a ty se obvykle v datech nevyskytují (alespoň se to zdá z našeho příkladu), i když jistá závislost může existovat (z pohledu KL).

Řešením je buď zmenšit hodnoty parametru *p* pro *implikační kvantifikátory* na přijatelnější hodnotu, anebo použít jinou třídu kvantifikátorů. Zajímavý a často používaný je například *Above average* kvantifikátor, který patří pro tuto práci do třídy *ostatní kvantifikátory*. Při zkušebním testování funkčnosti krabiček také byly některé pravidla validované pomocí *Fischerovských kvantifikátorů*, zatímco pomocí *implikačních kvantifikátorů* ne.

U tří validaci nastala chyba programu. Všechny tyto chyby nastaly při spouštění úlohy a žádná při validaci pravidel. Je to zřejmě způsobeno nedostatečnou odladěností krabiček implementujících GUHA procedury. Protože však vzniká v současné době nová implementace GUHA procedur v prostředí Ferda nezávislá na modulech systému LISp-Miner(v práci [17]), není důležité opravit tyto chyby ve staré verzi, ale spíše zajistit, aby se chyby nad stejným zadáním v nové verzi nevyskytly.

Co se týče použití nástroje na validaci, v této fázi nebyly nalezeny chyby a validace proběhla bez problému. Bohužel při používání prostředí Ferda a konkrétně krabiček pro GUHA procedury implementované v něm, vyskytly se chyby při běhu některých úloh. Jinak bylo prostředí funkční a dobře se s ním pracovalo.

7.4. Závěrečné zhodnocení

V části 5.3.1 byla diskutována užitečnost a použitelnost nástroje na validaci pravidel *background knowledge*. Testování na úkolu validace pravidel STULONG vypracovaný v této kapitole ukázalo, že nástroj je nejenom funkční a použitelný, ale také užitečný. Nástroj představuje pro uživatele jednoduchý způsob, jak sestavovat a validovat pravidla *background knowledge*. Nástroj je však užitečný také v tom, že nás nutí přemýšlet o praktickém významu jednotlivých kvantifikátorů a jejich výchozích hodnot.

Zjistili jsme například, že *implikační kvantifikátory* pro proceduru 4FT nepřináší mnoho pozitivních výsledků a že je třeba změnit výchozí hodnoty těchto kvantifikátorů

[1] Tedy hypotéza musí platit v alespoň pěti procentech případů.

anebo použít kvantifikátory jiné. Přitom kvantifikátor *fundované implikace* je jednoznačně nejčastěji používaný kvantifikátor v cele historii procedury. Testování ukázalo nutnost získání hlubších poznatků o kvantifikátorech pro pozdější automatické použití s výchozími hodnotami, například v systému Ever-Miner. K získání poznatků o kvantifikátorech se jeví výhodné používat nástroj na validaci pravidel *field knowledge* implementovaný v rámci této práce.

8. Závěr

Cílem diplomové práce bylo prozkoumat použití doménových znalostí při dobývání znalostí pomocí GUHA procedur. Těmito znalostmi rozumíme znalosti o oblasti, které se týkají analyzovaná data. Práce se zabývá dvěma druhy doménových znalostí: *background knowledge* a ontologiemi a klade si za úkol prozkoumat tyto znalosti vzhledem k použití v sestavování úloh a interpretaci výsledků GUHA procedur. Jestliže se v průběhu práce narazí na výsledky vhodné k implementaci, mají být nástroje vzešlé z těchto výsledků implementovány v prostředí Ferda.

8.1. Shrnutí vykonané práce

V rámci diplomové práce se autor seznámil s teoretickými poznatky potřebnými k porozumění problematice. Nejdříve byly nastudovány principy metody GUHA. Dále autor pokračoval studiem systému LISp-Miner a všech GUHA procedur implementovaných v tomto systému. Jelikož autor je jedním z tvůrců prostředí Ferda, nebylo potřeba blíže si osvojovat jeho funkčnost, až na principy přidání nových modulů (krabiček) do systému. S těmito principy autor neměl zkušenosti a jejich osvojení bylo nezbytné pro implementační část práce, jelikož tato část spočívala v implementaci nových modulů do prostředí.

Poté autor přistoupil ke studiu doménových znalostí. Zatímco u ontologií existuje množství literatury, které pojednává o teoretických základech a principech správného návrhu ontologií, u *background knowledge* nebyla k dispozici téměř žádná literatura. Začalo se tedy se studiem ontologií a osvojením znalosti o základech, rozdělení, tvorbě a implementaci tohoto druhu doménových znalostí. Dále byly prostudovány dostupné zdroje zkoumající využití ontologií pro dobývání znalostí (o tom se čtenář může dočíst v části 4.1.).

I když se ontologie zdály být vhodnou formou doménových znalostí pro tuto práci, autor pokračoval ve studiu druhého uvažovaného typu doménových znalostí, *background knowledge*. Pro tento typ neexistovala téměř žádná literatura a pouze malé zmínky o tom, co tento pojem vlastně znamená. Při hledání literatury autor narazil na systém Ever-Miner, nástroj na dobývání znalostí příští generace, o kterém se delší dobu uvažuje na VŠE. Při přemýšlení o něm v souvislosti s prostředím Ferda byl vymyšlen návrh architektury systému Ever-Miner založené na prostředí Ferda. Toto je první praktický výsledek práce a čtenář se o něm může dočíst v části 2.4.1.

Po seznámení s *background knowledge* se zjistilo, že tento typ znalostí, i když není probádaný v souvislosti s metodou GUHA, je velmi perspektivní, zejména co se týče úloh na ověřování známých vztahů v datech. Pokračovalo se dál ve zkoumání tohoto typu hledáním vhodné formalizace slovních pravidel. Z literatury se autor dozvěděl o formalizaci pomocí kvalitativních modelů, avšak tato formalizace byla shledána jako nepostačující pro účel práce (část 3.2.1.) Byla tedy vymyšlena formalizaci vlastní, založená na principech dobývání znalostí pomocí metody GUHA. Tato formalizace byla nazvaná *formalizace pomocí atributů, validačních literálů a abstraktních kvantifikátorů* a pojednává o ní část 3.2.2.

Se zavedenou formalizací začalo být zřejmé, že případná implementace validace pravidel *background knowledge* bude pro uživatele užitečnější než jakákoli realizovatelná implementace využití ontologií. Důvody jsou shrnuty v části 5.1. Proto autor vymyslel algoritmy na validaci pravidel a celkem 10 krabiček, které budou validaci realizovat v prostředí Ferda. Byly vymyšleny také dva nové abstraktní kvantifikátory pro proceduru KL, *rostoucí závislost* a *klesající závislost*, tak jak jsou definovány v části 5.3.5. Všechny krabičky byly implementovány.

S nově vyvinutým nástrojem vyvstala potřeba jeho testování na reálném úkolu. Proto bylo provedeno testování, které mělo za úkol ověřit platnost pravidel z datového zdroje STULONG, jak je popsáno v sedmé kapitole. Testování proběhlo úspěšně a potvrdilo funkčnost a použitelnost nástroje na validaci pravidel i práci s prostředím Ferda. Současně však poukázalo na možný problém praktické použitelnosti jednotlivých kvantifikátorů a interpretace jejich výsledků.

Co se týče ontologií, v práci jsou důsledně rozpracovány všechny postupy využití ontologií při dobývání znalostí zmíněné v předešlé literatuře a přidány některé nové (o tom pojednává 6. kapitola). U všech postupů byla posuzována jejich vhodnost pro pozdější implementaci a u perspektivních postupů byl vypracován podrobnější návrh implementace v prostředí Ferda. Tím byly podstatně rozšířeny stávající poznatky o využití ontologií při dobývání znalostí pomocí GUHA procedur. Ukázalo se, že nejvýhodnější pro uživatele i z hlediska možného pozdějšího použití v systému Ever-Miner je automatická tvorba atributů pomocí ontologií, i když tento postup není pro svou rozsáhlost a složitost rozmyšlen do té míry, aby byl implementován.

8.2. Náměty na další práci

Práce udává tři hlavní směry, kudy by se měl ubírat další vývoj v oblasti použití doménových znalostí pro dobývání znalostí pomocí GUHA procedur. První směr je prohloubení dosavadních znalostí o validaci pravidel *background knowledge*, zejména co se týče použitelnosti a významu jednotlivých abstraktních kvantifikátorů. Jak bylo zmíněno v části 7.4, některé často používané kvantifikátory se ukázaly pro validaci jako nevhodné. Je tedy třeba testovat použití různých kvantifikátorů na různorodých datech a nalézt jejich nastavení vedoucí k optimálním výsledkům, popřípadě vymyslet kvantifikátory nové. Důležitá je také úzká spolupráce s datovými experty a integrace jejich pohledu na data i na pravidla do výsledků příštích prací.

Druhým směrem vývoje je implementace použití ontologií. V práci byly navrženy konkrétní krabičky či algoritmy různě využívající ontologie. Bohužel zde nezbyl prostor pro implementaci těchto postupů. Navazující práce by měly tyto postupy implementovat. Důležité je zejména vytvoření krabiček *Ontologie* a *Mapování* (rozpracované v části 6.1), neboť slouží jako základ pro všechny další postupy pro využití ontologií. Z dlouhodobého hlediska se jeví jako nejperspektivnější zkoumání automatické tvorby atributů a vytvoření vhodných ontologií pro tuto tvorbu.

Třetí směr vývoje, který je současně největší výzvou pro studenty i vědecké pracovníky zabývající se dobýváním znalostí pomocí metody GUHA, je postupné vytváření systému Ever-Miner. O tomto systému všeobecně platí, že existuje mnoho návrhů, jak by měl systém fungovat, ale téměř žádné praktické výstupy z těchto návrhů. V práci byl představen návrh implementace systému v prostředí Ferda. Tento návrh by mohl být základním stavebním kamenem pro budoucí složitý systém. Je třeba zejména implementovat vrstvu zajišťující automatickou tvorbu atributů a úloh. Vývoj systému Ever-Miner v sobě také částečně zahrnuje předchozí dva směry vývoje, neboť výsledky práce na prohloubení znalosti validace mohou být použity pro řešení některých typových úloh pro systém Ever-Miner a práce na využití ontologií ke tvorbě atributů přímo přispívá k budoucí automatické tvorbě atributů.

Reference

(1) Clark P., Matwin S.: Using Qualitative Models to Guide Inductive Learning, Proceedings 10th International Machine Learning Conference (ML93), str. 49-56

(2) Clementine Data Mining Systém, http://www.spss.com/clementine

(3) Cross Industry Standard Process for Data Mining: http://crisp-dm.org

(4) Češpivová H., Rauch J., Svátek V., Kejkula M., Tomečková M.: Roles of Medical Ontology in Association Mining CRISP-DM Cycle, ECML/PKDD04 Workshop on Knowledge Discovery and Ontologies (KDO'04), Pisa 2004

(5) Češpivová Hana: Tvorba ontologií pro dobývání znalostí z databází, *Diplomová práce*, Fakulta informatiky a statistiky VŠE, Praha 2004

(6) DAML Data Sources, http://www.daml.org/data/

(7) Databáze Barbora o úvěrech fiktivní banky, ke stažení na http://ferda.sourceforge.net

(8) EUROMISE: Projekt STULONG . http://euromise.vse.cz

(9) Gomez-Perez A., Fernandez-Lopez M., Corcho O.: Ontological Engineering: with the examples from the areas of Knowledge Management, e-Commerce and the Semantic Web, Springer-Verlag London Berlin Heidelberg, ISBN 1-85233-551-3

(10) Gruber T.: It Is What It Does: The Pragmatics of Ontology as Language, Contract, and Content, http://www.cs.man.ac.uk/~stevensr/workshop/gruber.zip

(11) Gruber T.: What is an ontology?, http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

(12) Hájek P., Havránek T.: Mechanising Hypothesis Formation – Mathematical Foundations for a General Theory, Springer, Berlin Heidelberg, New York 1978

(13) Karban T.: Data Mining in Relational Databases, *Disertační práce*, Katedra softwarového inženýrství, Matematicko-fyzikální fakulta, Univerzita Karlova, rukopis

(14) Kejkula Martin: Self-Organized Data Mining – 20 Years after GUHA-80, prezentace na semináři KEG, http://gama.vse.cz/keg/seminar/keg-sem.html

(15) Kováč M., Kuchař T., Kuzmin. A, Ralbovský M.: Ferda, nové vizuální prostředí pro dobývání znalostí, Znalosti 2006, Sborník příspěvků 5. ročníku konference, Hradec Králové 2006, ISBN 80-248-1001-8

(16) Kováč M.: Uživatelsky orientovaný jazyk pro řešení úloh DZD, *Diplomová práce*, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha 2006

(17) Kuchař T.: Experimentální GUHA procedury, *Diplomová práce*, Matematicko-fyzikální fakulta, Univerzita Karlova 2006

(18) Labský M., Svátek V.: On the Design and Exploitation of Presentation Ontologies for Information Extraction, 3rd European Semantic Web Conference 2006, workshop 4, sborník

(19) Lín V., Rauch J., Svátek V.: Content-based Retrieval of Analytic Reports. In: Schroeder, M., Wagner G. (eds.). Rule Markup Languages for Business Rules on the Semantic Web, Sardinia 2002, 219-224

(20) Lín V.: 053 Definice KL-kvantifikátorů, dokumentace systému LISp-Miner

(21) Matwin S., Rouge T.: Explainable Induction with an Imperfect Qualitative Model, http://citeseer.ist.psu.edu/matwin95explainable.html.

(22) OWL Web Ontology Language Overview, http://www.w3.org/TR/owl-features/

(23) Projekt LM – Report Asistent, http://reportasistent.berlios.de/

(24) Rauch J., Šimůnek M., An Alternative Approach to Mining Association Rules, Lin T. Y., Ohsuga S., Liau C. J., Tsumoto S. (eds.) Data Mining: Foundations, Methods, and Applications, Springer-Verlag 2005

(25) Rauch J., Šimůnek M., Lín V.: Mining for Patterns Based on Contingency Tables by KL-Miner First Experience, ICDM2003 WORKSHOP *Foundations and New Directions of Data Mining*, http://www.cs.uvm.edu/~xwu/icdm-03.html

(26) Rauch J., Šimůnek M.: GUHA Metod and Granular Computing, Proceedings of IEEE International Conference on Granular Computing 2005, http://cs.sjsu.edu/~grc/grc2005/index.html

(27) Rauch J., Šimůnek M.: Projekt LISp-Miner, současný stav a další rozvoj, interní dokument projektu LISp-Miner

(28) Rauch J.: 032 Zadání pro KL-Miner, dokumentace systému LISp-Miner

(29) Rauch J.: EverMiner – Architektura, interní dokument projektu LISp-Miner

(30) Rauch J.: EverMiner – studie projektu, interní dokument projektu LISp-Miner

(31) Rauch J.: Studijní materiály ke kurzu GUHA Method, Association Rules and Knowledge Extraction, Laboratoire ERIC, Universite Lumiere Lyon 2, December 2005 and January 2006

(32) Schutte S.: Automating the 4FT-Miner Set-Up: Two Example Approaches from the Toolbox of AI, prezentace na semináři KEG, http://gama.vse.cz/keg/seminar/keg-sem.html

(33) Schutte S.: First considerations for applying AI techniques in LISp-Miner extension, Znalosti 2006, Poster na konferenci, Hradec Králové 2006

(34) Stránka projektu Ferda, http://ferda.sourceforge.net

(35) Svátek V., Rauch. J., Ralbovský M.: Ontology-Enhanced Association Mining. In: Ackermann, Berendt (eds.). Semantics, Web and Mining, Springer-Verlag, To appear

(36) Svátek, V.: Ontologie a WWW. Tutoriál ke konferenci Datakon 2002. http://nb.vse.cz/~svatek/temata.htm

(37) Systém LISp-Miner: http://lispminer.vse.cz

(38) The Protégé Ontology Editor, http://protege.stanford.edu/

(39) Unified Medical Language System, http://www.nlm.nih.gov/research/umls

(40) ZeroC, http://www.zeroc.com, Internet Communications Engine (Ice)

Dodatek A: Background knowledge

Dodatek obsahuje všechny *background knowledge* shromážděné v průběhu vypracování diplomové práce. Pravidla jsou tématicky rozdělena podle domén.

Zdravotnictví, databáze STULONG

Pokud roste krevní tlak, roste i	riziko onemocnění. riziko cévní mozkové příhody.
Pokud roste věk, roste i	počet onemocnění.
Pokud roste věk, klesá	spotřeba kávy. aktivita po zaměstnání. spotřeba alkoholu.
Pokud roste vzdělání, roste i	aktivita po zaměstnání. odpovědnost v zaměstnání. spotřeba vína. počet kontrolních návštěv.
Pokud roste vzdělání, klesá	kouření. aktivita v zaměstnání. BMI. spotřeba piva.
Pacienti s vyšší odpovědností v zaměstnání	častěji používají k cestě do zaměstnání auto.
Pokud roste odpovědnost v zaměstnání, roste i	počet kontrolních návštěv.
Pacienti s nižší odpovědností v zaměstnání	používají k cestě do zaměstnání veřejnou dopravu.
Pokud se zvyšuje kouření, zvyšuje se i onemocnění.	počet kardiovaskulárních počet nádorových onemocnění. BMI (v průběhu 3–5 návštěv).
Pokud se snižuje kouření, sníží se i	BMI (v průběhu 3–5 návštěv).
Pokud roste spotřeba piva, roste i	BMI.
Pokud rostou tělesné aktivity po zaměstnání,	roste i HDL cholesterol. klesá BMI (v průběhu 3–5 návštěv). klesá krevní tlak (na hodnotu < 120/80).
Pokud klesají tělesné aktivity po zaměstnání,	roste BMI (v průběhu 3–5 návštěv). roste cholesterol.

Pokud klesá spotřeba tuků, potom	klesá i cholesterol (v průběhu 3–5 návštěv).
Pokud roste BMI, roste i	hypertenze (hranicí je krevní tlak > 140/90), dá se prokázat zejména při více návštěvách (více než 10), 15., 20. rok studie. glykémie.
Pokud roste úroveň cholesterolu, roste i	úroveň triglyceridů.
Odchod do plného důchodu znamená, že	v následujících pěti letech roste BMI. rostou tělesné aktivity ve volném čase.
Pokud roste počet kontrolních návštěv, pak klesá	krevní tlak (na hodnotu < 140/90). cholesterol (na hodnotu < 6,0 mm/l). počet kuřáků. průměrný počet vykouřených cigaret.

Prodej piva – pravidla poskytl Vratislav Beneš

Pokud roste vzdálenost od komína, pak klesá	prodej piva.
Pokud roste teplota, pak roste	prodej méně alkoholických piv.
Čím menší pivovar	tím jsou lidé v jeho okolí větší patrioti na svoji značku.
Čím větší značka piva,	tím je prodej více ovlivněn reklamou.
Čím dražší značka piva, Čím levnější značka piva,	tím více jsou lidé věrní značce. tím jsou lidé více přelétaví v souvislosti s cenou.

Dražší značky piva se prodávají více na svátky (vánoce).

Levnější značky piva se prodávají v ekonomicky slabších regionech.

Mladší lidé preferují čepované pivo.
Starší lidé preferují lahvové pivo.

Dodatek B: Systém LISp-Miner

článek zveřejněný v Znalosti 2003, sborník příspěvků 2. ročníku konference, Ostrava 2003, ISBN 80-248-0229-5

Systém LISp-Miner

Jan Rauch[1], Milan Šimůnek[1,2]

[1] Fakulta informatiky a statistiky, VŠE Praha – nám W. Churchilla 4, Praha 3
Ústav informatiky AV ČR

Abstrakt. Je prezentován akademický projekt zaměřený na výuku a výzkum v oblasti dobývání znalostí z databází. Projektový tým se skládá z učitelů i studentů z více vysokých škol, navazována je i spolupráce se zahraničím. V rámci projektu je vyvíjen otevřený a volně dostupný softwarový systém LISp-Miner. Jedná se o modulární systém, který může být snadno rozšiřován pomocí dalších modulů zaměřených na specifický způsob zpracování analyzovaných dat. Při tvorbě nových modulů je možno využívat rozsáhlé prostředky vytvořené při implementaci existujících modulů. Je možné vytvářet i specializovaná webová rozhraní uzpůsobená potřebám konkrétní analýzy. Přizpůsobení se může týkat jak rozsahu funkčnosti, tak použité terminologie. V článku jsou popsány některé existující moduly a zároveň jsou zmíněny i současné směry rozvoje systému (např. multirelační analýza).

Klíčová slova: dobývání znalostí z databází, asociační pravidal, data mining

1. Úvod

Automatizované zpracování provozních dat se v současné době stalo standardem téměř ve všech oblastech podnikatelské činnosti i veřejné správy. Vznikají tak rozsáhlé a složitě strukturované databáze zaznamenávající někdy i desítky let dlouhou historii činnosti organizací jako jsou banky, pojišťovny, výrobní podniky, obchodní řetězce ale i nemocnice, školy atd. Ve většině případů lze z těchto databází získat podstatně více informací a znalostí než bylo původně předpokládáno.

Proces, jehož cílem je získání takovýchto nových znalostí se v angličtině nazývá *Knowledge discovery in databases*, často se používá i termín *data mining*. Podle [Hn 01] je data mining *analýza (často rozsáhlých) observačních dat s cílem nalézt netušené vztahy a sumarizovat data novými způsoby tak že jsou srozumitelná a užitečná pro jejich majitele*. Pro tento proces budeme používat termín *dobývání znalostí z databází*, (zkráceně DZD) slovo dobývání vyjadřuje fakt že získání nových znalostí vyžaduje vynaložení rozsáhlého a nelehkého úsilí.

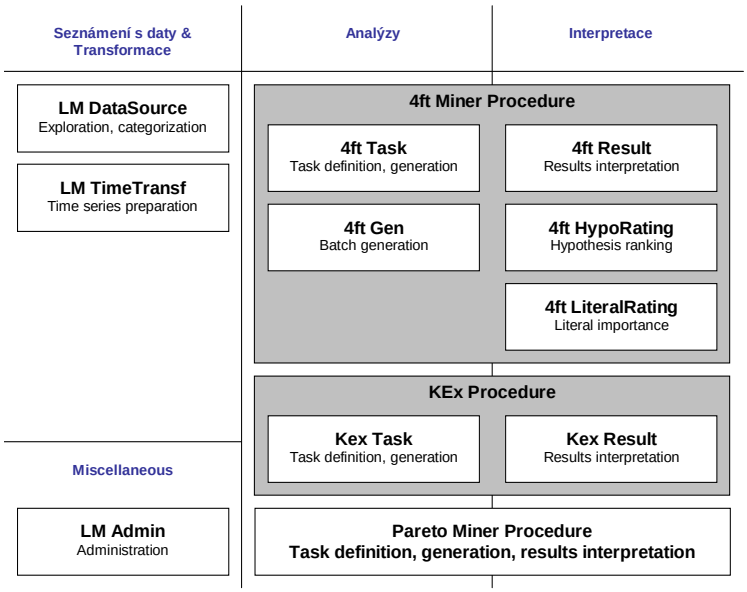
Dobývání znalostí z databází je předmětem jak rozsáhlého výzkumu a vývoje tak i mnoha obchodních aktivit. Je k dispozici řada softwarových produktů a metodologií pro DZD. Charakteristické je, že komerční systémy jsou značně drahé a těžko dostupné pro výuku. Charakteristické je i to výzkum a vývoj v oblasti DZD je stále spíše na počátku. Cíle článku je popsat současný stav akademického softwarového systému LISp-Miner určeného pro podporu výuky a výzkumu v oblasti DZD.

Systém LISp-Miner je vyvíjen na Fakultě informatiky a statistiky VŠE od roku 1996. Hlavním cílem je přiblížit studentům celý proces DZD, umožnit jim práci na vlastních analýzách a poskytnout i detailnější pohled na algoritmy a techniky použité při implementaci takového systému. Zároveň systém slouží jako platforma pro další výzkum v oblasti DZD a to jak na teoretické rovině, tak i prostřednictvím tvorby nových modulů, zejména v rámci diplomových a disertačních prací. Nezanedbatelným přínosem je i možnost využít systém LISp-Miner pro DZD v oblastech ve kterých nejsou dostupné komerční systémy (např. při zpracování medicínských dat). LISp-Miner je volně ke stažení na adrese http://lispminer.vse.cz.

Hlavní rysy systému jsou spolu s přehledem jeho částí popsány v odstavci 2. Podrobnosti o dvou z nich jsou v odstavcích 3 a 4. Další vývoj systému je naznačen v odstavci 5

2. Hlavní rysy systému LISp-Miner

LISp-Miner je modulární systém, přehled současných modulů je v obr. 1.



Obr. 1 – Přehled současných modulů systému LISp-Miner

Moduly rozdělujeme do třech základních skupin:

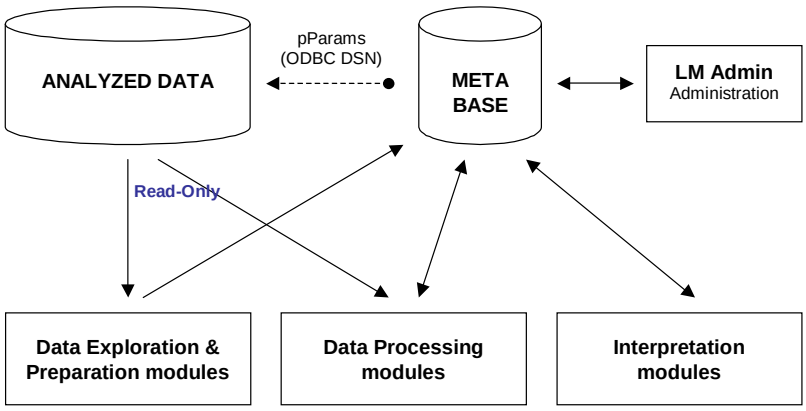
- Seznámení s daty a jejich transformace,
- Zpracování analýz
- Interpretace výsledků

Toto rozdělení umožňuje udržet složitost každého modulu na úrovní přijatelné pro uživatele a zároveň pomáhá dodržet metodikami doporučovaný postup DZD (příkladem je metodologie CRISP-DM).

Pro rychlé seznámení s daty slouží modul LM DataSource, který umožňuje frekvenční analýzy a širokou škálu transformací dat, podrobnosti jsou v odstavci 3. Pro tyto účely je určen i modul LM TimeTransf [Sl 02] určený pro výpočet různých charakteristik časových řad tak, aby vypočítané charakteristiky mohly být snadno využity v dalších částech systému LISp-Miner.

Nejvíce používanou částí systému LISp-Miner je procedura 4ft-Miner v hrubých rysech popsaná v odstavci 4. Jedná se o novou, oproti dřívějším značně rozšířenou implementaci GUHA procedury ASSOC [Ha 83]. Při implementaci procedury 4ft-Miner byly jednak využity dřívější zkušenosti [Ra 78, Ra 81] a jednak vyvinuty prostředky, které usnadnily i novou implementaci procedury KEX [BI 94] pro strojové učení. Procedura KEX je také součástí systému LISp-Miner, její podrobnější popis přesahuje rámec tohoto příspěvku. Bude jí věnována samostatná publikace. Vyvinuté prostředky významným způsobem usnadní i implementaci dalších analytických procedur. Několik poznámek v tomto směru je v odstavci 5.

Důležitou součástí systému je metabáze. Jedná se o specializovanou databázi která se vytváří pro každou aplikaci. Aplikací rozumíme řadu (obvykle desítky až stovky) běhů modulů LM DataSource, LM TimeTransf a procedur 4ft-Miner nebo KEX. V metabázi se uchovávají nezbytné informace o analyzované databázi (např. seznam realčních tabulek), definice transformací dat prováděných moduly LM DataSource a LM TimeTransf a zadání a výsledky procedur 4ft-Miner a KEX. Každý modul tedy načítá své vstupy z metabáze kam ukládá i své výstupy. Každý modul může (ale nemusí) pracovat i přímo s analyzovanou databází. , viz též obr 2.



Obr. 2 – Globální architektura systému LISp-Miner

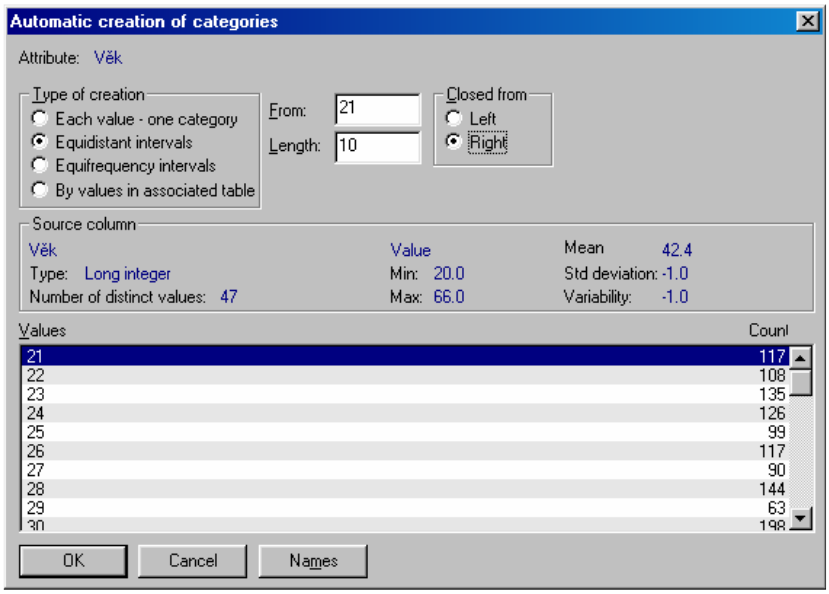
Uvedená architektura umožňuje snadno přidávat další moduly. Ty mohou využívat výstupy ostatních modulů jako své vstupy a své výstupy naopak zpřístupňovat jiným modulům pro další zpracování. Nový modul může být jak samostatnou aplikací, tak i webovým rozhraním, které spouští vybrané moduly v dávkovém režimu. Tak je možné velmi rychle vytvořit rozhraní specializovaná pro určitou oblast a konkrétní analýzu. V nich mohou být použity odborné termíny z analyzované oblasti a ovládání může být zjednodušené, aby s ním mohl snadno pracovat i samotný odborník na danou oblast. Takto lze rozšířit okruh lidí schopných provádět kvalifikované a hluboké analýzy dat, zejména pokud se jedná o opakované analýzy nad stále aktualizovanou databází. Rozhraní může být realizováno prostřednictvím WWW technologií. Jedna z takových implementací je zmíněna v [ST 02].

3. LM DataSource

Modul LM DataSource slouží pro seznámení se s daty a zejména pro jejich přípravu pro další zpracování. V rámci seznámení je možné procházet seznam tabulek v analyzované databázi, seznam sloupců v každé z nich a také data, která obsahují.

V rámci přípravy dat je možné definovat odvozené hodnoty z existujících sloupců tabulek. Pomocí databázových funkcí je možné dopočítat věk pacienta z jeho datumu narození, celkovou výši platu jako součet základu a prémií nebo zjistit den v týdnu podle datumu uskutečnění telefonního hovoru. Přestože se jedná o odvozené informace, jejich použití může výrazně zlepšit výsledky analýzy.

Během přípravy dat je však klíčovou fází vhodná kategorizace hodnot. Může se jednat o vytvoření intervalů ze spojité veličin, o seskupení okrajových hodnot do jedné kategorie, sloučení geograficky blízkých okresů do vyšších celků atd. nebo i kategorizaci spojité veličiny do intervalů. V obr. 3. je ukázka dialogového okna kterým se zahajuje kategorizace sloupce Věk.

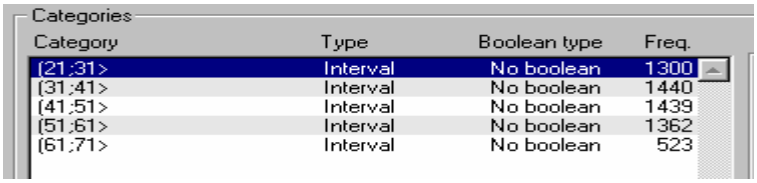


Obr. 3 – Ukázka okna pro kategoritzaci

V současné době jsou k dispozici tyto volby pro tvorbu kategorií:

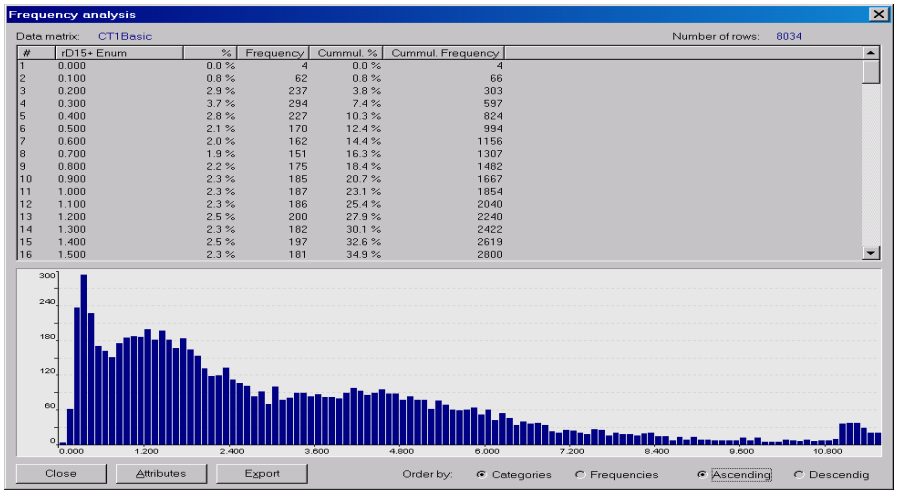
- **Enumerace** … každá hodnota v analyzované relační tabulce je samostatnou kategorií,
- **Ekvidistantní intervaly** … systém automaticky vytvoří intervaly o stejné délce s daným počátkem a s daným způsobem uzavření (zleva, zprava)
- **Ekvifrekvenční intervaly** … systém automaticky vytvoří zadaný počet intervalů tak, že do každého intervalu spadá pokud možno stejný počet záznamů z analyzované tabulky.

V obr. 3 je požadováno vytvoření intervalů o délce 10 uzavřených zprava. Ukázka výsledku je v obr. 4.



Obr. 4 - Ukázka intervalů vytvořených dle zadání z obr. 3.

Důležitou funkcí modulu LM DataSource je i frekvenční analýza, která napomáhá vhodné kategorizaci hodnot. Ukázka frekvenční analýzy je v obr. 5. Kategorie v seznamu i v grafu je možné řadit vzestupně/sestupně podle frekvence a seznam je možné exportovat do schránky systému Windows a použít v dalších aplikacích. Analyzujeme-li najednou více atributů, jsou v seznamu i grafu uvedeny všechny možné kombinace kategorií ze všech analyzovaných atributů.



Obr. 5 – Ukázka frekvenční analýzy pomocí modulu LM DataSource

4. Procedura 4ft-Miner

Procedura hledá zajímavé vztahy mezi různými, automaticky odvozovanými booleoský atributy. Hledané vztahy jsou ve formě značně zobecněných asociačních pravidel. Vstupem procedury 4ft-Miner je jednak matice dat která je výstupem z modulu LM DataSource a jednak relativně jednoduché zadání rozsáhlé množiny asociačních pravidel. Výstupem procedury jsou všechna asociační pravidla pravdivá v analyzované matici dat.

Asociační pravidlo pro matici dat M je výraz $\mathrm{ANT} \approx \mathrm{SUC}$, procedura 4ft-Miner pracuje i s podmíněnými asociačními pravidly $\mathrm{ANT} \approx \mathrm{SUC} / \mathrm{COND}$. Zde ANT, SUC a COND jsou booleovské atributy definované na matici M, ANT se nazývá *antecedent* , SUC sukcedent a COND je podmínka. Asociační pravidlo $\mathrm{ANT} \approx \mathrm{SUC}$ říká, že ANT a SUC jsou v relaci dané symbolem ≈. Tento symbol se nazývá 4ft-kvantifikátor.

Asociační pravidlo $\mathrm{ANT} \approx \mathrm{SUC}$ je pravdivé v matici dat M jestliže je pro čtyřpolní tabulku 4ft(ANT, SUC, M) splněna podmínka daná 4ft-kvantifikátorem ≈. Tabulka 4ft(ANT, SUC, M) pro ANT a SUC v M je čtveřice čísel dle Tab. 1.

M	SUC	¬SUC
ANT	*a*	*b*
¬ANT	*c*	*d*

Tab 1. – Čtyřpolní tabulka 4ft(ANT, SUC, M) pro ANT a SUC v M

Zde *a* je počet řádků matice M splňujících jak ANT tak SUC, *b* je počet řádků splňujících ANT a nesplňujících SUC, *c* je počet řádků nesplňujících ANT a splňujících SUC a *d* je počet řádků nesplňujících ani ANT ani SUC

Podmíněné asociační pravidlo $\mathrm{ANT} \approx \mathrm{SUC} / \mathrm{COND}$ je pravdivé jestliže pravidlo $\mathrm{ANT} \approx \mathrm{SUC}$ je pravdivé v matici M / COND. Matice $\mathcal{M}$ / COND se skládá ze všech řádků matice M splňujících COND.

Uvádíme několik příkladů 4ft-kvantifikátorů:

- Kvantifikátor $\Rightarrow_{p;\mathrm{Base}}$ **fundované implikace** s parametry $0 < p \leq 1$ a $Base > 0$, kterému odpovídá podmínka $\frac{a}{a+b} \geq p \wedge a \geq Base$. Asociační pravidlo $\mathrm{ANT} \Rightarrow_{p;Base} \mathrm{SUC}$ může být interpretováno tak, že „*100*p procent objektů splňujících* ANT *splňuje zároveň i* SUC“ nebo „z ANT *vyplývá* SUC s *pravděpodobností 100p procent*“.

- Kvantifikátor $\Leftrightarrow_{p;Base}$ **dvojité fundované implikace** s parametry $0 < p \leq 1$ a $Base > 0$, kterému je přiřazena podmínka $\frac{a}{a+b+c} \geq p \wedge a \geq Base$. Asociační pravidlo ANT $\Leftrightarrow_{p;Base}$ SUC může být interpretováno jako "*100p procent objektů splňujících* φ *nebo* SUC splňuje jak φ *tak* SUC" nebo "ANT ∨ SUC *implkuje* ANT ∧ SUC *na úrovni 100p procent*".

Procedura 4ft-Miner zahrnuje celkem 17 typů 4ft-kvantifikátorů, včetněn řady statistických podrobně popsaných na domovské stránce systému LISp-Miner na adrese http://lispminer.vse.cz\overview\4ft_quantifier.html.

Příkladem asociačního pravidla je výraz

Pohlaví(Muž) ∧ Věk(21-30) ∧ Bydliště(Praha, Brno) $\Rightarrow_{0.9;50}$ Půjčka(splácí).

Toto asociační pravidlo je v matici dat týkajících se půjček klientů banky pravdivé, jestliže nejméně 90% z maticí popsaných klientů – mužů ve věku 21-30 bydlících v Praze nebo Brně splácí svou půjčku a jestliže takových klientů je nejméně 50. Výraz Bydliště(Praha, Brno) je booleovský atribut - literál vytvořený z atributu Bydliště pomocí koeficientu Praha, Brno. Obecně platí, že antecedent, sukcedent i podmínka asociačního pravidla jsou konjunkce literálů nebo jejich negací.

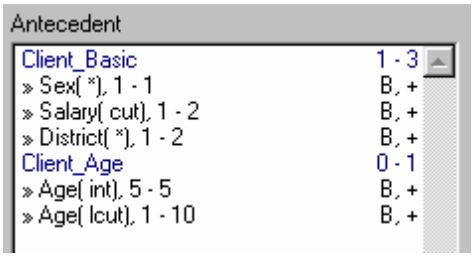
Krom rozšíření škály 4ft kvantifikátorů byly oproti předchozím implementacím metody GUHA také výrazně rozšířeny možnosti zadání úlohy. Množina všech asociačních pravidel, které mají být automaticky vygenerovány a verifikovány v zadané matici dat je dána:

- zadáním množiny relevantních antecedentů,
- zadáním množiny relevantních sukcedentů
- zadáním množin relevantních podmínek (může chybět)
- zadáním 4ft-kvantifikátoru.

Relevantní antecedenty se generují jako konjunkce literálů automaticky vytvářených ze zadané množiny antecedentových atributů. Alternativně je seznam antecedentových atributů možné rozdělit do podskupin definujících dílčí antecedenty. Dílčí antecedent je také konjunkce literálů, celý antecedent je potom konjunkcí dílčích antecedentů. Zavedení dílčích antecedentů umožňuje jemněji definovat relevantní antecedenty. Dílčí antecedent je zadán:

- seznamem atributů,
- označením každého atributu jako „basic" nebo „remaining" (dílčí antecedent musí obsahovat alespoň jeden basic atribut),
- minimálním a maximálním počtem atributů, které musí být ve vygenerovaném dílčím antecedentu,
- zadáním literálů, které budou pro každý atribut automaticky vygenerovány.

Množiny relevantních sukcedentů a podmínek jsou zadány analogicky pomocí dílčích sukcedentů a dílčích podmínek.



Obr. 8 – Ukázka zadání antecedentu

Každý dílčí antecedent může být pojmenován (např. Client_Basic, Client_Age, viz Obr. 8) a může s ním být zacházeno jako s autonomním objektem – může být přesouván mezi antecedentem sukcedentem a podmínkou jedné úlohy, ale i do zcela jiného zadání.

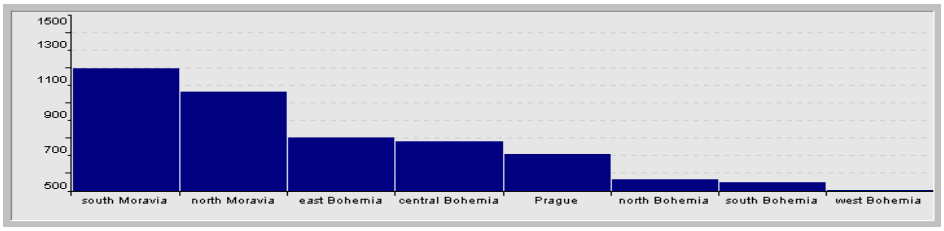
Zadání literálů které mají být automaticky je dáno typem a minimálním a maximálním počtem kategorií (možných hodnot atributu) Je k dispozici pět typů literálů, které naznačíme na příkladech. Budeme předpokládat, že máme atribut A s kategoriemi 1, 2, 3, 4 a 5.

- **Podmnožiny**: definice *podmnožiny s 2-3 kategoriemi* (symbolicky A(*), 2-3) definuje literály A(1,2), A(1,3), A(1,4), A(1,5), A(2,3), …, A(3,4), ..., A(4,5), A(1,2,3), A(1,2,4), …, A(3,4,5).
- **Intervaly**: definice *intervaly s 2-3 kategoriemi (symbolicky* A(int), 2-3) definuje literály A(1,2), A(2,3), A(3,4), A(4,5), A(1,2,3), A(2,3,4) a A(3,4,5).
- **Levé řezy**: definice levé řezy s maximálně 3 kategoriemi *(symbolicky* A(lcut) 1-3) definuje literály A(1), A (1,2) a A(1,2,3).
- **Pravé řezy:** definice pravé řezy s maximálně 4 kategoriemi *(symbolicky* A(rcut) 1-4) definuje literály A(5), A (5,4), A (5,4,3) a A(5,4,3,2).
- **Řezy** znamená pravé nebo levé řezy.

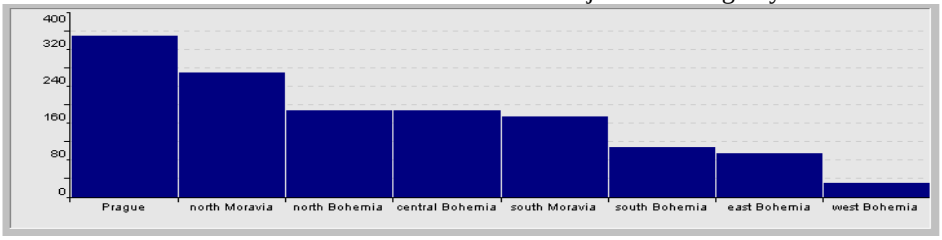
Pomocí velmi jednoduchého zadání jsme schopni nadefinovat úlohu, která bude vyžadovat vygenerovat a verifikovat miliony asociačních pravidel. Skutečné množství těchto pravidel bude záviset na množství atributů, které budou v jednotlivých cedentech, množství kategorií u těchto atributů a typech koeficientů, které se mají pro tyto atributy generovat. Aby bylo možné systém reálně používat, jsou implementovány algoritmy a optimalizace, které pro středně rozsáhlé úlohy zaručí získání výsledků v horizontu minut. V řadě úloh důležitých při výuce postačí většinou desítky vteřin.

5. Další vývoj

V současné době je připravována nebo již probíhá implementace několika procedur využívajících již vytvořené algoritmy a podprogramy. Naznačíme zde hlavní rysy procedury Pareto-Miner.
Motivace pro tuto proceduru je ukázána v obr. 9 a obr. 10. Oba se týkají rozdělení klientů banky mezi jednotlivé regiony. První se týká segmentu všech klientů, druhý pouze segmentu klientů s vysokým platem. Vidíme, že obě rozdělení se značně liší. Lze očekávat že informace o segmentech klientů které se zadaným způsobem liší od segmentu všech klientů ohledně rozdělení mezi regiony bude zajímavá pro vedení banky.



Obr. 9. - Distribuce všech klientů mezi jednotlivé regiony



Obr. 10 - Distribuce klientů s vysokým platem mezi jednotlivé regiony

Cílem procedury Pareto – Miner je nalézt všechny takové segmenty. Její vstup zahrnuje:

- matici dat
- analyzovaný atribut A (s několika kategoriemi),
- parametry definující rozsáhlou množinu podmínek stejným způsobem jako jsou pro proceduru 4ft-Miner
- kriterium zajímavosti pro jednotlivé podmínky

Kriterium zajímavosti se týká rozdělení řádků analyzované matice mezi jednotlivé hodnoty atributu A. Příklady takových kriterií jsou:

- rozdíl mezi rozdělením pro řádky splňující dílčí vygenerovanou podmínku a rozdělením pro všechny řádky, rozdíl může být měřen např. počtem kategorií atributu A s rozdílným pořadím,
- významný rozdíl mezi rozdělením pro řádky splňující vygenerovanou podmínku a rozdělením řádky splňující jednu předem danou podmínku.

Literatura

[BI 94] Berka,P. - Ivánek,J.: Automated knowledge acquisition for PROSPECTOR- like expert systems. In. (Bergadano, de Raedt eds.) Proc. ECML'94, Springer 1994, pp.339-342.

[Ha 78] Hájek, P. – Havránek T.: Mechanising Hypothesis Formation - Mathematical Foundations for a General Theory. Springer-Verlag, 1978, 396 pp.

[Ha 83] Hájek, P. – Havránek T. – Chytil, M.: Metoda GUHA. Praha, Academia, 1983, 314 pp. (in Czech).

[Hv 81] Havránek, T.: The present state of the GUHA software. International Journal of Man-Machine Studies, 15, (1981), pp. 253 – 264.

[Hn 01] David Hand, Heikki Manilla, Padhraic Smyth, Principles of Data Mining, MIT 2001

[Ra 78] Rauch, J.: Some Remarks on Computer Realisations of GUHA Procedures. International Journal of Man-Machine Studies, 10, 1978, pp. 23 – 28.

[Ra 81] Rauch J.: Main Problems and Further Possibilities of the Computer Realizations of GUHA Procedures. International Journal of Man-Machine Studies, 15, 1981, pp. 283-287

[Ra 02] Rauch, J.: Interesting Association Rules and Multi-relational Association Rules. In Communications of Institute of Information and Computing Machinery, Taiwan. Vol. 5., No. 2, May 2002, pp. 77 – 82.

[RS 00] Rauch, J. – Šimůnek, M.: Mining for 4ft Association Rules. In Discovery Science 2000. Red. Arikawa, S. – Morishita S. Springer Verlag 2000, pp. 268 – 272.

[RS 01] Rauch, J. – Šimůnek, M.: Mining for 4ft Association Rules by 4ft-Miner. In INAP 2001, The Proceeding of the International Conference On Applications of Prolog. Prolog Association of Japan, Tokyo October 2001, pp. 285 – 294.

[ST 02] Černý, Z. - Dolejší, P. - Rauch, J. – Šebek, M.: Dobývání znalostí v medicínských datech – případová studie. Nabídnuto k prezentaci na konferenci Znalosti 2003.

[Sl 02] Šlesinger, J: Předzpracování časových dat pro systém Lisp-Miner. nabídnuto k prezentaci na konferenci ZNALOSTI 2003

Dodatek C: Ontologie

Převzato z diplomové práce Hany Češpivové:

Češpivová Hana: Tvorba ontologií pro dobývání znalostí z databází, *Diplomová práce*, Fakulta informatiky a statistiky VŠE, Praha 2004

2 Ontologie

Nejprve je nutné čtenáře seznámit s pojmem ontologie a jejím významem (viz podkapitola 2.1). Ontologie popisuje entity určité věcné oblasti, jejich vlastnosti a vztahy mezi nimi. Lze ji využívat různými způsoby. Zachycuje znalosti o dané oblasti, umožňuje jejich analýzu, může sloužit jako „komunikační slovník“ mezi odborníky dané oblasti a programátory, lze ji využít v dalších aplikacích pro zpracování dat atd. Většina informací podkapitoly 2.1 je čerpána z [2].

Jak již bylo napsáno výše, pro účely diplomové práce byla vytvořena nová ontologie. Vznikla jako výřez lékařské ontologie UMLS (viz podkapitola 2.2) potřebný pro popsání entit a jejich vzájemných vztahů zachycených projektem STULONG. UMLS – Unified Medical Language System – není „klasickou ontologií“, ale propojením metatezauru, sémantické sítě a slovníku. Jde o projekt NLM (National Library of Medicine).

Jako nástroj pro tvorbu ontologie „Stulong“ byla použita aplikace Protégé-2000 (viz podkapitola 2.3). Protégé-2000 je prostředí pro tvorbu open-source ontologií, vyvinuté Stanford Medical Informatics. Jde programový nástroj pro systémové projektanty a doménové experty pro vývoj znalostních systémů. Aplikace vyvinuté s pomocí Protégé-2000 bývají používány pro řešení problémů a rozhodování v určité oblasti.

2.1 Ontologie – základní pojmy

2.1.1 Pojem ontologie

Ontologie je termín převzatý z filozofie. Tam představuje „učení zkoumající všeobecné základy bytí, jeho strukturu a zákonitosti“ [1].

Na začátku 90. let se tento termín začal používat i ve znalostním inženýrství. Ontologie zde popisuje to, co existuje v dané věcné oblasti a může tudíž být reprezentováno v rámci znalostní aplikace.

Definic ontologie je mnoho. Jednou z nejznámějších je definice T. Grubera, jednoho z duchovních otců ontologie: “Ontologie je explicitní specifikace konceptualizace[1].” [3]. Nebo také modifikace výše uvedené definice od W. Borsta: “Ontologie je formální specifikace sdílené konceptualizace.” [4].

A na závěr ještě definice uváděná pro aplikaci Protégé-2000: „ Ontologie je model konkrétní oblasti vědění – konceptů a jejich vlastností, stejně jako vztahů mezi nimi.“ [5].

[1] Pojem *konceptualizace* představuje systém termínů dané oblasti a vztahů mezi nimi.

2.1.2 Účel ontologie

Ontologie představuje společný slovník pro vědce, experty dané oblasti, vývojové pracovníky, či konečné uživatele návazných aplikací, kteří potřebují sdílet informace dané oblasti, což zahrnuje počítačem interpretovatelné definice základních pojmů oblasti a vztahů mezi nimi.

Některé z důvodů proč vytvářet ontologie [8]:

- podpora porozumění mezi lidmi a programovými agenty,
- možnost opětovného využití znalostí dané oblasti,
- vyjasnění (definování) předpokladů dané oblasti,
- oddělení znalostí dané oblasti od operačních znalostí,
- pro analýzu znalostí dané oblasti.

Podpora porozumění mezi lidmi a programovými agenty je jedním z nejčastěji zmiňovaných cílů při tvorbě ontologií. Jako příklad si uveďme různé internetové stránky obsahující medicínská data. Pokud tyto internetové stránky používají shodnou ontologii medicínských termínů, pak počítačoví agenti mohou čerpat a třídit informace z těchto internetových stránek a mohou pomocí nich odpovídat na otázky uživatele nebo je použít jako vstupní data pro jiné aplikace.

Možnost opětovného využití znalostí dané oblasti byla jednou z hnacích sil nedávného pokroku ve výzkumu ontologií. Mnoho oblastí, které jsou popisovány pomocí ontologií, mají něco společného (např.: vnímání času atd.). Pokud tedy někdo popíše tuto oblast ve své ontologii, ostatní mohou tuto „společnou“ část využít pro svou zkoumanou oblast. Při tvorbě nové ontologie je tedy možné použít již existující ontologii, nebo při tvorbě rozsáhlé ontologie dokonce i spojit několik již existujících ontologií. Například UMLS (použitá v této diplomové práci) je propojením několika dílčích ontologií, či konečná ontologie vytvořená pro tuto diplomovou práci je také založena na již existující ontologii.

Vyjasnění (definování) předpokladů dané oblasti umožňuje tyto předpoklady snadněji měnit, pokud se změní i naše znalosti o dané oblasti. Pro někoho bez odborných znalostí je těžké určitému předpokladu dané oblasti porozumět, natož ho změnit, při získání nových znalostí. Jednoznačná specifikace doménových znalostí je užitečná také pro nové uživatele, kteří musí zjistit, co konkrétní termíny v dané oblasti znamenají.

Oddělení znalostí dané oblasti od operačních znalostí (programových algoritmů) je dalším obecným využitím ontologií. Vezměme si úlohu sestavení produktu z určitých komponent, tak aby odpovídal požadované specifikaci a tvorbu programu, který toto sestavení provede nezávisle na produktu a jeho komponentách. To můžeme provést tak, že vytvoříme ontologii PC komponent a jejich vlastností a na ni aplikujeme algoritmus pro konfiguraci počítačů podle objednávku. Ten samý algoritmus pak můžeme použít pro konfiguraci výtahů, pokud „naplníme“ ontologii potřebnými daty (např. ontologie SYSYPHUS).

Analýza znalostí dané oblasti je možná, jakmile je k dispozici specifikace termínů dané oblasti. Formální analýza termínů je nesmírně hodnotná jak při možnosti použít již existující ontologie tak i při jejich rozšiřování.

Někdy není tvorba ontologie dané oblasti cílem sama o sobě. Vývoj ontologie se podobá definování dat a jejich struktury pro jiné programy. Generické modely úloh, aplikace nezávislé na oblasti a programoví (software) agenti používají ontologie a z nich vytvořené znalostní báze jako data.

2.1.3 Typy ontologií

Ontologie lze členit podle různých hledisek, zde si uvedeme ty nejvíce používané.

Členění podle „historických paradigmat“

Problematika ontologií se rozpadá (i když ne zcela disjunktně) na přinejmenším tři hlavní oblasti, které lze chápat jako součást vývoje tradičnějších oborů. Jak už to bývá, hranice mezi nimi se pomalu zmenšují.

- *Terminologické* či *lexikální* ontologie lze ztotožnit s pokročilými tezaury, používanými v knihovnictví a dalších oborech orientovaných na textové zdroje. Jejich charakteristickým rysem[1] je ústřední role *termínů*, které již nejsou dále (formálně) definovány. Používané *relace* mají z velké části taxonomický charakter (vymezení vztahu obecnějšího a speciálnějšího termínu), vedle toho bývá vyjádřena synonymie, meronymie (vztah termínů označujících celek a jeho část) a další relace obecného charakteru. Nejznámější terminologická ontologie je nepochybně WordNet (http://www.cogsci.princeton.edu/~wn/); z něj byl odvozen např. Sensus (http://www.isi.edu/natural-language/projects/ONTOLOGIES.html) nebo vícejazyčná varianta EuroWordNet (http://www.illc.uva.nl/EuroWordNet/).
- *Informační ontologie* jsou rozvinutím *databázových* konceptuálních schémat. Hrají roli nadstavby nad primárními (strukturovanými, např. relačně-databázovými) zdroji, pro které zabezpečují jednak konceptuální abstrakci potřebnou pro pojmové dotazování, jednak vyšší úroveň kontroly integrity než běžné nástroje.
- *Znalostní ontologie* navazují na výzkum v oblasti reprezentace znalostí v rámci *umělé inteligence*. Ontologie jsou zde chápány důsledně jako *logické teorie*, a jejich vazba na reálné objekty (instance) je oproti informačním ontologiím relativně volná. Třídy (koncepty) a relace jsou systematicky definovány prostřednictvím formálního jazyka. Příkladem takovéto ontologie je třeba Cyc (http://www.cyc.com/) nebo i ontologie „UMLS_2“, vytvořená pro účely této diplomové práce (viz kapitola 5).

Členění podle míry formalizace

Přestože je formalizace, jak již bylo řečeno, do jisté míry definiční vlastností ontologií, smysluplné využití mají i „ontologie“ zcela neformální či „semi-formální“. Jde zpravidla o glosáře, v nichž jsou jednotlivé pojmy vysvětleny přirozeným jazykem (volnou či strukturovanou formou). Ontologie vyjádřené ve formálních jazycích pak lze dále rozlišovat podle formálně-logických vlastností daného jazyka, jako je úplnost a rozhodnutelnost; tyto vlastnosti vycházejí z vlastností logického kalkulu, na kterém je jazyk založen, např. deskripční logiky.

[1] Ve srovnání s oběma zbývajícími typy ontologií (informačními a znalostními), které lze souhrnně charakterizovat jako *konceptuální*.

Většina formálních ontologií v sobě ovšem svým způsobem zahrnují i ontologii neformální. Jednotlivé konstrukty bývají totiž vybaveny dokumentační položkou, umožňující vyjádřit obsah přirozeným jazykem.

Členění podle předmětu formalizace

Jedná se o tradiční členění s řadou variant navržených různými autory; zde uvedeme pouze hlavní typy.

- *Doménové* ontologie jsou typem daleko nejfrekventovanějším. Jejich předmětem je vždy určitá specifická věcná oblast, vymezená šířeji (např. celá problematika medicíny nebo fungování firmy) či úžeji (problematika určité choroby, poskytování úvěru apod.). Příklady doménových ontologií se širokým záběrem jsou Enterprise Ontology *(http://www.aiai.ed.ac.uk/~entprise/enterprise/ontology.html)* nebo lékařská On9 *(http://saussure.irmkant.rm.cnr.it/onto)*.
- *Generické* ontologie usilují o zachycení obecných zákonitostí, které platí napříč věcnými oblastmi, např. problematiky času, vzájemné pozice objektů (topologie), skladby objektů z částí (mereologie) apod. Někdy se ještě výslovně vyčleňují tzv. ontologie *vyšší úrovně* („upper-level“), které usilují o zachycení nejobecnějších pojmů a vztahů, jako základu taxonomické struktury každé další (např. doménové) ontologie; nejnovějším výsledkem tohoto směru je SUMO – Standard Upper Merged Ontology (http://ontology.teknowledge.com/). Ontologie typu *common-sense* („přirozeného rozumu“) mohou naopak obsahovat řadu velmi specifických, avšak relativně doménově-nezávislých znalostí, které lidé používají v každodenním životě. Nejznámějším příkladem je *Cyc*.
- Jako *úlohové ontologie* jsou někdy označovány generické modely znalostních úloh a metod jejich řešení. Na rozdíl od ostatních ontologií, které zachycují znalosti o světě („tak, jak je“), se zaměřují na procesy odvozování. Mezi úlohy tradičně zachycené pomocí takových znalostních modelů patří např. diagnostika, zhodnocení („assessment“), konfigurace, nebo plánování.
- *Aplikační ontologie* jsou nejspecifičtější. Jedná se o smíšení modelů převzatých a adaptovaných pro konkrétní aplikaci, zahrnující zpravidla doménovou i úlohovou část (a tím automaticky i generickou část).

2.1.4 Využití ontologií v praxi

Za hlavní oblasti využití ontologií se v současnosti označují např.:

- *Znalostní management* ve firmách. Pro efektivní fungování organizace je třeba, aby se informace a znalosti (jak interního tak externího původu) neztrácely, a včas se dostávaly k těm pracovníkům, kteří je mohou využít. S pomocí ontologie je možné zachytit věcnou podstatu znalostí, a tím jednak zabezpečit jejich konzistenci, jednak usnadnit jejich vyhledání.
- *Elektronické obchodování* typu B-to-C i B-to-B. V prvním případě může ontologie usnadnit vyhledání požadovaného produktu zákazníkem, ve druhém se jedná o rychlé vyhledání potenciálního partnera, ale perspektivně také o automatizaci procesu sjednání obchodních podmínek.

- *Zpracování přirozeného jazyka* – terminologické ontologie mohou napomáhat např. při překladu nebo automatické sumarizaci textů.
- *Inteligentní integrace informací*. Ontologie může sloužit jako zastřešení datových schémat distribuovaných zdrojů (strukturovaných nebo semi-strukturovaných databází, případně „tabulárních“ webových stránek) na vysoké úrovni abstrakce.
- Pojmové *vyhledávání informací* jako vylepšení stávajících internetových vyhledávačů.
- Sémantické *webové portály* konstruované polo-automaticky na základě metadat od poskytovatelů informace.
- Inteligentní *výukové systémy*.

Skutečností ovšem je, že ambiciózní myšlenky akademického výzkumu značně předbíhají praxi, která k potřebě využívání ontologií dospívá jen pozvolna. Vzhledem k překotnému vývoji problematiky také nepřekvapí, že již realizované praktické aplikace často odrážejí starší stupeň vývoje ontologických jazyků.

2.1.5 Struktura ontologií

Přestože je základní struktura znalostních ontologií prakticky ve všech hlavních projektech, jazycích a nástrojích obdobná, používaná terminologie se značně liší, což znesnadňuje orientaci. Protože pro tvorbu ontologie „UMLS_2“ budeme jako nástroj využívat systém *Protégé-2000* (viz podkapitola 2.3), tak budeme používat jeho terminologii.

V průvodci pro tvorbu ontologie[1] [8] je uvedena tato definice: „Ontologie je formální explicitní popis pojmů dané oblasti (třídy), vlastností každého pojmu popisujícími různé aspekty a atributy pojmu (sloty) a omezení těchto atributů (facety). Ontologie společně s jednotlivými příklady pojmů (instancemi) tvoří znalostní bázi (knowledge base).“

Třídy

Základem většiny ontologií[2] jsou třídy, které označují množiny konkrétních objektů dané oblasti. Místo termínu třída se někdy používá výraz koncept („pojem“) případně kategorie. Termín třída úzce souvisí s termínem rámec – základním pojmem mnoha systémů umělé inteligence.

Třídy v ontologii tvoří hierarchii, v níž nižší třídy dědí vlastnosti a omezení tříd vyšších. Tyto vlastnosti se pak na nižších úrovních mohou dále konkretizovat (zjemňovat). V praxi se také využívá vícenásobná dědičnost – jedna podtřída může spadat a tudíž i dědit vlastnosti a omezení od dvou či více odlišných vyšších tříd.

[1] *Ontology Development 101: A Guide to Creating Your First Ontology* – dokument dostupný na stránkách Stanfordské univerzity (zdroj), jejímž dílem je i již zmiňovaný *Protégé-2000*. Podle toho průvodce byla vytvořena 1. verze ontologie, postup tvorby je zdokumentován dále (viz kapitola 5).

[2] Jak už bylo výše uvedeno, budeme tvořit znalostní ontologii. Uváděná struktura proto odpovídá hlavně znalostním ontologiím.

Definice v Protégé-2000: „Třída je abstrakt reprezentující pojem v dané oblasti jako soubor odpovídajících tříd. Třída může mít množinu slotů, které představují vlastnosti dané třídy.“

Sloty, vlastnosti, role, atributy

Jak již bylo napsáno výše, každá třída má definovány určité vlastnosti (atributy) či sloty (role). Každá nižší třída dědí vlastnosti od třídy vyšší a od této vyšší třídy se odlišuje tím, že má definovány další nové vlastnosti, případně jsou zděděné vlastnosti dále konkretizovány nebo měněny.

Definice v Protégé-2000: „ Slot je vlastností dané třídy. Zděděný slot, je vlastnost přidělená dané třídě prostřednictvím dědění od rodičovské třídy (tedy třídy o úroveň vyšší).“

Omezení atributů (slotů), facety

Atributy či sloty mohou mít definována omezení (facety). Jedná se o omezení vlastnosti dané třídy, např. omezení oboru hodnot, kardinalita atd.

Definice v Protégé-2000: „Faceta je vlastností slotu. Některé facety závisí na hodnotě typu facety. Například typ slotu *integer – číslo* má minimum a maximum.“

V Protégé-2000 jsou k dispozici tato omezení: typ hodnoty slotu (boolean, celé číslo, desetinné číslo, řetězec alfanumerických znaků, seznam hodnot, třída a instance), vzorové hodnoty, kardinalita, defaultní hodnota, popřípadě minimum a maximum.

Instance

Instance odpovídá konkrétnímu objektu reálného světa. Bývá prvkem určité třídy, ale může být do ontologie vloženo i volně bez vazby na konkrétní třídu.

Rozhodnutí, zda určitá entita bude považována za třídu nebo instanci, závisí především na úhlu pohledu a na účelu ontologie.

Definice v Protégé-2000: „Instance je konkrétní výskyt informace o dané oblasti, který je vložen do znalostní báze.“ Instance lze vkládat pomocí formuláře vygenerovaného v Protégé-2000.

2.2 UMLS

Pro účel této diplomové práce byla vytvořena ontologie „UMLS_2“. Jde vlastně o znalostní lékařskou ontologii. Byla vytvořena jako „výřez“ z již existující lékařské ontologie UMLS. Tento „výřez“ byl vybrán podle potřeb projektu STULONG, tedy podle používaných lékařských pojmů v tomto projektu.

UMLS (Unified Medical Language System) je dlouhodobým projektem National Library of Medicine (dále jen NLM) [6]. Jde o výzkum a vývoj systému, který by měl usnadnit vyhledávání (získávání) a integraci informací z četných počítačem

zpracovatelných lékařských informačních zdrojů. Jde o tyto zdroje: lékařská literatura, vědecké záznamy, databanky, znalostní systémy a seznamy lidí a organizací.

Největšími bariérami pro získávání a integraci informací z těchto zdrojů jsou:

- rozmanitost slovní zásoby a klasifikací použitých v různých zdrojích a různými uživateli a
- celkové množství a šíře distribuce potenciálně relevantních informačních zdrojů.

Tyto bariéry odrazují profesionály a vědce ve zdravotní sféře od užívání dostupných počítačem zpracovatelných informací a také brání vývoji efektivního vyhledávacího rozhraní, které by těmto uživatelům mohlo pomoci.

Vize UMLS je založena na vývoji „intelektuálního polotovaru“ ve formě počítačem zpracovatelného zdroje znalostí. Ten by měl být využíván širokou škálou aplikačních programů, aby kompenzoval rozdíly týkající se definování konceptů v různých počítačem zpracovatelných zdrojích nebo různými uživateli. Cílem je zjednodušení, vývoj systému, který spojí informace ze záznamů o pacientech, bibliografických databází, faktografických databází, expertních systémů atd. UMLS také může ulehčit vývoj aplikací pro indexování a pro tvorbu dat.

Projekt UMLS je řízen multi-disciplinárním týmem zaměstnanců NLM a jimi vybranými vědci. Na počátku roku 2003 mělo licenci na UMLS více než 1500 institucí nebo osob po celém světě. Současně byl využíván ve škále různých aplikací a výzkumných projektů. Například sama NLM využívá metatezaurus UMLS pro NLM Gateway a PubMed pro zlepšení vyhledávání MEDLINE. Další projekt NLM – ClinicalTrials.gov – používá UMLS pro automatické přidávání příbuzných (synonymních) slov k vstupnímu termínu při vyhledávání. Indexing Initiative (http://ii.nlm.nih.gov) používá UMLS jako hlavní část pro investigativní automatickou metodu indexování.

UMLS zahrnuje 3 zdroje znalostí: Metatezaurus, sémantickou síť a speciální slovník.

Metatezaurus

Metatezaurus obsahuje sémantické informace o medicínských pojmech, jejich různá pojmenování a vztahy mezi nimi. Je vytvořený z tezauru, klasifikace (třídění), systémů kódování a seznamů řízených termínů, které jsou rozvíjeny a udržovány mnoha různými organizacemi. Obsahuje a propojuje mnoho standardních vědeckých a medicínských slovníků.

Je to vlastně databáze informací o konceptech, které se vyskytují v jednom či více výskytech různých řízených slovníků a klasifikací používaných v oboru medicíny. Metatezaurus uchovává významy, vlastnosti, hierarchické a další vztahy mezi termíny obsaženými v jeho zdrojových slovnících.

Metatezarus se používá v mnoha různých aplikacích. Například: informační vyhledávání z databází nebo textových informačních zdrojů při zadávání vstupního termínu (indexu) člověkem; propojování záznamů o pacientech se souvisejícími informacemi v bibliografických, full-textových nebo faktografických databázích; výzkum zpracování přirozeného jazyka a automatického indexování, atd. V mnoha případech je využití Metatezauru zlepšeno, když je použit v kombinaci se SPECIALIST Lexicon, lexikálními programy a sémantickou sítí.

Sémantická síť

Sémantická síť je sítí obecných kategorií nebo sémantických druhů, pod které byly rozděleny všechny koncepty v metatezauru. Jejím cílem je poskytnout konzistentní kategorizaci všech konceptů obsažených v metatezauru a dále poskytnout množinu užitečných vztahů mezi koncepty.

Všechny informace o konkrétních konceptech lze nalézt v metatezauru; sémantická síť poskytuje informace pouze o skupině základních sémantických druhů nebo kategorií (pod které jednotlivé koncepty spadají) a také definuje množinu vztahů, které mohou mezi sémantickými druhy být. Verze sémantické sítě z roku 2003 obsahuje 135 sémantických druhů a 54 vztahů (viz příloha 1).

Sémantická síť slouží jako autorita pro sémantické druhy, pod které jsou přiřazeny jednotlivé koncepty z metatezauru. Sémantická síť tyto typy definuje a to jak slovním popisem tak i prostřednictvím informace vycházející z jejich hierarchie.

Speciální slovník – SPECIALIST Lexicon

Slovník SPECIALIST Lexicon obsahuje syntaktické informace o medicínských termínech a může eventuálně pokrýt většinu jednotlivých termínů v názvech konceptů vyskytujících se v metatezauru. Tento slovník byl vytvořen pro poskytování lexikálních informací potřebných pro SPECIALIST Natural Language Processing System (NLP – systém pro zpracování přirozeného jazyka). Je to obecný anglický slovník zahrnující mnoho lékařských termínů. Lexikálním vstupem pro každé slovo nebo termín je syntaktický, morfologický a pravopisný záznam.

S UMLS je kromě slovníku distribuováno také množství lexikálních programů, jakožto silné nástroje pro vyhledávání, indexování a zpracovávání slov.

2.3 Protégé-2000

Nástrojem pro tvorbu ontologie pro účel této diplomové práce je systém Protégé-2000. Jako začátečníkovi v tvorbě ontologií mi byl doporučen při konzultaci. Postupem času se ukázalo, že to bylo dobré rozhodnutí, protože Protégé 2000 je uživatelsky jednoduchý a dobře se mi s ním pracovalo. Navíc pro něj byla vypracována uživatelská příručka [8] pro tvorbu prvních ontologií, kterou jsem využila pro tvorbu ontologie „Stulong“ (viz kapitola 5).

Protégé-2000, prostředí pro tvorbu open-source ontologií, byl vyvinutý Stanford Medical Informatics[1], za podpory mnoha aktivních uživatelů. Je to programový nástroj používaný systémovými projektanty a doménové experty pro vývoj KBS (počítačové systémy, které obsahují znalostní bázi dané oblasti a programy s pravidly pro

[1] Protégé-2000 byl vyvinut na Stanfordské univerzitě. Vytvořili ho medicínští informatici s podporou Defense Advanced research Projects Agency, National Cancer Institute, National Institute of Standards and Technology, National Library of Medicine, National Science Foundation a s další podporou od svých přidružených členů DaimlerChrysler, Fast Track Systems a Sowerby Centre for Health Informatics v Newcastlu.

zpracování znalostí a pro řešení problémů souvisejících s danou oblastí). Aplikace vyvinuté s pomocí Protégé-2000 jsou používány pro řešení problémů a rozhodování v určité oblasti.

Protégé-2000 by měl představovat:

- nástroj, který umožňuje uživateli sestavit doménovou ontologii, vytvořit formuláře pro vkládání dat a vkládat data,
- platformu, která může být rozšířena pomocí grafických pomůcek pro tabulky, grafy či animace kvůli zpřístupnění jiných aplikací týkajících se znalostních systémů (KBS),
- knihovnu, kterou mohou používat jiné aplikace k přístupu a využití znalostní báze.

Na domovské stránce systému Protégé-2000 [7] je popsán přibližně takto:

Systém Protégé prošel dlouhým vývojem od svých počátků v roce 1987, kdy vznikl jako meta-nástroj pro KBS. Původně to byla malá aplikace zaměřená na tvorbu nástrojů pro získávání znalostí pro několik speciálních programů v lékařském projektování. Z tohoto původního nástroje se systém Protégé vyvinul v odolnou a rozšiřitelnou platformu pro vývoj a výzkum KBS. Současná verze Protégé-2000 může být spuštěna na různých platformách, podporuje nástavby uživatelských interface zhotovených na zakázku, obsahuje znalostní model Open Knowledge Base Connectivity (OKBC – protokol pro zpřístupnění znalostních základen uložených v systémech reprezentujících znalosti), umožňuje interakci se standardními formáty ukládání, jako jsou relační databáze, XML a RDF a může být používána stovkami lidí a výzkumných skupin či organizací.

Dřívější verze Protégé/Win definovala třídy a ukládala instance těchto tříd odděleně jako klasický databázový systém. Na rozdíl od ní Protégé-2000 usnadňuje práci současně jak s třídami tak i s instancemi. Takže konkrétní instance může být použita na úrovni třídy a třída může být uložena jako instance. Obdobně sloty, které dříve byly používány pouze v rámci tříd, jsou nyní povýšený na tu samou úroveň jako třídy.

Nástroj Protégé-2000 zastává všechny tyto úlohy díky jednotnému GUI (graphical user interface – grafický uživatelský interface). Jeho nejvyšší úroveň je složena z překrývajících se panelů pro celistvou prezentaci jednotlivých částí a pro jejich praktickou editaci. Tato forma „panelové“ nejvyšší úrovně dovoluje integraci

- vytváření ontologie tříd popisujících konkrétní subjekt,
- tvorby nástroje pro sbírání znalostí,
- vkládání vlastních instancí a tvorby znalostní báze a
- realizaci aplikací.

Jedním z podnětů pro vývoj Protégé-2000 bylo to, že znalostní systémy jsou obvykle velmi drahé (jak při tvorbě, tak při nákupu). Přepokládá se totiž, že vývoj KBS je týmovou prací, která zahrnuje jak vývojové projektanty tak odborníky dané oblasti. Ti však bývají méně obeznámeni s počítačovým softwarem. Protégé-2000 je navržen tak, aby vedl vývojové projektanty a odborníky v průběhu návrhu systému. Měl by vývojovým projektantům umožnit použití již existujících ontologií či PSM a tím zkrátit čas potřebný pro vývoj a údržbu programu. Některé aplikace mohou používat tu samou

doménovou ontologii pro řešení různých problémů nebo ta samá PSM může být použita různými ontologiemi.

Protégé-2000 je v současné době používán hlavně v klinické medicíně a biomedicíně, ačkoli může být využit v jakémkoli oboru, kde lze využít modelování hierarchií pojmů dané oblasti.

Literatura:

(1) Kolektiv autorů: Slovník cizích slov. 1. vydání Olomouc 1992.

(2) Svátek, V.: Ontologie a WWW. Tutoriál ke konferenci Datakon 2002. http://nb.vse.cz/~svatek/temata.htm

(3) Gruber, T.R.: A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition 1993.

(4) Haustein, S., Pleumann, J.: Is Participation in the Semantic Web Too Difficult? In: (Horrocks, I., Hendler, J., eds.) The Semantic Web – ISWC 2002. Springer 2002.

(5) Protégé-2000: User´s Guide. http://protege.stanford.edu/useit.html

(6) UMLS. http://www.nlm.nih.gov/research/UMLS/

(7) Protégé-2000. http://protege.stanford.edu

(8) Noy, N. F., McGuinness, D.: Ontology Development 101: A Guide to Creating Your First Ontology. http://protege.stanford.edu/publications/entology_development/

Dodatek D: Obsah přiloženého CD

V tabulce jsou uvedeny adresáře a významné soubory uložené na přiloženém CD.

Název souboru či adresáře	**Popis obsahu souboru či adresáře**
`BKzdroj`	Zdrojový kód implementace validace background knowledge
`BKzdroj\c#`	Kód implementace krabiček
`BKzdroj\slice`	SLICE návrh krabiček
`ferda`	Zdrojové kódy prostředí Ferda (s validací *background knowledge*)
`ferdaInstalace`	Adresář pro instalaci prostředí Ferda
`ferdaInstalace\howToInstall.html`	Pokyny k instalaci prostředí Ferda
`obrazky`	Obrázky použíté v diplomové práci
`stulongValidace`	Datový zdroj STULONG a projekty na validaci pravidel background knowledge
`text`	Text diplomové práce
`zdroje`	Některé ze zdrojů z literatury, pojmenované číslem, pod kterým jsou uvedeny v referencích.