VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ
Ústav automatizace a informatiky

Expertní systémy

Jiří Dvořák

2004

Obsah

Předmluva

Expertní systémy patří mezi nejúspěšnější aplikace umělé inteligence. Od svého komerčního uvedení na začátku 80-tých let minulého století prodělaly bouřlivý rozvoj a v současnosti jsou používány v mnoha oblastech lidské činnosti, jako např. ve vědě, technice, výrobě, obchodě atd.

Tyto učební texty jsou určeny pro předmět *Expertní systémy* v magisterských studijních programech *Inženýrská informatika a automatizace* (specializace *Informatika*) a *Aplikovaná informatika a řízení*. Obsahově tato problematika souvisí s předměty *Algoritmy umělé inteligence* a *Jazyky pro umělou inteligenci*.

Učební texty jsou členěny do sedmi kapitol. Prvá kapitola obsahuje úvod do expertních systémů. Další dvě kapitoly jsou věnovány způsobům reprezentace znalostí a mechanismům jejich využívání, přičemž se neberou do úvahy neurčitosti. Metodami zpracování neurčitosti se zabývá kapitola 4. Kapitola 5 charakterizuje proces tvorby expertního systému. Úzkým místem tohoto procesu je získávání znalostí. V něm hraje významnou roli strojové učení, kterému je věnována kapitola 6. Z mnoha prostředků pro tvorbu expertních systémů byl pro podrobnější prezentaci v těchto učebních textech zvolen volně šiřitelný systém CLIPS, založený na dopředném usuzování a porovnávání se vzorem. Jiným důležitým způsobem usuzování je usuzování zpětné, kterým je vybaven např. vyhodnocovací systém jazyka Prolog. Tento jazyk by měl být popsán v plánovaných učebních textech pro předmět *Jazyky pro umělou inteligenci*.

V těchto učebních textech jsou prezentovány některé výsledky, dosažené v rámci výzkumného záměru CEZ: J22/98: 261100009 *Netradiční metody studia komplexních a neurčitých systémů*.

1. Úvod do expertních systémů

1.1 Charakteristika expertních systémů

Feigenbaum definuje ***expertní systém*** (ES) jako počítačový program, simulující rozhodovací činnost experta při řešení složitých úloh a využívající vhodně zakódovaných, explicitně vyjádřených znalostí, převzatých od experta, s cílem dosáhnout ve zvolené problémové oblasti kvality rozhodování na úrovni experta.

Expertní systémy se vyznačují následujícími charakteristickými rysy (typickým je prvý rys, další rysy jsou žádoucí, ale nemusejí být vždy přítomny):

- oddělení znalostí a mechanismu jejich využívání (tím se expertní systémy odlišují od klasických programů),
- schopnost rozhodování za neurčitosti,
- schopnost vysvětlování.

V literatuře se můžeme setkat také s pojmem ***znalostní systém*** (*knowledge-based system*), který je podle staršího pojetí obecnější než pojem expertní systém. Expertní systém tedy lze chápat jako zvláštní typ znalostního systémů, který se vyznačuje používáním expertních znalostí a některými dalšími rysy, jako je např. vysvětlovací mechanismus. V poslední době však dochází ke stírání rozdílů mezi těmito pojmy.

Problematice expertních systémů jsou věnovány např. knihy (Giarratano a Riley 1998), (Gonzalez a Dankel 1993), (Jackson 1999), (Liebowitz a De Salvo 1989), (Nikolopoulos 1997) a (Popper a Kelemen 1988).

1.2 Struktura expertního systému

Expertní systém obsahuje tyto základní složky (jejich vzájemné vazby ukazuje obr. 1.1):

- bázi znalostí,
- inferenční mechanismus,
- I/O rozhraní (uživatelské, vývojové, vazby na jiné systémy),
- vysvětlovací modul,
- modul pro akvizici (získávání) znalostí.

Báze znalostí

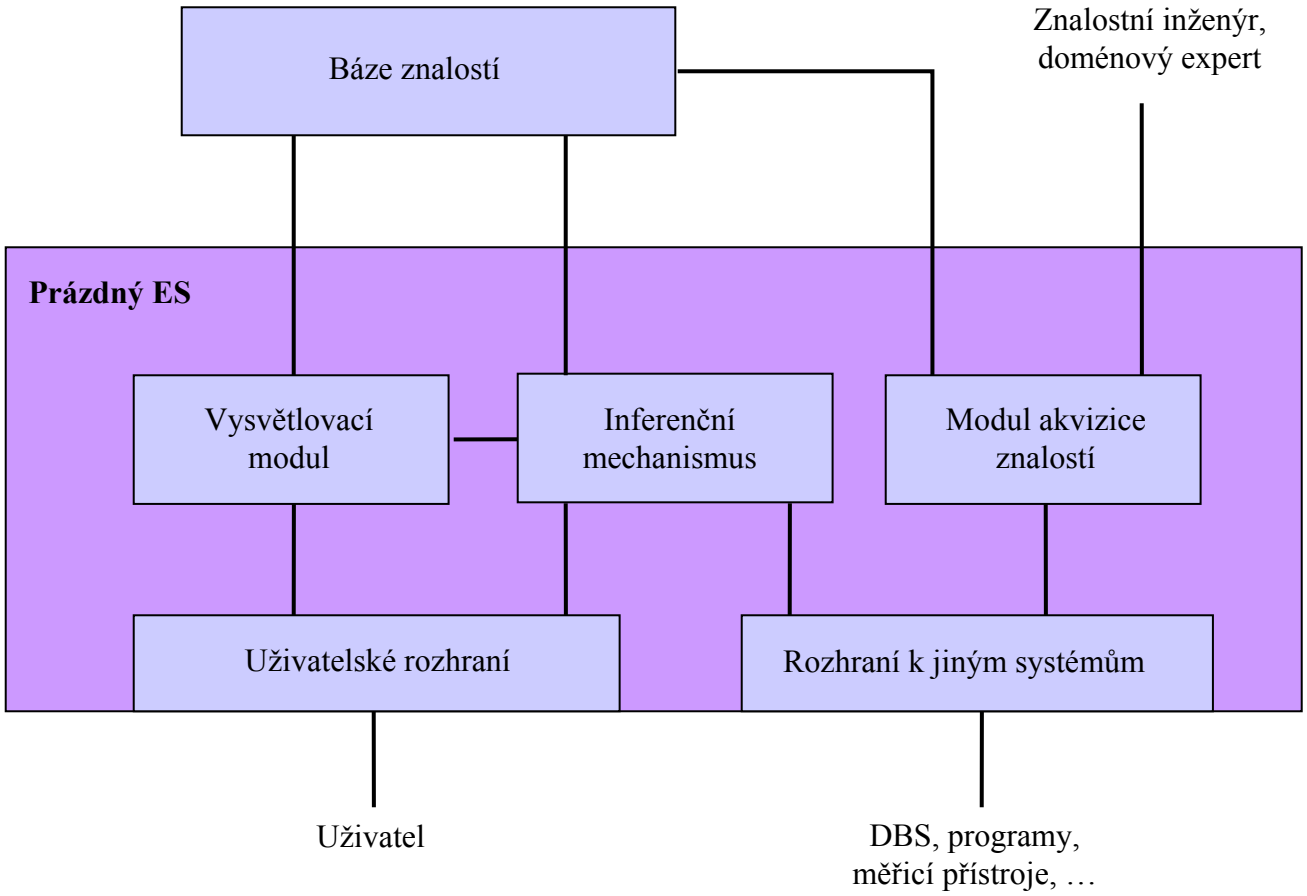
Báze znalostí obsahuje znalosti z určité domény a specifické znalosti o řešení problémů v této doméně. Znalosti mohou být nejrůznějšího charakteru:

- od nejobecnějších znalostí až po znalosti vysoce specifické
- od „učebnicových“ znalostí až po znalosti „soukromé“ (tj. takové, jaké by si expert ani nedovolil publikovat)
- od exaktně prokázaných znalostí až k nejistým heuristikám

- od jednoduchých znalostí až po metaznalosti (tj. znalosti o znalostech)

Pro reprezentaci znalostí jsou nejčastěji používány tyto prostředky:

- matematická logika,
- pravidla (*rules*),
- rozhodovací stromy (*decision trees*)
- sémantické sítě (*semantic nets*),
- rámce a scénáře (*frames and scripts*),
- objekty (*objects*).



Obr 1.1. Architektura expertního systému

Znalosti můžeme klasifikovat jako:

- *mělké* (*shallow knowledge*), které jsou založeny na empirických a heuristických znalostech.
- *hluboké* (*deep knowledge*), založené na základních strukturách, funkcích a chování objektů.

Obdobně můžeme i usuzování charakterizovat jako mělké (*shallow reasoning*) nebo hluboké (*deep reasoning*). Pro hluboké usuzování jsou užitečnými modely rámce a objekty.

V průběhu řešení konkrétního problému se vytváří ***báze faktů***, která obsahuje data k řešenému problému (vstupní údaje a postupně odvozované výsledky).

Inferenční mechanismus

Inferenční mechanismus obsahuje obecné (doménově nezávislé) algoritmy schopné řešit problémy na základě zadaných faktů pomocí manipulace se znalostmi z báze znalostí. Typický inferenční mechanismus je založen na

- inferenčním pravidle pro odvozování nových poznatků z existujících znalostí,
- strategii prohledávání báze znalostí.

V (Giarratano a Riley 1998) je uveden následující přehled metod inference:

- *Dedukce* – logické usuzování, při němž závěry musejí vyplývat z předpokladů.
- *Indukce* – postup od specifického případu k obecnému.
- *Abdukce* – usuzování směřující ze správného závěru k předpokladům, které jej mohly způsobit.
- *Heuristiky* – pravidla „zdravého rozumu“ založená na zkušenostech.
- *Generování a testování* – metoda pokusů a omylů.
- *Analogie* – odvozování závěru na základě podobnosti s jinou situací.
- *Defaultní inference* – usuzování na základě obecných znalostí v případě absence znalostí specifických.
- *Nemonotonní inference* – je možná korekce resp. ústup od dosavadních znalostí na základě nového pozorování.
- *Intuice* – obtížně vysvětlitelný způsob usuzování, jehož závěry jsou možná založeny na nevědomém rozpoznání nějakého vzoru. Tento typ usuzování zatím nebyl v ES implementován a snad by se k němu mohlo blížit usuzování neuronových sítí.

Důležitou schopností inferenčního mechanismu je zpracování neurčitosti.Neurčitost v expertních systémech se může vyskytovat jednak v bázi znalostí a jednak v bázi faktů. Zdroji neurčitosti jsou:

- nepřesnost, nekompletnost, nekonzistence dat,
- vágní pojmy,
- nejisté znalosti.

Neurčitost může být reprezentována a zpracovávána např. pomocí následujících přístupů a prostředků:

- pravděpodobnostní (Bayesovské) přístupy,
- faktory jistoty,
- Dempster-Shaferova teorie,
- fuzzy logika.

1.3 Typy expertních systémů

Expertní systémy můžeme klasifikovat podle různých hledisek. Podle obsahu báze znalostí můžeme expertní systémy rozdělit na:

- ***problémově orientované***, jejichž báze znalostí obsahuje znalosti z určité domény.
- ***prázdné*** (*shells*), jejichž báze znalostí je prázdná.

Podle charakteru řešených problémů můžeme expertní systémy rozdělit na

- ***diagnostické***, jejichž úkolem je určit, která z hypotéza z předem definované konečné množiny cílových hypotéz nejlépe koresponduje s daty týkajícími se daného konkrétního případu.
- ***plánovací***, které obvykle řeší takové úlohy, kdy je znám cíl řešení a počáteční stav a je třeba s využitím dat o konkrétním řešeném případu nalézt posloupnost kroků, kterými lze cíle dosáhnout.

1.4 Tvorba expertních systémů

V procesu tvorby expertního systému se vyskytují následující činnosti:

- výběr hardwaru a softwaru,
- návrh uživatelského rozhraní,
- akvizice znalostí (získání a reprezentace znalostí),
- implementace,
- validace a verifikace.

Vytvářením expertních systémů se zabývá ***znalostní inženýrství*** (*knowledge engineering*). V procesu tvorby ES představuje úzké místo akvizice znalostí (*knowledge acqusition bottleneck*). Toto úzké místo pomáhají překonat metody ***strojového učení*** (*machine learning*).

Nástroje pro tvorbu expertních systémů můžeme rozdělit do těchto skupin:

- prázdné expertní systémy (např. EXSYS, FLEX, G2, HUGIN, M4, ...),
- speciální programová prostředí (CLIPS, OPS5, Lisp, Prolog, ...),
- obecná programová prostředí (např. Pascal, Delphi, C, C++Builder, ...).

1.5 Aplikace ES

Aby mělo smysl použít expertní systém pro řešení nějakého problému, musejí být splněny dvě následující podmínky:

1. Musí se jednat o problém složitý rozsahem nebo neurčitostí vztahů, pro nějž exaktní metoda řešení buď není k dispozici, nebo není schopna poskytnout řešení v požadované době.
2. Efekty plynoucí z použití expertního systému musejí převyšovat vynaložené náklady. To znamená, že by mělo jít o problém s opakovanou potřebou řešení a značnými finančními dopady, pro nějž lidští experti jsou drazí nebo omezeně dostupní.

Typické kategorie způsobů použití expertních systémů:

- *Konfigurace* – sestavení vhodných komponent systému vhodným způsobem.
- *Diagnostika* – zjištění příčin nesprávného fungování systému na základě výsledků pozorování.
- *Interpretace* – vysvětlení pozorovaných dat.
- *Monitorování* – posouzení chování systému na základě porovnání pozorovaných dat s očekávanými.
- *Plánování* – stanovení posloupnosti činností pro dosažení požadovaného výsledku.
- *Prognózování* – předpovídání pravděpodobných důsledků zadaných situací.
- *Ladění* – sestavení předpisu pro odstranění poruch systému.
- *Řízení* – regulace procesů (může zahrnovat interpretaci, diagnostiku, monitorování, plánování, prognózování a ladění).
- *Učení* – inteligentní výuka při níž studenti mohou klást otázky např. typu *proč*, *jak*, *co kdyby*.

Příklady aplikací expertních systémů je možno najít např. v knize (Mital a Anand 1994).

Výhody expertních systémů:

- schopnost řešit složité problémy,
- dostupnost expertíz a snížené náklady na jejich provedení,
- trvalost a opakovatelnost expertíz,
- trénovací nástroj pro začátečníky,
- uchování znalostí odborníků odcházejících z organizace.

Nevýhody expertních systémů:

- nebezpečí selhání ve změněných podmínkách,
- neschopnost poznat meze své použitelnosti.

1.6 Historie vývoje ES

Poté, co při řešení praktických problémů selhaly obecné metody řešení, byla pochopena nutnost využívat specifické (expertní) znalosti z příslušné problémové domény. Etapy vývoje je možno specifikovat takto:

1965-70	počáteční fáze (Dendral)
1970-75	výzkumné prototypy (MYCIN,PROSPECTOR, HEARSAY II)
1975-80	experimentální nasazování
1981-	komerčně dostupné systémy

Při charakteristice vývoje expertních systémů se také hovoří o dvou generacích. Charakteristické rysy prvé generace ES jsou tyto:

- jeden způsob reprezentace znalostí,
- malé schopnosti vysvětlování,
- znalosti pouze od expertů.

Druhá generace ES má následující charakteristiky:

- modulární a víceúrovňová báze znalostí,
- hybridní reprezentace znalostí,
- zlepšení vysvětlovacího mechanismu,
- prostředky pro automatizované získávání znalostí.

V rámci 2.generace ES se také objevují ***hybridní systémy***, v nichž se klasické paradigma expertních systémů kombinuje s dalšími přístupy, jako jsou neuronové sítě a evoluční metody.

2. Pravidlové expertní systémy

2.1 Reprezentace znalostí v pravidlových systémech

Znalosti v pravidlových systémech jsou reprezentována pomocí pravidel (*rules*), která mohou mít například takovéto tvary:

IF předpoklad THEN závěr

IF situace THEN akce

IF podmínka THEN závěr AND akce

IF podmínka THEN důsledek1 ELSE důsledek2

Část pravidla za IF (levá strana pravidla) se nazývá *antecedent*, podmínková část, nebo také část vzorů. Tato část může sestávat z individuálních podmínek, resp. vzorů. Část pravidla za THEN (pravá strana pravidla) se nazývá konsekvent a může také obsahovat několik akcí nebo závěrů. V předpokladové části se mohou vyskytnout spojky AND a OR, v důsledkové části se může vyskytnout spojka AND. Součástí pravidla může být také tzv. *kontext*, ve kterém má být pravidlo uvažováno.

Příklady pravidel:

IF auto_startuje = ne AND světla_svítí = ne THEN diagnóza = vybitá_baterie

IF barometr = stoupá THEN možnost_deště = nízká AND možnost_slunečna = vysoká

Pravidlo může také obsahovat neurčitosti. Např. v pravidle

IF výška_osoby(X) = velká THEN hmotnost_osoby(X) = velká

se jednak objevuje neurčitý pojem velká, jednak pravidlo samo je nemusí platit vždy. V rámci této kapitoly se však neurčitostmi nebudeme zabývat a budeme předpokládat deterministická pravidla.

Pravidlo IF p THEN q neznamená totéž, co implikace $p \Rightarrow q$. V expertním systému se provede akce q, je-li p splněno. Naproti tomu implikace je definována pravdivostní tabulkou a její význam může být v přirozeném jazyce vyjádřen řadou způsobů.

Pravidlo IF E THEN H se často zapisuje také ve tvaru

$$E \rightarrow H,$$

kde E představuje pozorování (*evidence*) a H znamená hypotézu (*hypothesis*).

Pravidlovým systémům se také říká produkční systémy. Produkční pravidla jsou formalismus, který byl původně jako přepisovací pravidla používán v teorii automatů, formálních gramatikách a při navrhování programovacích jazyků.

Většina znalostních systémů je založena na pravidlech, nebo kombinuje pravidla s jiným způsobem reprezentace. Pravidlové systémy se od klasických logických systémů odlišují nemonotonním uvažováním a možností zpracování neurčitosti. Neurčitost se může vyskytnout jednak v předpokladech pravidla, jednak se může týkat pravidla jako celku.

2.2 Inference v pravidlových systémech

Inference v pravidlových systémech je založena na pravidle *modus ponens*:

$$\frac{E,\ E \rightarrow H}{H}$$

To znamená, že jestliže platí předpoklad E a pravidlo $E \rightarrow H$, pak platí závěr H. Modus ponens (*ponere* znamená *tvrdit*) představuje přímé usuzování. Nepřímé usuzování je dáno pravidlem *modus tollens* (*tollere* znamená *popřít*):

$$\frac{\neg H,\ E \rightarrow H}{\neg E}$$

Řešení problému spočívá v nalezení řady inferencí (*inference chain*), které tvoří cestu od definice problému k jeho řešení. Existují dvě základní strategie v procesu usuzování:

- ***Usuzování řízené daty*** (dopředné řetězení, *forward chaining*):

  Při této strategii se začíná se všemi známými daty a postupuje se k závěru. Usuzování řízené daty je vhodné pro problémy zahrnující syntézu (navrhování, konfigurace, plánování, rozvrhování, …).

- ***Usuzování řízené cíli*** (zpětné řetězení, *backward chaining*):

  Tato strategie vybírá možný závěr a pokouší se dokázat jeho platnost hledáním dat, které jej podporují. Je vhodná pro diagnostické problémy, které mají malý počet cílových hypotéz.

I v pravidlových systémech můžeme rozlišovat mezi mělkým a hlubokým uvažováním, a to podle:

- délky inferenčního řetězce,
- podle kvality znalostí v pravidlech.

Například odvození závěru H z pravidel

$$A \rightarrow E \qquad C \wedge F \rightarrow G$$

$$B \wedge E \rightarrow F \qquad D \wedge G \rightarrow H$$

je hlubším usuzováním, než odvození H z pravidla

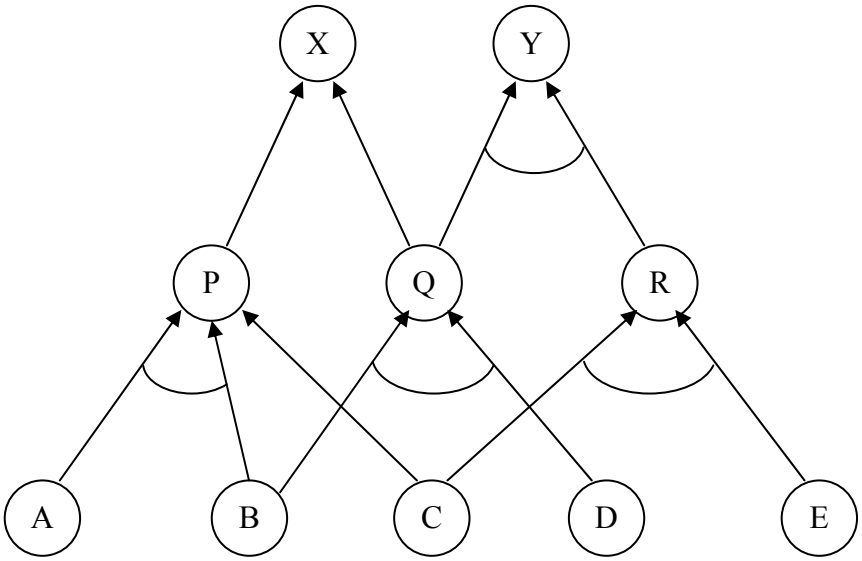
$$A \wedge B \wedge C \wedge D \rightarrow H$$

Pravidlový systém může být implementován dvěma základními způsoby a to jako ***inferenční síť*** (*inference network*), nebo jako ***systém porovnávání se vzorem*** (*pattern-matching system*).

V případě inferenční sítě jsou závěry pravidel fakta, která korespondují s předpoklady jiných pravidel. Inferenční síť může být reprezentována jako graf, jehož uzly jsou fakta a orientované hrany odpovídají pravidlům. Inferenční sítě jsou užitečné pro domény, kde počet možných řešení je limitován, jako jsou např. klasifikační nebo diagnostické problémy. Tyto systémy jsou však méně flexibilní. Inferenční sítě se snadněji implementují a snadněji se v nich zajišťuje vysvětlování. Na obr. 2.1 je ukázán příklad inferenční sítě pro následující systém pravidel:

$$A \wedge B \rightarrow P \qquad B \wedge D \rightarrow Q \qquad P \rightarrow X \qquad Q \wedge R \rightarrow Y$$
$$C \rightarrow P \qquad C \wedge E \rightarrow R \qquad Q \rightarrow X$$

Uvedená inferenční síť je příkladem tzv. AND/OR grafu (obloučky spojující hrany odpovídají operaci konjunkce).



Obr. 2.1. Příklad inferenční sítě

V systému porovnávání se vzorem jsou závěry pravidel obecnější a můžeme je chápat jako kolekce faktů, které mohou nebo nemusejí korespondovat se vzory popsanými v předpokladech jiných pravidel. Vztahy mezi fakty a pravidly se ustavují až při běhu na základě úspěšného porovnání faktů se vzory, které se nacházejí v levých částech pravidel. V případě shody všech vzorů v levé části pravidla s fakty v bázi faktů se mohou provést akce v pravé části pravidla (např. to může být zápis faktu do báze faktů nebo zrušení faktu v bázi faktů).

Systémy založené na porovnání se vzorem se vyznačují vysokou flexibilitou a schopností řešit problémy. Jsou spíše aplikovatelné v doménách, kde počet možných řešení je vysoký nebo neomezený, jako je navrhování, plánování a syntéza.V těchto doménách nejsou předdefinovány vztahy mezi fakty a pravidly. V systémech porovnávání se vzorem se hůře zajišťuje podpora rozhodování za neurčitosti. V rozsáhlých aplikacích hrozí snížení efektivnosti při vyhledávání aplikovatelných pravidel.

2.3 Dopředné řetězení

V algoritmu dopředného řetězení se opakují tři následující kroky:

1. ***Porovnání*** (*matching*):

   Pravidla ze znalostní báze jsou porovnávána se známými fakty, aby se zjistilo, u kterých pravidel jsou splněné předpoklady.

2. ***Řešení konfliktu*** (*conflict resolution*):

   Z množiny pravidel se splněnými předpoklady se vybírá pravidlo podle priority a v případě více pravidel se stejnou prioritou podle nějaké strategie. Příklady strategií řešení konfliktu:

   - strategie hledání do hloubky (*depth strategy*), při níž jsou preferována pravidla používající aktuálnější data (data, která se v bázi faktů vyskytují kratší dobu)
   - strategie hledání do šířky (*breath strategy*), při níž jsou preferována pravidla používající starší data
   - strategie složitosti resp. specifičnosti (*complexity strategy*) – preferována jsou speciálnější pravidla (pravidla mající více podmínek)
   - strategie jednoduchosti (*simplicity strategy*) – preferována jsou jednodušší pravidla

3. ***Provedení*** (*execution*):

   Provede se pravidlo vybrané v předchozím kroku. Důsledkem provedení pravidla může být přidání nového faktu do báze faktů, odstranění faktu z báze faktů, přidání pravidla do báze znalostí apod.

Obvykle je přitom uplatňována podmínka, že pravidlo může být aktivováno pouze jednou se stejnou množinou faktů.

Kromě pravidel se v bázi znalostí mohou vyskytovat také metapravidla. Rozdíl mezi nimi je v tom, že pravidla provádějí uvažování, kdežto metapravidla řídí uvažování.

Vhodnými aplikacemi pro dopředné řetězení jsou:

- Monitorování a diagnostika řídicích systémů pro řízení procesů v reálném čase, kde data jsou kontinuálně získávána a měněna a kde existuje málo předem určených vztahů mezi vstupními daty a závěry. V těchto aplikacích se z důvodu potřeby rychlé odezvy používá inferenční síť.
- Problémy zahrnující syntézu (navrhování, konfigurace, plánování, rozvrhování, …). V těchto aplikacích existuje mnoho potenciálních řešení a pravidla proto musejí vyjadřovat znalosti jako obecné vzory. Přesné vztahy (inferenční řetězce) tudíž nemohou být předem určeny a musejí být použity systémy porovnávání se vzorem.

2.4 Zpětné řetězení

Algoritmus zpětného řetězení sestává z těchto kroků:

1. Utvoř zásobník a naplň jej všemi koncovými cíli.
2. Shromáždi všechna pravidla schopná splnit cíl na vrcholu zásobníku. Je-li zásobník prázdný, pak konec.
3. Zkoumej postupně všechna pravidla z předchozího kroku.
   a) Jsou-li všechny předpoklady splněny, pak odvoď závěr (proveď pravidlo). Jestliže zkoumaný cíl byl koncový, pak jej odstraň ze zásobníku a vrať se na krok 2. Jestliže to byl podcíl (dílčí cíl), odstraň jej ze zásobníku a vrať se ke zpracování předchozího pravidla, které bylo dočasně odloženo.

b) Jestliže fakty nalezené v bázi faktů nesplňují předpoklady pravidla, je zkoumání pravidla ukončeno.

c) Jestliže pro některý parametr předpokladu chybí hodnota v bázi faktů, zjišťuje se, zda existuje pravidlo, z něhož by mohla být tato hodnota odvozena. Pokud ano, parametr se vloží do zásobníku jako podcíl, zkoumané pravidlo se dočasně odloží a přejde se na krok 2. V opačném případě se tato hodnota zjistí od uživatele a pokračuje se v kroku 3.a) zkoumáním dalšího předpokladu.

4. Jestliže pomocí žádného ze zkoumaných pravidel nebylo možné odvodit hodnotu důsledku, pak daný cíl zůstává neurčen. Odstraní se ze zásobníku a pokračuje se krokem 2.

Zpětné řetězení je vhodnější pro aplikace, mající mnohem více vstupů než možných závěrů. Dobrou aplikací pro zpětné řetězení je diagnostika, kde člověk komunikuje se znalostním systémem a zadává data pomocí klávesnice. Většina diagnostických systémů byla implementována pomocí inferenční sítě, protože vztahy mezi fakty jsou obvykle dobře známy. Ideální pro zpětné řetězení jsou rovněž klasifikační problémy. Tento typ aplikace může být implementován buď pomocí inferenční sítě nebo pomocí vzorů v závislosti na složitosti dat.

2.5 Algoritmus Rete

Dopředný systém porovnávání se vzorem je velmi neefektivní. V každém cyklu se musejí opakovaně porovnávat všechna pravidla se všemi fakty v bázi faktů. Podle odhadu až 90% času práce produkčního systému je věnováno opakovanému porovnávání se vzorem. Přitom po provedení pravidla většina báze faktů a tudíž také jejich účinků na pravidla zůstává nezměněna.

Rete je účinný porovnávací algoritmus redukující dobu porovnávání na základě síťové struktury, ve které jsou uloženy informace o ztotožnění podmínek s fakty v bázi faktů. Tento algoritmus používá následující síťovou strukturu:

- Uzly sítě jsou startovací uzel a dále jeden uzel pro každou podmínku a konjunkci podmínek. Konjunktivní uzly s výstupním stupněm 0 korespondují s pravidly. S každým uzlem je spojena množina faktů, se kterými je podmínka ztotožněna.
- Hrany sítě jsou označeny vazbami nebo vztahy proměnných vyskytujících se v počátečním uzlu hrany.

Fakt, který je přidáván do báze faktů (resp. rušen v bázi faktů), je reprezentován příznakem. Tento příznak je zpočátku umístěn ve startovacím uzlu, a odtud se pak šíří po síti. Příznak může projít hranou, jestliže jeho argumenty splňují vztah spojený s hranou. Jestliže v uzlu, odpovídajícím pravidlu, všechny příznaky splňují podmínky pravidla, pak se pravidlo přidá do konfliktní množiny (resp. odstraní z konfliktní množiny).

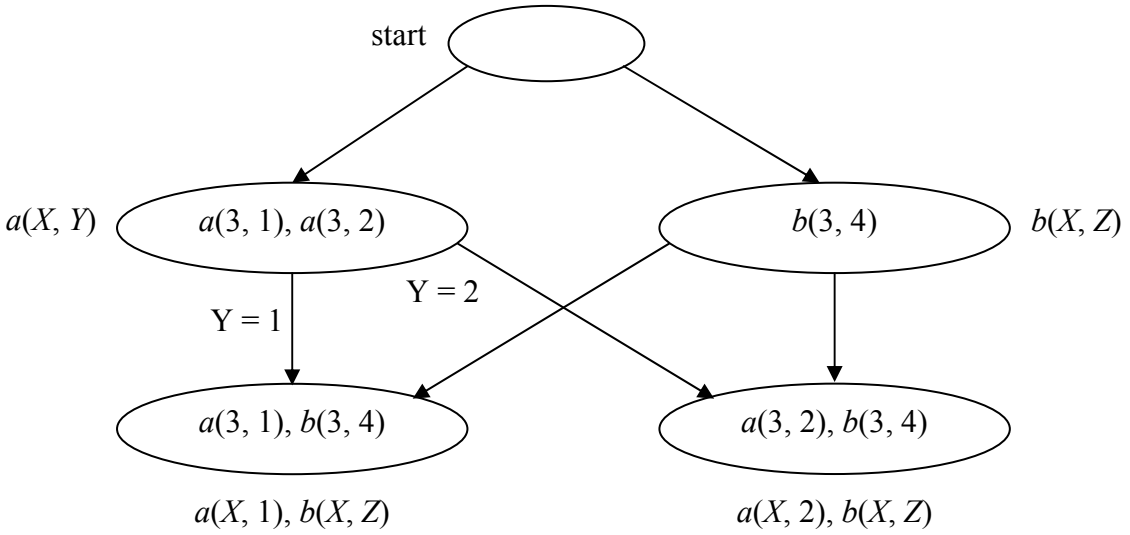
Příklad síťové struktury pro algoritmus Rete:

Předpokládejme, že máme v bázi znalostí dvě pravidla:

IF $a(X, 1)$ AND $b(X, Z)$ THEN $g1(X, Z)$

IF $a(X, 2)$ AND $b(X, Z)$ THEN $g2(X, Z)$

Pak odpovídající strukturu ukazuje obr. 2.2. Tento obrázek odpovídá situaci, že v bázi faktů jsou fakty $a(3, 1)$, $a(3, 2)$, $b(3, 4)$.



Obr. 2.2. Příklad síťové struktury pro algoritmus Rete

Popišme nyní kroky vedoucí ke stavu na obr. 2.2.

1. Předpokládejme, že na počátku je báze faktů prázdná. Pak také všechny uzly této struktury jsou prázdné.
2. Nechť se v bázi faktů objeví fakt $a(3, 1)$. Pak bude umístěn do uzlu označeného symbolem $a(X, Y)$ a bude postupovat dál po hraně s podmínkou $Y = 1$, takže se dostane do uzlu s označením $a(X, 1)$, $b(X, Z)$. Odpovídající pravidlo však ještě nebude přidáno do konfliktní množiny, protože není k dispozici žádný fakt, který lze unifikovat s $b(3, Z)$.
3. Jestliže je do báze faktů přidán fakt $b(3, 4)$, objeví se v uzlu označeném symbolem $b(X, Z)$ a po hranách se dostane do uzlů s označením $a(X, 1)$, $b(X, Z)$ a označením $a(X, 2)$, $b(X, Z)$. Prvý argument příznaku $b(3, 4)$ se shoduje s prvým argumentem příznaku $a(3, 1)$, takže prvé pravidlo bude přidáno do konfliktní množiny.
4. Obdobně přidání faktu $a(3, 2)$ do báze faktů povede k jeho umístění do uzlů s označením $a(X, Y)$ a označením $a(X, 2)$, $b(X, Z)$ a následně k přidání druhého pravidla do konfliktní množiny.

Pokud je nějaký fakt z báze faktů odstraněn, pak se jeho příznak podobně šíří po síti, přičemž se mažou všechny jeho kopie vyskytující se v uzlech a z konfliktní množiny jsou pak také odstraněna odpovídající pravidla. Např. odstranění faktu $b(3, 4)$ z báze faktů by vedlo k odstranění obou pravidel z konfliktní množiny.

2.6 Příklady pravidlových systémů

ART	prázdný ES založený na Lispu, dopředné řetězení, algoritmus Rete
CLIPS	programové prostředí, dopředné řetězení, algoritmus Rete
EXSYS	prázdný expertní systém, dopředné a zpětné řetězení
M.4	programové prostředí, dopředné a zpětné řetězení, porovnávání se vzorem
ILOG-RULES	programové prostředí, dopředné řetězení, algoritmus Xrete
OPS5	programové prostředí, dopředné řetězení, algoritmus Rete

2.7 Výhody a nevýhody pravidlových systémů

Výhody:

- modularita,
- uniformita,
- přirozenost.

Možné nevýhody a problémy:

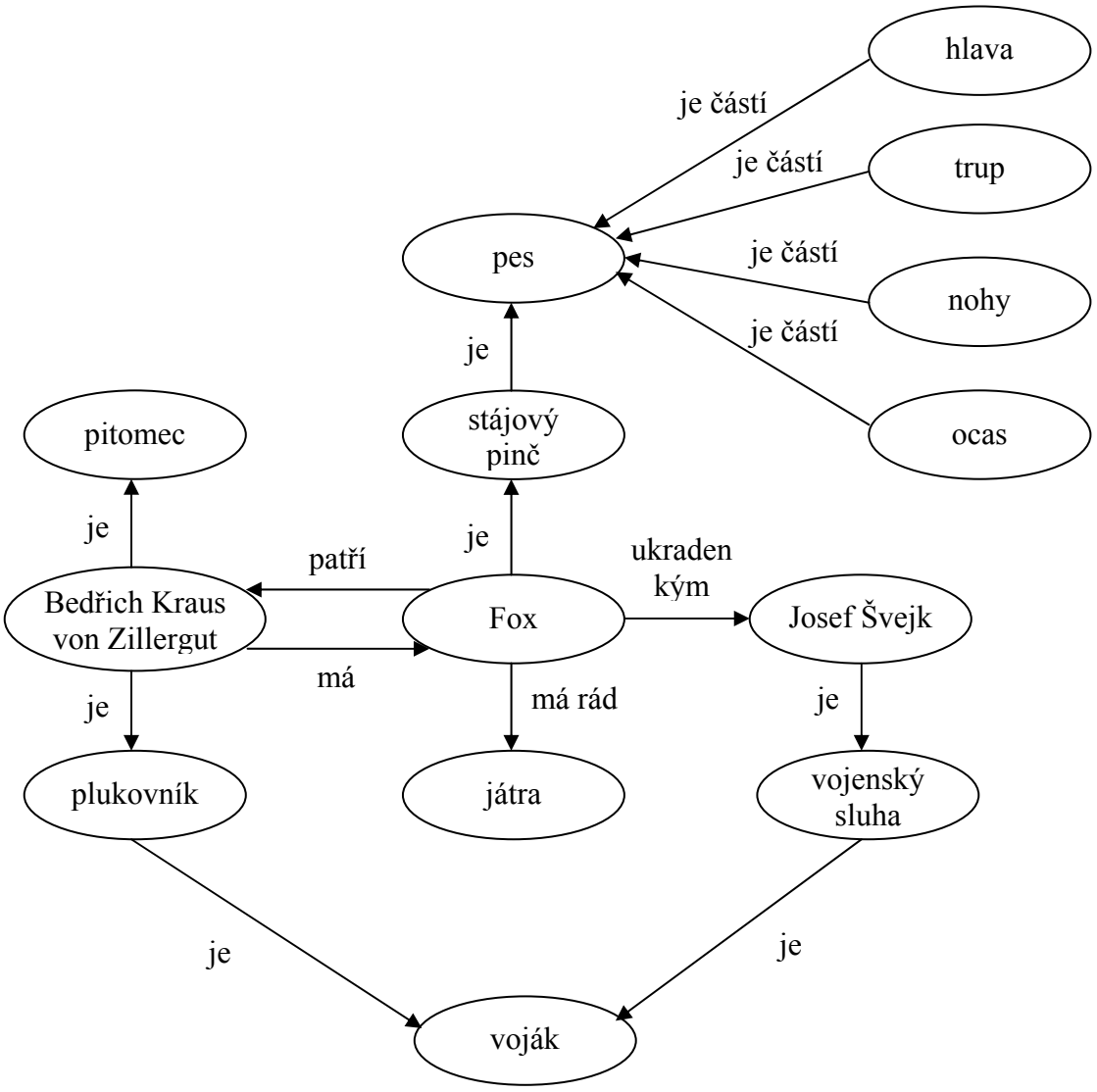
- nebezpečí nekonečného řetězení,
- přidání nové rozporné znalosti,
- modifikace existujících pravidel,
- neefektivnost,
- neprůhlednost,
- pokrytí domény (existují domény vyžadující příliš mnoho pravidel).

3. Nepravidlové a hybridní expertní systémy

Mezi nepravidlové reprezentace znalostí patří např. *rozhodovací stromy*, *sémantické sítě*, ***Petriho sítě***, ***rámce*** a ***objekty***.

3.1 Sémantické sítě

Sémantická síť (*semantic net*) je ohodnocený orientovaný graf. Uzly reprezentují objekty a hrany představují vztahy mezi objekty. Místo pojmu sémantická síť se také používá pojem *asociativní síť*. Sémantická síť poskytuje vyšší úroveň porozumění akcím, příčinám a událostem, které se vyskytují v odpovídající doméně. To umožňuje úplnější usuzování znalostního systému o problémech z této domény. Sémantická síť umožňuje reprezentaci fyzikálních, kauzálních a taxonomických vztahů.



Obr. 3.1. Příklad sémantické sítě

Ukázka sémantické sítě je uvedena na obrázku 3.1. Jsou zde zachyceny vztahy *je*, *je částí*, *má*, *patří*, *má rád*, *ukraden kým*. Zejména vztahy *je* (*is-a*, ISA), *je částí* (*part-of*), *má* (*has-a*) se v sémantických sítích vyskytují velmi často. Je nutné si dát pozor na interpretaci vztahu *is-a*, který může mít např. tyto významy: *je instancí, je prvkem, je podmnožinou, je podtřídou, je ekvivalentní s.* Např. *Fox* je instancí třídy *stájový pinč* a třída *stájový pinč* je podtřídou třídy *pes*. V některých systémech a knihách se *is-a* používá ve významu *je instancí*, zatímco vztah mezi třídami se označuje jako *a-kind-of* (AKO). Ale např. v systému ART *is-a* označuje vtah mezi třídami a pro označení vztahu *je instancí* se užívá *instance-of*.

Sémantická síť podporuje dědičnost a tranzitivitu. Např. třída *stájový pinč* dědí všechny vlastnosti třídy *pes* a tyto vlastnosti se tedy předpokládají také u všech jejích instancí. Příkladem tranzitivního uvažování je tato úvaha: Bedřich Kraus von Zillergut je plukovník, plukovník je voják a tedy Bedřich Kraus von Zillergut je voják.

Výhody sémantické sítě:

- explicitní a jasné vyjádření,
- redukce doby hledání (pro dotazy typu dědičnosti nebo rozpoznávání).

Nevýhody:

- neexistence standardních definic jmen vazeb,
- neexistence interpretačních standardů,
- nebezpečí chybné inference,
- nebezpečí kombinatorické exploze (zvláště tehdy, je-li odpověď na otázku negativní).

3.2 Rámce

Rámce (*frames*) jsou struktury pro reprezentaci stereotypních situací a odpovídajících stereotypních činností (scénářů). Tento prostředek reprezentace vychází z poznatku, že lidé používají pro analyzování a řešení nových situací rámcové struktury znalostí získaných na základě předchozích zkušeností.

Rámce se pokoušejí reprezentovat obecné znalosti o třídách objektů, znalosti pravdivé pro většinu případů. Mohou existovat objekty, které porušují některé vlastnosti popsané v obecném rámci. Rámce jsou preferovaným schématem reprezentace v modelovém a případovém usuzování (*model-based reasoning*, *case-based reasoning*). Pro reprezentaci rámců se používají speciální jazyky, jako např. KRYPTON, FRL, KSL.

Rámec je tvořen jménem a množinou *atributů*. Atribut (*rubrika*, *slot*) může dále obsahovat položky (*links*, *facets*), jako např. aktuální hodnotu (*current*), implicitní hodnotu (*default*), rozsah možných hodnot (*range*).

Dalšími položkami slotu mohou být speciální procedury, jako např. *if-needed*, *if-changed*, *if-added*, *if-deleted*. Tyto procedury jsou automaticky aktivovány, jestliže nastanou příslušné situace. Např. v systému FLEX se používají následující typy událostmi řízených procedur:

- *launches* (aktivují se při vytváření instance rámce)

- *watchdogs* (aktivují se při přístupu k aktuální hodnotě slotu)
- *constraints* (aktivují se před změnou hodnoty slotu)
- *demons* (aktivují se po změně hodnoty slotu)

Příklady rámců:

Rámec Majetek

Jméno:	majetek
Specializace čeho:	objekt
Typ:	rozsah: (auto, loď, dům)
	if-added: procedure PŘIDEJ_MAJETEK
Vlastník:	if-needed: procedure NAJDI_VLASTNÍKA
Umístění:	rozsah: (doma, práce, mobilní)
Stav:	rozsah: (chybí, špatný, dobrý)
Záruka:	rozsah: (ano, ne)

Rámec Auto

Jméno:	auto
Specializace čeho:	majetek
Typ:	rozsah: (sedan, sportovní_vůz)
Výrobce:	rozsah: (GM, Ford, Chrysler)
Umístění:	mobilní
Kola:	4
Převodovka:	rozsah: (manuální, automatická)
Motor:	rozsah: (benzínový, naftový)

Rámec Janovo auto

Jméno:	Janovo_auto
Specializace čeho:	auto
Typ:	sportovní_vůz
Výrobce:	GM
Vlastník:	Jan Chodec
Převodovka:	automatická
Motor:	benzínový
Stav:	dobrý
Záruka:	ano

Rámce mohou být *generické* nebo *specifické*. Příkladem generických rámců jsou rámce *Majetek* a *Auto*, zatímco rámec *Janovo auto* je specifický. Rámec *Auto* reprezentuje celou třídu aut, kdežto v případě rámce *Janovo auto* se jedná o určité konkrétní auto.

Mezi rámci mohou existovat vztahy dědičnosti, které umožňují distribuovat informace bez nutnosti jejich zdvojování. Rámec může být specializací jiného obecnějšího rámce (vztah typu *specialization-of*) a současně může být zobecněním jiných rámců (vztah typu *generalization-of*). Vztah *specialization-of* může mít podobu *a-kind-of* (rámec je podrámcem jiného rámce) nebo *an-instance-of* (rámec je instancí jiného rámce). Ve výše uvedeném příkladu je rámec *Majetek*

podrámcem rámce *Objekt*, rámec *Auto* je podrámcem rámce *Majetek* a rámec *Janovo_auto* je instancí rámce *Auto*.

V systému FLEX existují např. tyto vztahy mezi rámci:

- Rodič – potomek (*is-a*, *is-a-kind-of*):

  Tento vztah může být typu 1:1, 1:*n*, *n*:1. Dědění některého atributu může být pro určitý rámec potlačeno.

- Rámec – instance rámce (*is-an-instance-of*).

  Tento vztah je typu 1:1. Přitom je navíc možné dědění nějakého specifického atributu od nějakého specifického rámce.

- Vlastnictví rámce (atributem rámce může být jiný rámec).

Výhody rámců:

- snazší usuzování řízené očekáváním (na základě využití démonů),
- organizace znalostí (větší strukturovanost a organizace než v sémantických sítích),
- samořízení (schopnost rámců určit svou vlastní aplikovatelnost v dané situaci),
- uchovávání dynamických hodnot (ve slotech rámců); výhodné při simulaci, plánování, diagnostice, … .

Možné nevýhody:

- potíže s odlišností objektů od prototypu,
- obtížné přizpůsobení novým situacím,
- obtížný popis detailních heuristických znalostí.

3.3 Objekty

Objekty podobně jako rámce sdružují deklarativní znalosti a procedurální znalosti. Objekt je programová struktura, obsahující jak *data*, tak *metody* (procedury), které s těmito daty pracují. Data objektu jsou přístupná pouze prostřednictvím metod objektu. Tato vlastnost se označuje jako *zapouzdření* (*encapsulation*) . Objekt je *instance třídy*. Třída je skupina objektů, které mají stejné vlastnosti (datové složky) a stejné chování (metody).

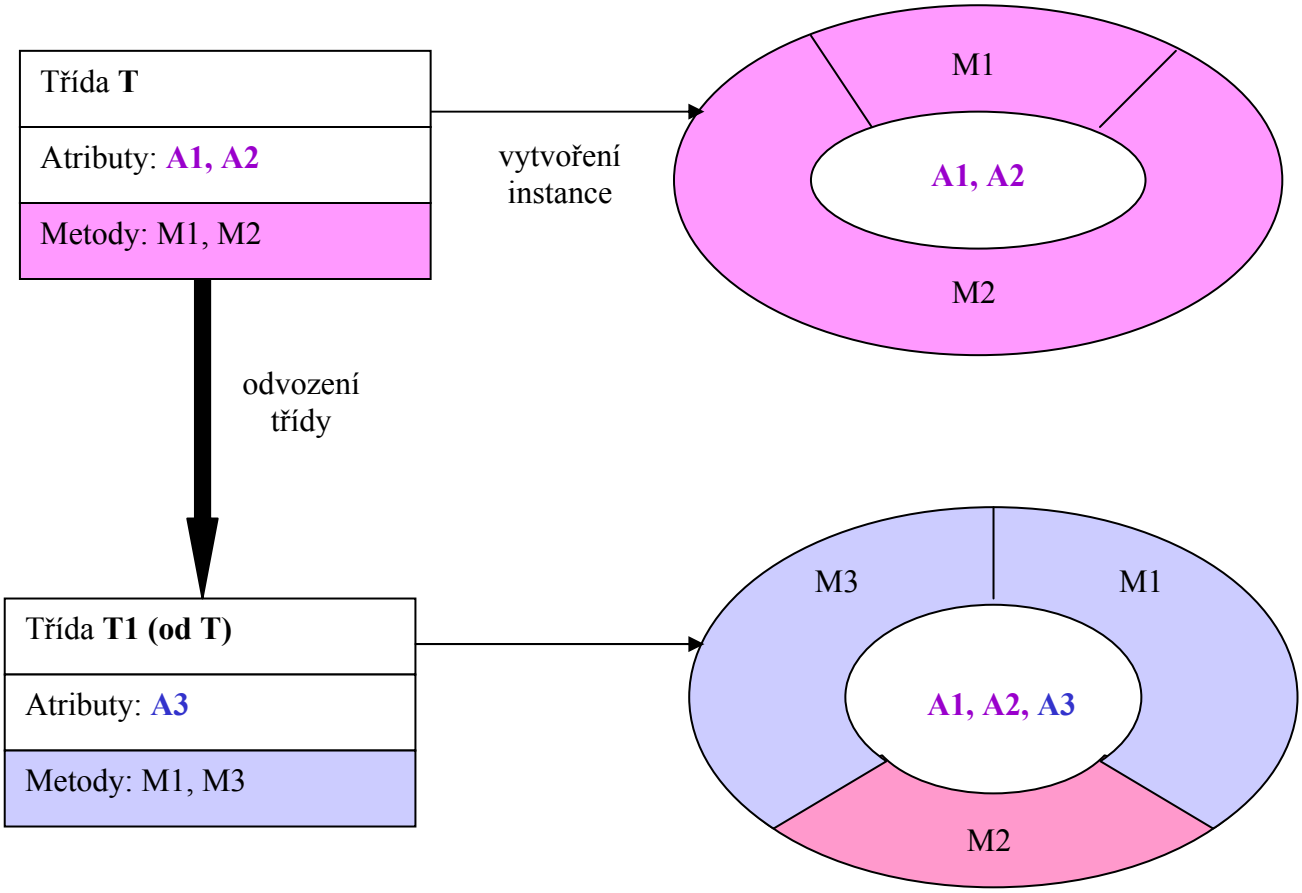
Mezi třídami mohou existovat tyto typy vztahů:

- Dědičnost:

  Od jedné třídy (bázové, rodičovské) můžeme odvodit třídu jinou (odvozenou, dceřinnou). Dceřinná třída dědí všechny složky své rodičovské třídy a k nim může přidat svoje vlastní. Zděděné metody je možno předefinovat. Objekt třídy *předek* může být v programu zastoupen objektem třídy *potomek* (může se jednat i o nepřímého potomka). Tento typ vztahu je ilustrován obrázkem 3.2.

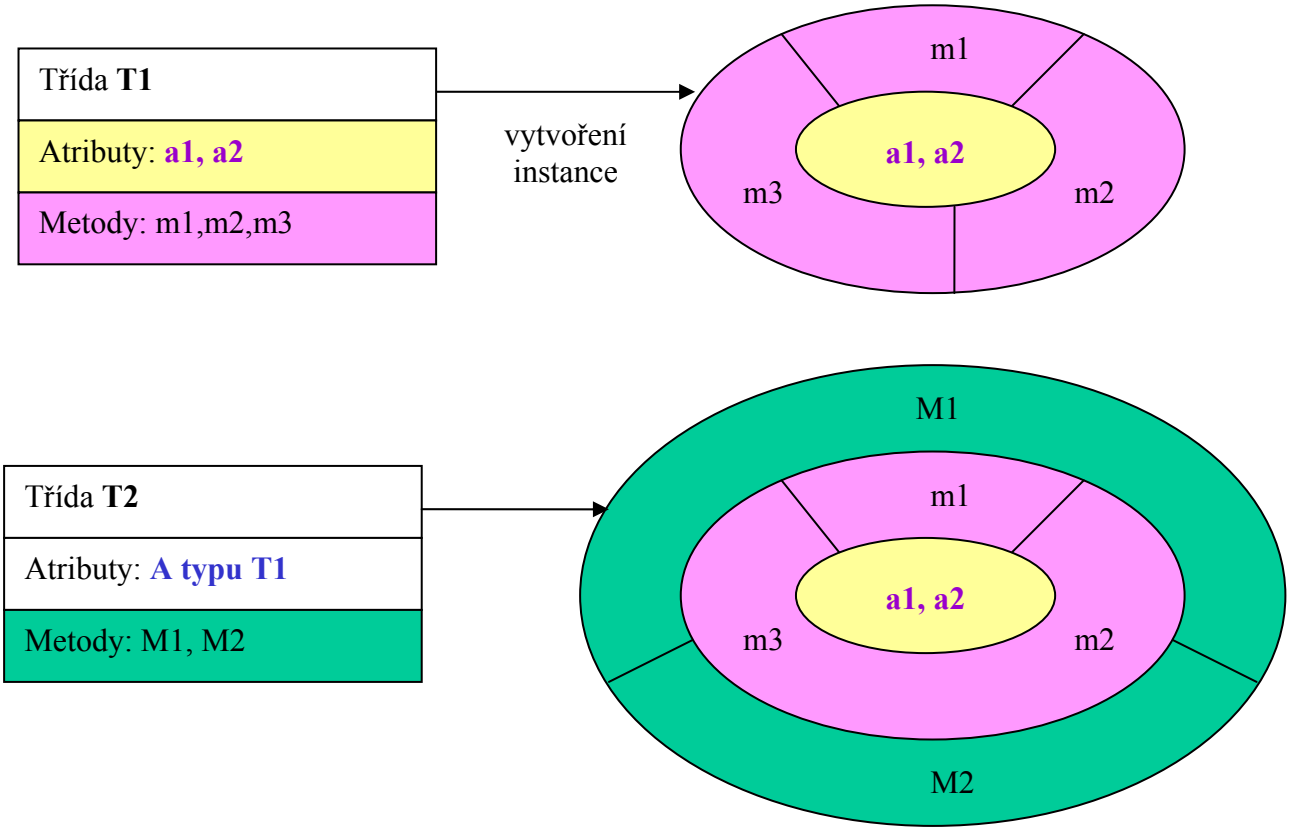
- Vlastnictví (skládání) :

  Složkou třídy může být jiná třída. Skládání objektů ilustruje obr. 3.3.



Obr 3.2. Třídy, objekty a dědičnost

Na obr. 3.2 třída *T*1 dědí od třídy *T* atributy *A*1 a *A*2 a metodu *M*2. Dále definuje atribut *A*3, metodu *M*3 a předefinovává metodu *M*1.



Obr 3.3. Skládání tříd a objektů

Objekty spolu komunikují tak, že si navzájem posílají zprávy; obvykle to znamená, že jeden objekt (odesilatel zprávy) volá metodu jiného objektu (příjemce zprávy). Existují dva způsoby určení příjemce zprávy:

- časná vazba (*early binding*) – příjemce zprávy je určen v okamžiku kompilace.
- pozdní vazba (*late binding*) – příjemce zprávy je určen až za běhu programu; pomocí pozdní vazby se realizuje *polymorfismus* (mnohotvarost).

Jazyky pro objektové programování rozdělujeme na:

- jazyky čistě objektové (Smalltalk, Actor, CLOS, …)
- jazyky podporující OOP, ale umožňující programovat i neobjektově (Borland Pascal, Object Pascal, C++, …)

Výhody objektů:

- abstrakce
- zapouzdření (ukrývání informace)
- dědičnost
- polymorfismus
- znovupoužitelnost kódu

Nevýhody (podobné jako u rámců):

- jak se vypořádat s odchylkami od normy?
- jak zohlednit nové dosud neuvažované situace?

3.4 Hybridní systémy

Zatímco u 1.generace expertních systémů byl v rámci jednoho systému používán pouze jeden způsob reprezentace znalostí, expertní systémy 2.generace obvykle používají hybridní (kombinované) reprezentace znalostí. Hybridní reprezentace, které jsou dostupné v nejčastěji používaných prostředcích pro vývoj ES, kombinují pravidlově, rámcově a objektově orientované techniky. Umožňují tzv. *modelový* přístup k tvorbě systému a usnadňují vývoj modelů.

Hybridní systémy, kombinující rámce a pravidla, používají např. rámce pro reprezentaci strukturálních znalostí a pravidla pro usuzování o těchto znalostech. Rámce také mohou být využity pro implementaci sémantických sítí.

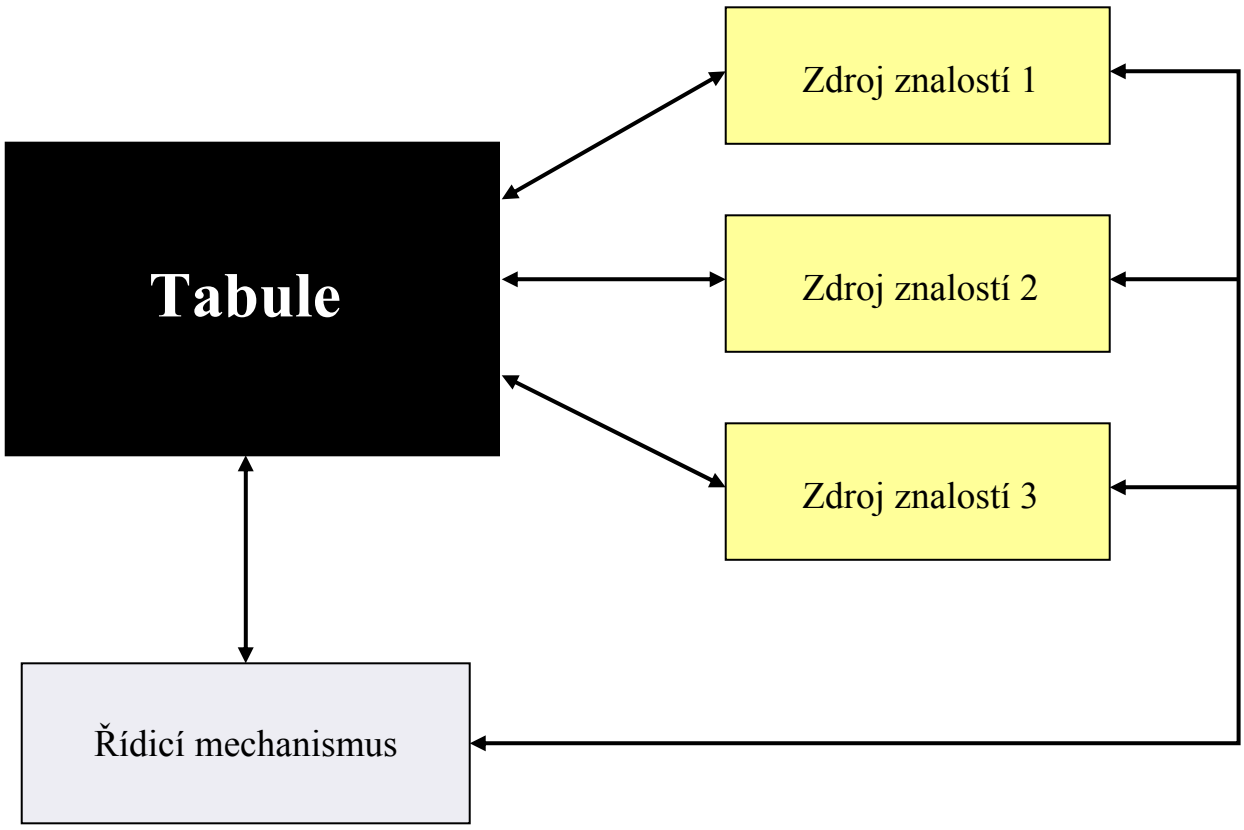
Mezi hybridní systémy patří např.:

ACQUIRE	prázdný ES, pravidla, rámce a objekty
CLIPS	programové prostředí, objekty a pravidla
FLEX	programové prostředí implementované v Prologu, pravidla a rámce, dopředné a zpětné řetězení
G2	objektově orientované prostředí, pravidla, modely a procedury, diagnostické a řídící aplikace
Rete++	programové prostředí, pravidla a objekty, dopředné a zpětné řetězení
Rtworks	programové prostředí, objekty a pravidla
XpertRule	programové prostředí, rozhodovací stromy a tabulky příkladů

3.5 Systémy typu tabule

Architektura typu tabule (*blackboard architecture*) je příkladem implementace tzv. oportunistického usuzování (*opportunistic reasoning*, *opportunistic problem solving*). Při tomto způsobu znalosti nejsou striktně aplikovány v přímém nebo zpětném usuzování, ale jsou používány v nejpříhodnější době nejvhodnějším způsobem. Metoda usuzování je volena dynamicky v závislosti na tom, co systém naposledy zjistil. Tato forma usuzování je vhodná v aplikacích, kde znalosti o řešení problémů mohou být rozčleněny do nezávislých modulů, které pak kooperují při řešení problému.

Práci systému typu tabule je možno charakterizovat následujícím příkladem. Experti jsou ve třídě s tabulí. Komunikovat mohou pouze písemně přes tabuli a k dispozici mají pouze jeden kousek křídy. Řídicí mechanismus sleduje myšlenky expertů, vyhodnocuje možné příspěvky a rozhoduje, kdo dostane křídu.



Obr 3.4. Struktura systému typu tabule

Strukturu systému typu tabule ukazuje obr. 3.4. Hlavní komponenty jsou tyto:

- ***Zdroje znalostí***: Obsahují dílčí znalosti potřebné pro řešení dílčích problémů (mohou to být individuální znalostní systémy); jsou přípustné různé reprezentace znalostí.
- ***Tabule***: Společná databáze, přes niž zdroje znalostí komunikují.
- ***Řídicí mechanismus***: Koordinuje zdroje znalostí; doporučuje akce, které mohou provést; určuje, které zdroje jsou nejvhodnější k tomu, aby přispěly k nalezení řešení; rozhoduje, co je aktuálně v popředí zájmu.

Činnost systému typu tabule probíhá v následujících krocích:

1. Zdroj znalostí provádí nějakou změnu na tabuli. Záznam o těchto změnách je zapsán do oblasti řídicích dat.
2. Každý zdroj znalostí zkoumá relevantní informace na tabuli, určuje, které akce by mohl provést a navrhuje tyto akce řídicímu mechanismu.
3. Řídicí mechanismus zkoumá informace z předchozích dvou kroků a určuje ohnisko zájmu.
4. Řídicí mechanismus vybere zdroj znalostí a objekt tabule. Systém se vrací na krok 1.

   Kritéria ukončení jsou zajišťována při vytváření systému. Obvykle jsou zabudována do jednoho ze zdrojů znalostí.

Příkladem systému typu tabule je systém GBB (*Generic Blackboard Builder*), což je objektově orientované programové prostředí pro vývoj systémů typu tabule. GBB byl vytvořen v jazyku Common Lisp. GBB je používán nejen pro vývoj nových aplikací, ale také pro integrování existujících aplikací jako komponent sofistikovanějších aplikací. GBB představuje otevřené a rozšiřitelné prostředí, kde moduly mohou být psány v jakémkoli jazyku. K dalším rysům systému GBB patří:

- velmi účinné prostředky pro vyhledávání objektů,
- prostředky pro inteligentní řízení a integraci,
- grafické rozhraní pro tvorbu, ladění a používání komponent.

Systémy typu tabule jsou vhodné pro diverzifikované problémy (různé formy vstupních dat, nejasně definované cíle, použití mnohonásobných linií uvažování) a pro distribuovaná prostředí. K dalším výhodám patří hierarchická organizace, datová abstrakce, možnost odložení rozhodnutí, volné seskupování znalostí a jejich užití. Nevýhodami těchto systémů je to, že jsou drahé pro vytváření a používání a že je obtížné určit vhodné rozčlenění znalostí.

4. Zpracování neurčitosti

4.1 Neurčitost v expertních systémech

Neurčitost je charakteristickým rysem složitých systémů. Vlastní povaha reality způsobuje, že poznatky, které z ní získáváme, jsou neurčité či vágní. Příčinami neurčitosti jsou:

- problémy s daty; např.:
  - chybějící nebo nedostupná data
  - nespolehlivá data (např. z důvodu chyb měření)
  - nepřesná nebo nekonzistentní reprezentace dat
- nejisté znalosti; např.:
  - znalost nemusí být platná ve všech případech
  - znalost může obsahovat vágní pojmy.

Neurčitost bývá v ES vyjadřována obvykle numerickými parametry, které se v různých systémech nazývají různě, např. *váhy*, *míry*, *stupně důvěry*, *faktory jistoty*. Tyto numerické parametry se přiřazují jednotlivým tvrzením nebo pravidlům. Často nabývají hodnot z intervalu $\langle 0,1 \rangle$ nebo $\langle -1,1 \rangle$.

Většinou se neurčitost vyjadřuje pomocí jediného čísla. Postupně se však začínají prosazovat přístupy, v nichž je neurčitost vyjadřována dvojicí čísel (tato dvojice může být např. interpretována jako interval hodnot). Existují také systémy, kde se pracuje s kvalitativně vyjádřenými neurčitostmi.

Přístupy ke zpracování neurčitosti můžeme rozdělit do dvou základních skupin:

- přístupy založené na *ad hoc* modelech, použitých např. v:
  - systému MYCIN (jsou zde použity faktory jistoty),
  - systému PROSPECTOR (pseudobayesovské přístupy).
- přístupy založené na teoretických principech, např. na:
  - teorii pravděpodobnosti,
  - teorii fuzzy množin,
  - teorii fuzzy míry (sem patří např. Dempster-Shaferova teorie a teorie možnosti).

Omezme se v této kapitole na problematiku pravidlových systémů. Při zpracování neurčitosti se zde střetáváme s následujícími problémy (problémy aproximativní inference):

- Jak kombinovat neurčitá data v předpokladu pravidla?
- Jak kombinovat neurčitost předpokladu pravidla a neurčitost pravidla jako celku?
- Jak stanovit neurčitost závěru, k němuž vede několik pravidel?

4.2 Bayesovský přístup

Bayesovský přístup je nejstarší a nejlépe definovanou technikou pro zpracování neurčitosti. Uvažujme znalost ve tvaru pravidla $E \rightarrow H$, které říká, že předpoklad (pozorování, *evidence*) E podporuje závěr (hypotézu, *hypothesis*) H. Neurčitost závěru H v závislosti na předpokladu E může být kvantifikována pomocí podmíněné pravděpodobnosti $P(H \mid E)$, která je dána podle *Bayesových vzorců* takto:

$$P(H \mid E) = \frac{P(E \mid H)P(H)}{P(E)}$$

$$P(H \mid E) = \frac{P(E \mid H)P(H)}{P(E \mid H)P(H) + P(E \mid \neg H)P(\neg H)}$$

Dále můžeme definovat apriorní a aposteriorní pravděpodobnostní šanci následovně:

Apriorní pravděpodobnostní šance:

$$O(H) = \frac{P(H)}{P(\neg H)} = \frac{P(H)}{1 - P(H)}$$

Aposteriorní pravděpodobnostní šance:

$$O(H \mid E) = \frac{P(H \mid E)}{P(\neg H \mid E)} = \frac{P(H \mid E)}{1 - P(H \mid E)}$$

Pravděpodobnost lze ze šance vypočítat podle vztahu

$$P = \frac{O}{O + 1}$$

Z Bayesových vzorců pro $P(H \mid E)$ a $P(\neg H \mid E)$ plyne, že

$$O(H \mid E) = L \cdot O(H)$$

kde

$$L = \frac{P(E \mid H)}{P(E \mid \neg H)}$$

L se nazývá *mírou postačitelnosti* (velká hodnota $L >> 1$ říká, že předpoklad E je postačitelný k dokázání hypotézy H).

Obdobně platí

$$O(H \mid \neg E) = \hat{L} \cdot O(H)$$

kde

$$\hat{L} = \frac{P(\neg E \mid H)}{P(\neg E \mid \neg H)}$$

je *míra nezbytnosti* (malá hodnota $0 < \hat{L} << 1$) znamená, že E je nezbytné pro dokázání H).

4.3 Prospectorovské systémy (pseudobayesovský přístup)

Míry postačitelnosti a nezbytnosti zadává pro každé pravidlo expert jako svoje subjektivní váhy. Pravidlo $E \to H$ se vlastně chápe jako pravidlo

if E then H with váha L else H with váha $L^{\wedge}$,

resp. jako dvojice pravidel

$$E \to H\,(L) \text{ a } \neg E \to H\,(L^{\wedge}) .$$

Místo uvedených měr může expert zadat pravděpodobnosti $P(H \mid E)$ a $P(H \mid \neg E)$, pomocí nichž se pak tyto míry vypočtou. Např.

$$L = \frac{P(H \mid E)}{1 - P(H \mid E)} \cdot \frac{1 - P(H)}{P(H)}$$

Předpokládejme nyní, že k výroku E není přiřazena logická hodnota pravda nebo nepravda, ale je k dispozici pouze nějaké relevantní pozorování E'. Pak platí

$$\begin{aligned} P(H \mid E') &= P(H \wedge E \mid E') + P(H \wedge \neg E \mid E') = \\ &= P(H \mid E \wedge E')P(E \mid E') + P(H \mid \neg E \wedge E')P(\neg E \mid E') \end{aligned}$$

Můžeme udělat tento oprávněný předpoklad: Víme-li, že E je pravda nebo nepravda, pak pozorování E' nepřináší žádnou další informaci o H. Potom můžeme předchozí vztah přepsat do tvaru

$$\begin{aligned} P(H \mid E') &= P(H \mid E)P(E \mid E') + P(H \mid \neg E)P(\neg E \mid E') = \\ &= P(H \mid \neg E) + \left[P(H \mid E) - P(H \mid \neg E)\right]P(E \mid E') \end{aligned}$$

Vztah

$$P(H \mid E') = P(H \mid \neg E) + \left[P(H \mid E) - P(H \mid \neg E)\right]P(E \mid E')$$

představuje lineární závislost $P(H \mid E')$ na $P(E \mid E')$.

Protože však expert nezávisle zadává $P(H \mid E)$ a $P(H \mid \neg E)$ (přímo nebo prostřednictvím L a $L^{\wedge}$) a dále $P(H)$ a $P(E)$, je uvedená přímka přeurčena a může dojít k rozporu (zadávané údaje nemusejí být konzistentní). Proto se výše uvedený teoretický vztah nahrazuje nějakou aproximací. Obvykle se fixují tyto tři body grafu:

$[0, P(H \mid \neg E)]$, $[P(E), P(H)]$ a $[1, P(H \mid E)]$.

Mezi těmito body se pak závislost interpoluje lineárními funkcemi.

Uvažujme např. případ $P(H \mid \neg E) \le P(H) \le P(H \mid E)$. Pak můžeme $P(H \mid E')$ aproximovat pomocí vztahů

$$P(H \mid E') = P(H \mid \neg E) + \frac{P(H) - P(H \mid \neg E)}{P(E)} \cdot P(E \mid E')$$

pro $0 \le P(E \mid E') \le P(E)$ a

$$P(H \mid E') = P(H) + \frac{P(H \mid E) - P(H)}{1 - P(E)} \cdot \big(P(E \mid E') - P(E)\big)$$

pro $P(E) \le P(E \mid E') \le 1$.

Problém kombinace více pravidel se stejným závěrem je v prospectorovských systémech řešen následujícím způsobem. Mějme pravidla $E_1 \to H$, $E_2 \to H$, ... , $E_n \to H$. Pak se aposteriorní šance za předpokladu nezávislosti evidencí E_i vypočte takto:

$$O(H \mid E_1 \wedge ... \wedge E_n) = L_1 \cdot ... \cdot L_n \cdot O(H)$$

Pokud místo přesných evidencí E_i jsou k dispozici pouze pozorování E_i' , pak se aposteriorní šance vypočte podle vztahu

$$O(H \mid E_1' \wedge ... \wedge E_n') = L_1' \cdot ... \cdot L_n' \cdot O(H)$$

kde

$$L_i' = \frac{O(H \mid E_i')}{O(H)}$$

Problém kombinace předpokladů řeší prospectorovské systémy tak, že vztahy pro výpočet vah disjunkce, konjunkce a negace předpokladů přebírají z fuzzy logiky.

Disjunkce předpokladů:

$$P(E_1 \vee E_2) = \max\{P(E_1), P(E_2)\}$$

Konjunkce předpokladů:

$$P(E_1 \wedge E_2) = \min\{P(E_1), P(E_2)\}$$

Negace předpokladu:

$$P(\neg E) = 1 - P(E)$$

Výhody bayesovských přístupů:

- dobré teoretické základy
- dobře definovaná sémantika rozhodování

Nevýhody:

- potřeba velkého množství pravděpodobnostních dat
- nebezpečí neúplnosti a nekonzistence dat
- předpoklad nezávislosti evidencí E_i bývá v praxi zřídka splněn
- možnost ztráty informace v důsledku popisu neurčitosti jedním číslem
- obtížnost vysvětlování

4.4 Přístup založený na faktorech jistoty

Faktory jistoty (*certainty factors*) byly poprvé použity v systému MYCIN. Cílem bylo eliminovat některé slabiny čistě pravděpodobnostního přístupu. Znalosti jsou vyjádřeny opět ve tvaru pravidel $E \rightarrow H$, přičemž s každým pravidlem je spojen faktor jistoty *CF*. Tento faktor nabývá hodnot z intervalu $\langle -1,1 \rangle$ a je určen pomocí měr důvěry a nedůvěry *MB* a *MD*:

$$CF = \frac{MB - MD}{1 - \min\{MB, MD\}}$$

Faktor jistoty vyjadřuje stupeň důvěry v hypotézu *H*, jestliže předpoklad *E* je pravdivý (1 znamená absolutní důvěru, –1 absolutní nedůvěru).

Míra důvěry (*measure of belief*) je definována takto:

$$MB(H, E) = \begin{cases} 1 & \text{pro } P(H) = 1 \\ \dfrac{\max\{P(H \mid E), P(H)\} - P(H)}{1 - P(H)} & \text{jinak} \end{cases}$$

Míra důvěry nabývá hodnot z intervalu $\langle 0,1 \rangle$. Vyjadřuje stupeň, ve kterém je důvěra v hypotézu *H* podporována pozorováním evidence *E*.

Míra nedůvěry (*measure of disbelief*) je definována vztahem:

$$MD(H, E) = \begin{cases} 1 & \text{pro } P(H) = 0 \\ \dfrac{P(H) - \min\{P(H \mid E), P(H)\}}{P(H)} & \text{jinak} \end{cases}$$

Míra nedůvěry nabývá hodnot z intervalu $\langle 0,1 \rangle$. Vyjadřuje stupeň, ve kterém je nedůvěra v hypotézu *H* podporována pozorováním evidence *E*.

Předpoklad *E* v pravidle $E \rightarrow H$ nemusí být znám s absolutní jistotou. Může být odvozen z jiného pravidla nebo zadán uživatelem s nějakým faktorem jistoty *CF*(*E*). Pak se výsledný faktor jistoty vypočte takto:

$$CF_{new}(H, E) = CF_{old}(H, E) \cdot CF(E)$$

Jestliže přitom předpoklad *E* obsahuje konjunkci nebo disjunkci dílčích podmínek, pak se při výpočtu *CF*(*E*) použijí následující vzorce:

$$CF(E_1 \vee E_2) = \max\{CF(E_1), CF(E_2)\}$$

$$CF(E_1 \wedge E_2) = \min\{CF(E_1), CF(E_2)\}$$

Problém kombinace více pravidel se stejným závěrem je v systému MYCIN řešen následovně. Mějme pravidla $E_1 \rightarrow H$, $E_2 \rightarrow H$, ... , $E_n \rightarrow H$. Označme $CF_n = CF(H, E_1, \ldots, E_n)$. Výpočet CF_n se provede podle následujícího vzorce:

$$CF_n = \begin{cases} CF_{n-1} + CF(H, E_n) \cdot (1 - CF_{n-1}) \\ \qquad \text{pro } CF_{n-1} > 0 \text{ a } CF(H, E_n) > 0 \\ CF_{n-1} + CF(H, E_n) \cdot (1 + CF_{n-1}) \\ \qquad \text{pro } CF_{n-1} < 0 \text{ a } CF(H, E_n) < 0 \\ \dfrac{CF_{n-1} + CF(H, E_n)}{1 - \min\{|CF_{n-1}|, |CF(H, E_n)|\}} \qquad \text{jinak} \end{cases}$$

Výhody přístupů založených na faktorech jistoty:

- jednoduchý a účinný výpočetní model
- shromáždění potřebných dat podstatně snazší než v jiných metodách
- snazší implementace vysvětlovacího mechanismu

Nevýhody:

- absence pevných teoretických základů
- implicitní předpoklad nezávislosti evidencí E_i

4.5 Dempster-Shaferova teorie

Dempster-Shaferova teorie byla vyvinuta ve snaze o překonání některých nedostatků pravděpodobnostního přístupu, jako např. reprezentace neznalosti (ignorance) a požadavku, že součet měr důvěry v událost a její negaci musí být roven 1. Zatímco pravděpodobnost představuje stupeň, ve kterém je tvrzení považováno za pravdivé, míra domnění v DS teorii představuje podporu tomuto tvrzení. Růst pravděpodobnosti hypotézy redukuje pravděpodobnost komplementu, kdežto v DS teorii růst podpory hypotézy nezpůsobuje změnu podpory komplementu. DS teorie je obecnější než bayesovský přístup, jelikož důvěra ve tvrzení a důvěra v negaci tohoto tvrzení nemusí být v součtu rovna 1.

DS teorie vychází z pojmu prostředí, což je úplný systém vzájemně disjunktních základních hypotéz :

$$X = \{h_1, h_2, \ldots, h_n\}$$

Základní přiřazení (*basic probability assignment*, *mass probability function*) je funkce definovaná na množině všech podmnožin množiny X a nabývající hodnot z intervalu $\langle 0, 1 \rangle$, tj.

$$m\colon 2^X \to \langle 0, 1 \rangle,$$

která má tyto vlastnosti:

$$m(\varnothing) = 0, \quad \sum_{A \subseteq X} m(A) = 1$$

Hodnota $m(A)$ představuje míru důvěry, že platí právě hypotéza A, přičemž nevypovídá nic o míře důvěry ve složky množiny A. Nemusí tedy platit $m(B) \leq m(A)$ pro $B \subseteq A$. Řekneme, že A je *fokální element,* jestliže $m(A) > 0$.

Míra domnění (*measure of belief*) v platnost hypotézy A je definována jako součet základních přiřazení všech podmnožin množiny A:

$$Bel(A) = \sum_{B \subseteq A} m(B)$$

Na rozdíl od základního přiřazení tedy $Bel(A)$ vyjadřuje míru domnění v hypotézu A nebo v jakoukoli z jejích podmnožin. Platí následující vztahy:

$$Bel(A) + Bel(A^c) \leq 1, \text{ kde } A^c = X - A,$$

$$Bel(A) \leq Bel(B) \text{ pro } A \subseteq B,$$

$$Bel(X) = 1.$$

Míra věrohodnosti (*measure of plausibility*) je definována takto:

$$Pl(A) = 1 - Bel(A^c)$$

Hodnota $Pl(A)$ vyjadřuje míru chyby při zamítnutí A. Platí

$$Bel(A) \leq Pl(A)$$

$$Pl(A) + Pl(A^c) \geq 1,$$

$$Pl(A) \leq Pl(B) \text{ pro } A \subseteq \text{B}.$$

Míra domnění a míra věrohodnosti vymezují ***interval domnění***:

$$\langle\, Bel(A), Pl(A) \,\rangle$$

Tento interval vyjadřuje rozsah naší jistoty o hypotéze A. Rozdíl $Pl(A) - Bel(A)$ se nazývá *nejistota* o hypotéze A nebo *ignorance*.

Kombinace základních přiřazení je dána ***Dempsterovým pravidlem***:

$$m_3(C) = \frac{\sum_{A,B \subseteq X,\ A \cap B = C} m_1(A) \cdot m_2(B)}{1 - \sum_{A,B \subseteq X,\ A \cap B = \varnothing} m_1(A) \cdot m_2(B)}$$

Základní přiřazení m_1 a m_2 mohou pocházet od různých expertů, případně m_1 může být výchozí základní přiřazení a m_2 základní přiřazení získané na základě nových skutečností.

Jestliže pozorujeme B, pak se základní přiřazení, domnění a věrohodnost A mění takto:

$$m(A \mid B) = \frac{\sum_{C \subseteq X,\ C \cap B = A} m(C)}{1 - \sum_{C \subseteq X,\ C \cap B = \varnothing} m(C)} \quad \text{pro } A \neq \varnothing; \qquad m(\varnothing|B) = 0$$

$$Bel(A \mid B) = \frac{Bel(A \cup B^c) - Bel(B^c)}{1 - Bel(B^c)}$$

$$Pl(A \mid B) = \frac{Pl(A \cap B)}{Pl(B)}$$

4.6 Fuzzy přístupy ke zpracování neurčitosti

Fuzzy přístupy ke zpracování neurčitosti se opírají o fuzzy množiny, lingvistické proměnné a vícehodnotovou logiku.

4.6.1 Fuzzy množiny

Fuzzy množina A v univerzu *U* je definována jako dvojice

$$A = (U, \mu_A)$$

kde $U \neq \emptyset$ je klasická množina a μ_A je funkce příslušnosti (charakteristická funkce), $\mu_A : U \to \langle 0; 1 \rangle$. Hodnota $\mu_A(x)$ se nazývá stupeň příslušnosti prvku x k fuzzy množině A. Pojem fuzzy množiny je zobecněním pojmu klasické množiny, neboť klasickou množinu A lze definovat stejným způsobem s tím, že $\mu_A : U \to \{0; 1\}$.

Prázdná fuzzy množina $\emptyset$ je definována funkcí příslušnosti $\mu_\emptyset(x) = 0$ pro všechna $x \in U$.

Nosič (*support*) fuzzy množiny *A* je klasická množina

$$supp(A) = \{x \mid \mu_A(x) > 0\}$$

Jádro (*kernel*) fuzzy množiny *A* je klasická množina

$$ker(A) = \{x \mid \mu_A(x) = 1\}$$

Výška (*height*) fuzzy množiny *A* je definována takto

$$hgt(A) = \sup\{\mu_A(x) \mid x \in U\}$$

Řekneme, že fuzzy množina je *normální*, jestliže *ker A* $\neq \emptyset$ a *hgt A* = 1.

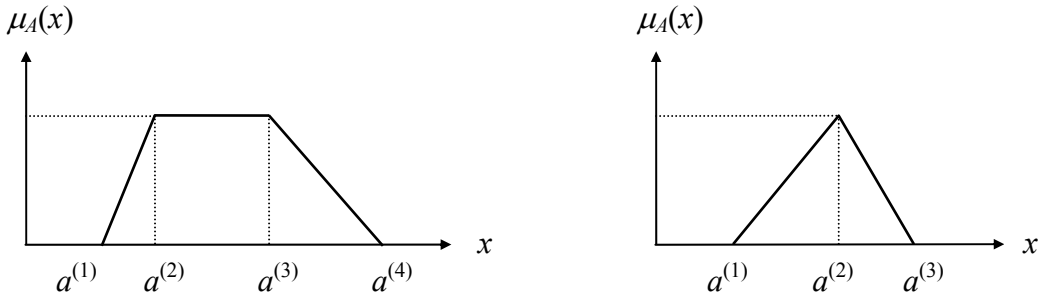
Zvláštním případem fuzzy množiny je *fuzzy číslo*. *Fuzzy číslo A* je fuzzy množina na universu reálných čísel, která je určena čtveřicí bodů $(a^{(1)}, a^{(2)}, a^{(3)}, a^{(4)})$ a po částech spojitou funkcí příslušnosti s následujícími vlastnostmi: (i) $a^{(1)} \leq a^{(2)} \leq a^{(3)} \leq a^{(4)}$; (ii) $\mu_A(x) = 0$ pro $x \leq a^{(1)}$, $x \geq a^{(4)}$; (iii) $\mu_A(x) = 1$ pro $a^{(2)} \leq x \leq a^{(3)}$; (iv) $\mu_A(x)$ je rostoucí na $\langle a^{(1)}, a^{(2)} \rangle$ a klesající na $\langle a^{(3)}, a^{(4)} \rangle$.

V praktických aplikacích je obvykle obtížné kvalifikovaně stanovit funkce příslušnosti se složitějším průběhem. V takových případech se proto používají lichoběžníková nebo

trojúhelníková fuzzy čísla. Funkce příslušnosti lichoběžníkového fuzzy čísla je plně určena čtveřicí $(a^{(1)}, a^{(2)}, a^{(3)}, a^{(4)})$ takto:

$$\mu_A(x) = \max\left(\min\left(\frac{x-a^{(1)}}{a^{(2)}-a^{(1)}}, \frac{x-a^{(4)}}{a^{(3)}-a^{(4)}}, 1\right), 0\right)$$

Trojúhelníkové fuzzy číslo je plně určeno trojicí hodnot $(a^{(1)}, a^{(2)}, a^{(3)})$. Můžeme je považovat za zvláštní případ lichoběžníkového fuzzy čísla, kde $a^{(2)} = a^{(3)}$. Příklady lichoběžníkového a trojúhelníkového fuzzy čísla jsou ukázány na obr. 4.1.



Obr. 4.1. Příklady fuzzy čísel (a) lichoběžníkové, (b) trojúhelníkové

Mějme fuzzy množiny $A = (U, \mu_A)$, $B = (U, \mu_B)$. Pak základní operace s fuzzy množinami jsou definovány takto:

Doplněk fuzzy množiny *A*: $\overline{A} = (U, \mu_{\overline{A}})$, $\mu_{\overline{A}}(x) = 1 - \mu_A(x)$

Sjednocení fuzzy množin *A* a *B*: $A \cup B = (U, \mu_{A\cup B})$ $\mu_{A\cup B}(x) = \max\{\mu_A(x), \mu_B(x)\}$

Průnik fuzzy množin *A* a *B*: $A \cap B = (U, \mu_{A\cap B})$ $\mu_{A\cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}$

Vztahy rovnosti a inkluze jsou definovány následujícím způsobem. Řekneme, že

A = B, právě když $\mu_A(x) = \mu_B(x)$ pro všechna $x \in U$,

A $\subseteq$ B, právě když $\mu_A(x) \le \mu_B(x)$ pro všechna $x \in U$,

A $\subset$ B, právě když $\mu_A(x) < \mu_B(x)$ pro všechna $x \in U$.

Nechť $A = (U, \mu_A)$, $B = (V, \mu_B)$. Pak *kartézský součin* fuzzy množin *A* a *B* definujeme jako fuzzy množinu

$$A \times B = (U \times V, \mu_{A\times B}),\ \mu_{A\times B}(x, y) = \min\{\mu_A(x), \mu_B(y)\}$$

Fuzzy relace je fuzzy množina

$$R = (U_1 \times U_2 \times ... \times U_n, \mu_R)$$

kde $U_1, U_2, ..., U_n$, jsou klasické množiny a $\mu_R : U_1 \times U_2 \times ... \times U_n \to \langle 0, 1\rangle$. Fuzzy relace rozšiřuje pojem relace na takové případy, v nichž nelze jednoznačně rozhodnout, zda mezi

danými dvěma či více objekty existuje vztah nebo ne. Kartézský součin fuzzy množin je zvláštním případem fuzzy relace.

Nechť $m < n,\ 1 \le i_1 < i_2 < ... < i_m \le n$ a $R = (U_{i_1} \times U_{i_2} \times ... \times U_{i_m}, \mu_R)$. *Cylindrické rozšíření* fuzzy relace R na $U_1 \times U_2 \times ... \times U_n$ je fuzzy relace $Cyl(R) = R^*$ s funkcí příslušnosti

$$\mu_{R^*}(x_1, x_2, ..., x_n) = \mu_R(x_{i_1}, x_{i_2}, ..., x_{i_m})$$

Nechť $R = (U \times V, \mu_R)$ a $S = (V \times W, \mu_S)$ jsou fuzzy relace. *Silná kompozice* relací R a S je fuzzy relace $R \circ S = (U \times W, \mu_{R \circ S})$ s funkcí příslušnosti

$$\mu_{R \circ S}(x, z) = \sup_{y \in V} \min\{\mu_R(x, y), \mu_S(y, z)\}$$

4.6.2 Lingvistická proměnná

Lingvistická (slovní, jazyková) proměnná je taková proměnná, jejíž hodnotami jsou slova. Významy těchto slov jsou reprezentovány jako fuzzy množiny v nějakém univerzu.

Lingvistická proměnná je obecně definována jako uspořádaná pětice

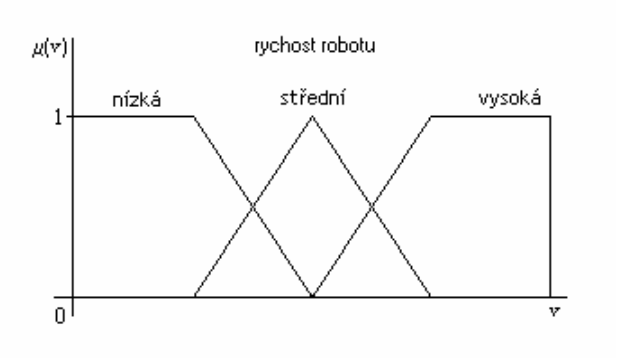
$$X = (X, T, U, G, M)$$

kde X je jméno proměnné, T je množina termů (tj. slovních hodnot, kterých může proměnná nabývat, U je univerzum (neprázdná klasická množina), G je množina syntaktických pravidel, pomocí nichž jsou generovány hodnoty z T a M je množina sémantických pravidel, která interpretují hodnoty z T jako fuzzy množiny s univerzem U.

Např. pro slovní proměnnou X = *věk* může množina T obsahovat hodnoty *mladý*, *starý*, *ne starý*, *velmi mladý*, *víceméně mladý*, *spíše starý*, … . G je gramatika, která určuje, jak se tyto hodnoty generují ze základních (terminálních) symbolů *mladý*, *starý*, *a*, *nebo*, *ne*, *velmi*, *víceméně*, *spíše*, … . Základní slovní hodnoty (*mladý*, *starý*) jsou reprezentovány fuzzy množinami, zatímco logickým operátorům (*a*, *nebo*, *ne*, …) a jazykovým operátorům (*velmi*, *víceméně*, *spíše*, …) mohou odpovídat fuzzy množinové operace.

Uvedený příklad je příkladem tzv. *strukturované* lingvistické proměnné. Lingvistická proměnná se nazývá *nestrukturovaná* (*normální*), jestliže množina T je dána konečným výčtem hodnot (tedy není zapotřebí gramatika G) a identifikátor každé slovní hodnoty z T je totožný s názvem fuzzy množiny, která je interpretací této slovní hodnoty. V důsledku toho můžeme lingvistickou proměnnou reprezentovat trojicí $X = (X, T, U)$, kde T je konečná množina fuzzy množin s univerzem U.

Např. slovní proměnná *rychlost robotu* může nabývat tří slovních hodnot *nízká, střední* a *vysoká*, kterým odpovídají fuzzy čísla definovaná na intervalu reálných čísel (viz. obr. 4.2). Obvykle se požaduje, aby součet funkcí příslušnosti byl pro každý bod univerza roven 1.



Obr. 4.2. Slovní proměnná *rychlost robotu*

4.6.3 Vícehodnotová logika

Vícehodnotová logika je logika s více než dvěma pravdivostními hodnotami, které se zpravidla nacházejí v intervalu $\langle 0, 1\rangle$. Pojem vícehodnotové logiky je starší než pojem fuzzy množiny, avšak k jejímu většímu rozšíření dochází až v souvislosti s nástupem fuzzy množin. Stupeň příslušnosti $\mu_A(x)$ prvku x k fuzzy množině A lze totiž interpretovat jako pravdivostní hodnotu predikátu $x \in A$.

Množinu $C = \langle 0, 1\rangle$ zde budeme považovat za množinu logických (pravdivostních) hodnot, přičemž 0 představuje pravdu a 1 nepravdu. Logická konstanta je tedy libovolné reálné číslo z intervalu $\langle 0, 1\rangle$. Logická proměnná je proměnná nabývající hodnot z množiny C.

Množinu $L = \{\vee, \wedge, \&, \Rightarrow\}$ budeme nazývat množinou logických spojek a její prvky logickými spojkami s těmito názvy: *disjunkce*, *konjunkce*, *odvážná konjunkce*, *implikace*.

Nechť W je konečná množina logických proměnných. Formulí nazveme konečný řetězec, definovaný následujícími pravidly:

1. Je-li $\alpha \in C$, pak α je formule.
2. Je-li $\beta \in W$, pak β je formule.
3. Jestliže φ a ψ jsou formule a $* \in L$, pak $(\varphi * \psi)$ je formule.

Interpretace formule je dosazení logických konstant za logické proměnné.

Nechť Q je množina všech formulí a $\Omega(Q)$ množina všech jejich interpretací. *Pravdivostním ohodnocením* nazveme zobrazení V: $\Omega(Q) \to$ C, splňující následující požadavky:

1. $V(\alpha) = \alpha$ pro každé $\alpha \in C$.
2. $V(\varphi \vee \psi) = \max\big(V(\varphi), V(\psi)\big)$ pro všechna φ, $\psi \in \Omega(Q)$.
3. $V(\varphi \wedge \varphi) = \min\big(V(\varphi), V(\psi)\big)$ pro všechna φ, $\psi \in \Omega(Q)$.
4. $V(\varphi \,\&\, \psi) = \max\big(0, V(\varphi) + V(\psi) - 1\big)$ pro všechna φ, $\psi \in \Omega(Q)$.
5. $V(\varphi \Rightarrow \psi) = \min\big(1, 1 - V(\varphi) + V(\psi)\big)$ pro všechna φ, $\psi \in \Omega(Q)$.

Operaci negace definujeme takto:

$$\neg\varphi = \varphi \Rightarrow 0$$

Pro pravdivostní ohodnocení negace pak dostaneme obvyklý vztah:

$$V(\neg\varphi) = V(\varphi \Rightarrow 0) = \min\bigl(1, 1 - V(\varphi)\bigr) = 1 - V(\varphi)$$

Mezi konjunkcí a odvážnou konjunkcí existuje následující vztah:

$$V(\varphi \,\&\, \psi) \le V(\varphi \wedge \psi) \text{ pro všechna } \varphi,\ \psi \in \Omega(Q)$$

Použití odvážné konjunkce je namístě tam, kde mezi dílčími výroky existuje vnitřní souvislost. Např. místo konjunkce výroků „Jan je tlustý“ a „Jan je podobný Petrovi“ bychom měli použít odvážnou konjunkci, neboť tloušťka je součástím podoby a tedy pravdivost prvého výroku je částečně obsažena v pravdivosti druhého výroku. Výsledná pravdivost by pak měla být obecně menší než pravdivost každého z výroků.

Implikace definovaná vztahem 5 je *Lukasiewiczova implikace*. Existuje řada dalších definic zobecněné implikace, jako např.

Kleene-Dienesova:

$$V(\varphi \Rightarrow \psi) = \max\bigl(1 - V(\varphi), V(\psi)\bigr)$$

Zadehova:

$$V(\varphi \Rightarrow \psi) = \max\bigl(1 - V(\varphi), \min\bigl(V(\varphi), V(\psi)\bigr)\bigr)$$

Gödelova:

$$V(\varphi \Rightarrow \psi) = \begin{cases} 1 & \text{pro } V(\varphi) \le V(\psi) \\ V(\psi) & \text{jinak} \end{cases}$$

Definice dalších zobecněných implikací je možno najít v (Druckmüller 1998), kde je také provedeno jejich srovnání. Z něj vychází nejlépe Lukasiewiczova implikace.

4.6.4 Kompoziční pravidlo usuzování

Uvažujme pravidlo

$$\text{IF } X = A \text{ THEN } Y = B$$

Při fuzzy přístupu ke zpracování neurčitost se hodnoty A a B chápou jako fuzzy množiny, obvykle reprezentující nějaké slovní hodnoty. Příkladem může být pravidlo

IF teplota = nízká THEN tlak = malý.

Nechť $A = (U, \mu_A)$, $B = (V, \mu_B)$. Pak pravidlo IF $X = A$ THEN $Y = B$ můžeme chápat jako fuzzy relaci

$$R = (U \times V, \mu_R)$$

Ve fuzzy systémech se charakteristická funkce této relace často definuje vztahem

$$\mu_R(x, y) = \min\{\mu_A(x), \mu_B(y)\}$$

a relace se označuje názvem *Mamdaniho implikace*, ačkoli o se implikaci v pravém smyslu slova nejedná.

Inference v deterministických pravidlových systémech je založena na použití pravidla *modus ponens*. Ve fuzzy systémech se pracuje se zobecněným pravidlem *fuzzy modus ponens*:

$$\frac{X = A', \quad \text{IF } X = A \text{ THEN } Y = B}{Y = B'}$$

Nechť $A' = (U, \mu_{A'})$. Pak fuzzy množina $B' = (V, \mu_{B'})$ může být určena takto:

$$\mu_{B'}(y) = \sup_{x \in U} \min\{\mu_{A'}(x), \mu_R(x, y)\}$$

Je-li univerzum U konečná množina, můžeme operátor *sup* nahradit operátorem *max*. Jedná se vlastně o kompozici unární relace A' a binární relace R. Proto tento způsob usuzování označuje názvem *kompoziční pravidlo usuzování*. Jiný způsob usuzování založený na fuzzy logice je implementován v systému LMPS.

Předpokládejme nyní, že znalostní báze je tvořena m pravidly tvaru

$$\text{IF } X_1 = A_{i1} \text{ AND } X_2 = A_{i2} \text{ AND } \ldots \text{ AND } X_n = A_{in} \text{ THEN } Y = B_i$$

kde $A_{ij} = (U_j, \mu_{A_{ij}})$ a $B_i = (V, \mu_{B_i})$. Těmto pravidlům odpovídají fuzzy relace

$$R_i = (U_1 \times U_2 \times \ldots \times U_n \times V, \mu_{R_i})$$

Podmínku na levé straně i-tého pravidla můžeme také vyjádřit ve tvaru $X = A_i$, kde $X = (X_1, X_2, \ldots, X_n)$ a $A_i = A_{i1} \times A_{i2} \times \ldots \times A_{in}$. Fuzzy množina A_i může být také vyjádřena jako průnik cylindrických rozšíření fuzzy množin $A_{ij} = (U_j, \mu_{A_{ij}})$ na univerzum $U_1 \times U_2 \times \ldots \times U_n$. Obvykle se předpokládá, že jednotlivá pravidla jsou propojena spojkou OR (nebo). Té odpovídá operace sjednocení fuzzy množin, takže báze znalostí může být reprezentována relací

$$R = \bigcup_{i=1}^{m} R_i$$

Nechť nyní je položen dotaz

$$X_1 = A_{01} \text{ a } X_2 = A_{02} \text{ a } \ldots \text{ a } X_n = A_{0n}\text{; jaká je hodnota } Y\text{?}$$

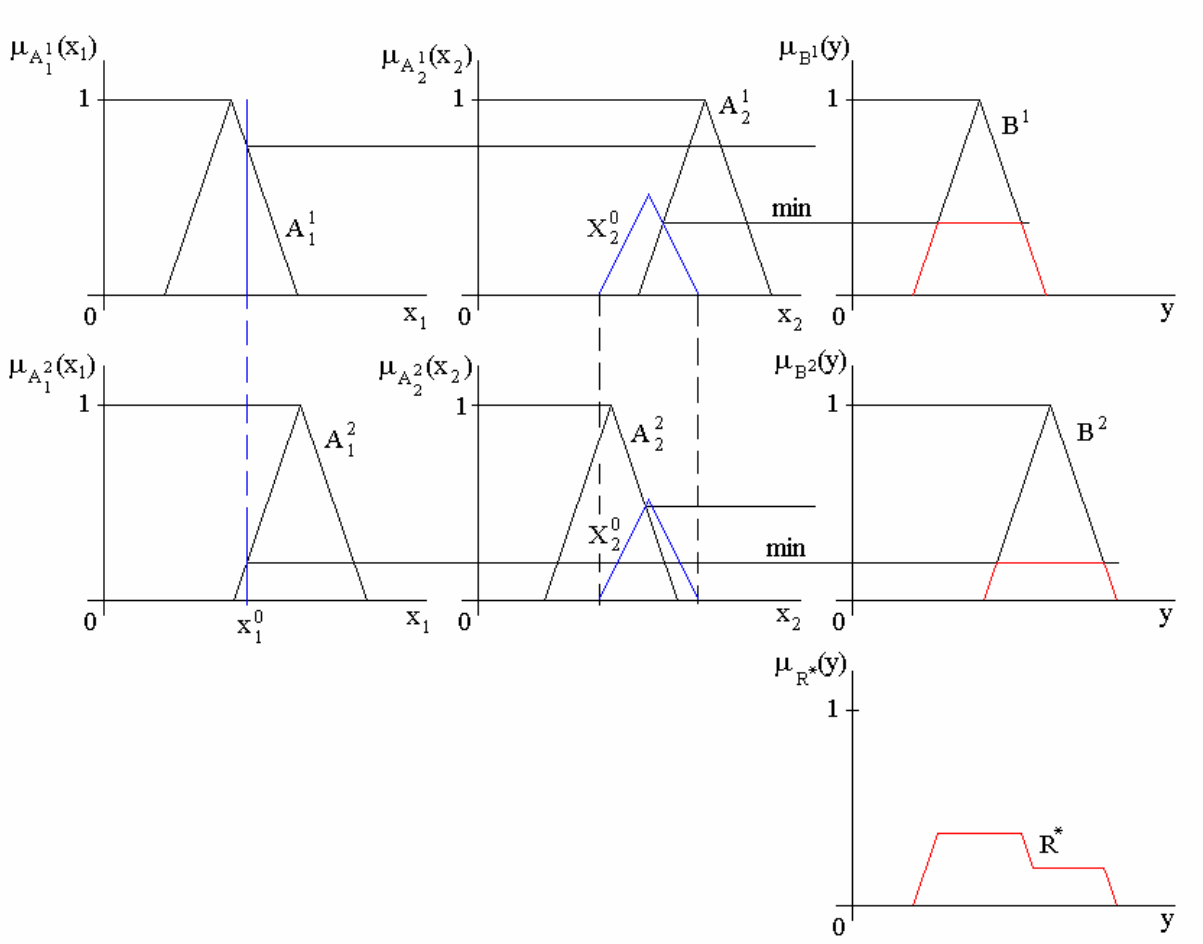
Odpovědí systému je fuzzy množina $B_0 = (V, \mu_{B_0})$, jejíž funkce příslušnosti je určena vztahem

$$\mu_{B_0}(y) = \sup_{\mathbf{x} \in U} \min\left\{ \min_{j=1,\ldots,n} \mu_{A_{0j}}(x_j), \max_{i=1,\ldots,m} \mu_{R_i}(\mathbf{x}, y) \right\}$$

kde $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ a $U = U_1 \times U_2 \times \ldots \times U_n$. Při použití Mamdaniho interpretace relací R_i můžeme tento vztah převést do tvaru

$$\mu_{B_0}(y) = \max_{i=1,\dots,m} \min\left\{ \mu_{B_i}(y), \min_{j=1,\dots,n} \sup_{x_j \in U_j} \min\left\{ \mu_{A_{0j}}(x_j), \mu_{A_{ij}}(x_j) \right\} \right\}$$

Uvedený tvar umožňuje efektivnější výpočet. Popsaný Mamdaniho inferenční mechanismus pro dvě pravidla se dvěma nezávisle proměnnými ilustruje obr. 4.3. Předpokládá se dotaz tvořený ostrou hodnotou x_1^0 prvé proměnné a fuzzy hodnotou X_2^0 proměnné druhé. Výsledná odpověď je označena symbolem R^*.



Obr. 4.3. Mamdaniho inferenční mechanismus

Popsaný inferenční mechanismus byl implementován v práci (Krček 2003) pro lokální navigaci tříkolového mobilního robota. V modelu byly použity čtyři vstupní proměnné (vzdálenost od cíle, úhel natočení vzhledem k cíli, vzdálenost od nejbližší překážky a úhel natočení vzhledem k nejbližší překážce) a dvě výstupní proměnné (rychlost robota a úhel natočení předního kola). Fungování systému bylo ověřeno simulačním programem. Informace o této aplikaci je možné najít také v práci (Krček a Dvořák 2003).

4.6.5 Systém LMPS

LMPS (*Linguistic Model Processing System*, Druckmüller 1991) je systém pro slovní modelování funkcí a relací. Slovní model je formule, ve které jsou nahrazeny logické proměnné charakteristickými funkcemi fuzzy množin, které sémanticky interpretují slovní hodnoty lingvistických proměnných.

Systém LMPS rozlišuje dva typy slovních proměnných:

- slovní proměnná s reálným univerzem – její hodnoty jsou reprezentovány fuzzy množinami z univerza, které je množinou reálných čísel,
- slovní proměnná s univerzem slovních hodnot – její hodnoty nelze kvantifikovat užitím množiny reálných čísel (příkladem může být slovní proměnná *nemoc*).

V systému LMPS se používají tři typy slovních modelů: CCD-model, CIC-model a CI&-model. CCD-model je vhodný pro modelování relací, které nejsou funkcemi. Tento model lze použít i k modelování funkcí, ale CIC-model a CI&-model mají pro tento účel lepší vlastnosti.

CCD-model je tvořen CC-prohlášeními, která jsou propojena spojkou disjunkce. CC-prohlášení má tvar

$$X_1 \text{ je } A_1 \text{ a } X_2 \text{ je } A_2 \text{ a } \ldots \text{ a } X_n \text{ je } A_n \text{ a } Y \text{ je } B,$$

kde X_j a Y jsou lingvistické proměnné a A_j a B jsou jejich slovní hodnoty. CC-prohlášení je vlastně sémantickou interpretací IF-THEN pravidla při použití Mamdaniho „implikace“. CCD-model interpretuje pravdu tak, že pravdivé je to, co tvrdí alespoň jedno prohlášení. Model není citlivý na spory mezi prohlášeními a nebere v úvahu redundantní informace.

CIC-model a CI&-model jsou tvořeny CI-prohlášeními tvaru

$$\text{Jestliže } X_1 \text{ je } A_1 \text{ a } X_2 \text{ je } A_2 \text{ a } \ldots \text{ a } X_n \text{ je } A_n \text{ pak } Y \text{ je } B$$

CI-prohlášení je sémantickou interpretací IF-THEN pravidla při použití Lukasiewiczovy implikace. V CIC-modelu jsou prohlášení propojena pomocí konjunkce, kdežto v CI&-modelu je k tomuto účelu použita odvážná konjunkce. V těchto modelech se za pravdivé považuje to, co není v rozporu s žádným prohlášením. Oba modely jsou citlivé na spory (absolutní spor v prohlášeních vede k úplné ztrátě informace) a berou v úvahu redundantní informace. Vlastnosti CIC-modelu redundantní informace obecně zhoršují, kdežto u CI&-modelu je tomu naopak (pokud tyto informace nejsou vzájemně sporné).

Systému LMPS jsou zdávány dotazy tvaru

$$\text{Jaká je hodnota } Y, \text{ jestliže } X_1 \text{ je } H_1 \text{ a } X_2 \text{ je } H_2 \text{ a } \ldots \text{ a } X_n \text{ je } H_n ?$$

Při výpočtu odpovědi na zadaný dotaz systém LMPS postupuje následujícím způsobem. Předpokládejme, že hodnoty proměnné X_j (j = 1, … ,n) jsou definovány na univerzu U_j a hodnoty proměnné Y na univerzu V. Každé CC-prohlášení je nahrazeno formulí

$$\alpha_1 \wedge \alpha_2 \wedge \ldots \alpha_n \wedge \beta(y)$$

a každé CI-prohlášení je nahrazeno formulí

$$\alpha_1 \wedge \alpha_2 \wedge \ldots \alpha_n \Rightarrow \beta(y)$$

pro každé $y \in V$, kde

$$\alpha_j = \max_{x \in U_j} \{\min(\mu_{H_j}(x), \mu_{A_j}(x))\} \text{ pro } X_j \text{ s reálným univerzem}$$

$$\alpha_j = \begin{cases} 1 & \text{je - li } H_j = A_j \\ 0 & \text{je - li } H_j \neq A_j \end{cases} \text{ pro } X_j \text{ s univerzem slovních hodnot}$$

$$\beta(y) = \mu_B(y) \text{ pro } Y \text{ s reálným univerzem}$$

$$\beta(y) = \begin{cases} 1 & \text{je - li } y = B \\ 0 & \text{je - li } y \neq B \end{cases} \text{ pro } Y \text{ s univerzem slovních hodnot}$$

Pak pro každou hodnotu $y \in V$ jsou tyto formule spojeny logickými spojkami $\vee$, resp. $\wedge$, resp. & podle toho, zda se jedná o CCD, resp. CIC, resp. CI&-model. Tak pro každé $y \in V$ vznikne jediná formule, jejíž pravdivostní ohodnocení určuje stupeň příslušnosti hodnoty y k fuzzy množině, která je odpovědí na zadaný dotaz.

4.6.6 Defuzzifikace

Odpovědí fuzzy expertního systému na zadaný dotaz je fuzzy množina. Často ale v praxi potřebujeme, aby odpovědí byla jediná ostrá hodnota. Proces, v němž nějaké fuzzy množině přiřazujeme ostrou hodnotu, která ji v jistém smyslu nejlépe reprezentuje, nazýváme *defuzzifikací*.

Nechť odpovědí je fuzzy množina $B_0 = (V, \mu_{B_0})$, kde V je množina reálných čísel. Mezi nejčastěji používané metody defuzzifikace patří tyto metody:

- *Metoda těžiště* (COA, *center of area*).

  Defuzzifikovaná ostrá hodnota se určí jako souřadnice těžiště plochy shora ohraničené charakteristickou funkcí odpovědi. V případě spojitého univerza V je tato souřadnice určena vztahem

$$y_0 = \frac{\int_V y \mu_{B_0}(y) dy}{\int_V \mu_{B_0}(y) dy}$$

  Pro diskrétní univerzum se místo integrálu použije suma.

- *Metoda maxima*.

  Jako ostrá odpověď je vybrán bod absolutního maxima funkce $\mu_{B_0}(y)$, tj.

$$y_0 = \arg\max_{y \in V} \mu_{B_0}(y)$$

  Pokud je takových bodů více, může se použít některá z následujících metod.

- *Metoda prvého maxima* (FOM, *first of maxima*).

  Ostrou odpovědí je prvý bod, v němž funkce $\mu_{B_0}(y)$ nabývá absolutního maxima.

- *Metoda průměrného maxima* (MOM, *mean of maxima*).

  Ostrou odpovědí je průměr z hodnot y, v nichž funkce $\mu_{B_0}(y)$ nabývá absolutního maxima.

Defuzzifikace odpovědi v systému LMPS probíhá následovně. Pokud charakteristická funkce odpovědi nabývá svého maxima v jediném prvku univerza V, pak tento prvek představuje nejpravdivější možnou odpověď na zadaný dotaz. Je-li takových prvků více, pak v případě reálného univerza je vybrán ten prvek, který leží nejblíže těžišti plochy, shora ohraničené charakteristickou funkcí odpovědi.

4.7 Bayesovské sítě

Bayesovská síť (*Bayesian belief network*) je orientovaný acyklický graf, jehož uzlům odpovídají náhodné proměnné a vazby reprezentují kauzální závislosti mezi těmito proměnnými. Hrana $X \rightarrow Y$ znamená, že X kauzálně ovlivňuje Y (pozorování X poskytuje kauzální podporu Y, pozorování Y poskytuje diagnostickou podporu pro X). Bayesovská síť umožňuje provádět prediktivní i diagnostické inference.

Každému uzlu je přiřazena tabulka rozdělení pravděpodobnosti. Jestliže uzel nemá žádné předchůdce (rodiče), jedná se o nepodmíněnou pravděpodobnost, v opačném případě jde o podmíněnou pravděpodobnost.

Nechť $G = (V, E)$ je orientovaný acyklický graf a nechť $v \in V$. Definujme následující množiny:

$$C(v) = \{u \in V \mid (u, v) \in E\},$$

$$D(v) = \{w \in V \mid \text{existuje cesta z } v \text{ do } w\},$$

$$A(v) = \{x \in V \mid \neq v \text{ a } x \notin C(v) \cup D(v)\}.$$

Množina $C(v)$ je množinou bezprostředních předchůdců (rodičů) uzlu v, $D(v)$ je množinou všech následníků uzlu v (nejen bezprostředních). V případě bayesovské sítě prvky $C(v)$ nazýváme také *příčinami*.

Nechť (Ω, P) je pravděpodobnostní prostor, kde $\Omega = \Omega_1 \times \ldots \times \Omega_n$, a nechť X_i ($i = 1, \ldots, n$) je projekce Ω na Ω_i (tj. $X_i : \Omega \rightarrow \Omega_i$ je náhodná proměnná). Nechť (V, E) je orientovaný acyklický graf, kde $V = \{X_1, \ldots, X_n\}$. Řekneme, že (V, E, P) je ***bayesovská síť***, jestliže pro všechna $X_i \in V$ a všechna $W \subseteq A(X_i)$ jsou X_i a W podmíněně nezávislé při daném $C(X_i)$. To znamená, že když $W = \{Y_1, \ldots, Y_k\}$, $C(X_i) = \{Z_1, \ldots, Z_m\}$, $P(Y_1 \wedge \ldots \wedge Y_k \wedge Z_1 \wedge \ldots \wedge Z_m) \neq 0$, pak

$$P(X_i \mid Y_1 \wedge \ldots \wedge Y_k \wedge Z_1 \wedge \ldots \wedge Z_m) = P(X_i \mid Z_1 \wedge \ldots \wedge Z_m).$$

Zjednodušeně řečeno, když známe příčiny X_i, nic jiného než X_i samotné nebo jeho následníci nám nemůže dát nějaké další informace o X_i. Místo pojmu bayesovská síť se někdy užívají pojmy *kauzální síť* či *influenční diagram*.

Nechť X je náhodná proměnná s oborem hodnot $O(X)$ a P je pravděpodobnost. Symbolem $P(X)$ rozumíme funkci definovanou na $O(X)$ tak, že pro $x \in O(X)$ je $P(x) = P(X = x)$. Zápisy pravděpodobností můžeme dále zjednodušit následujícím způsobem. Nechť $V = \{X_1, \ldots, X_n\}$, $C(X_i) = \{Z_1, \ldots, Z_m\}$. Pak definujeme

$$P(V) = P(\{X_1\} \cup \ldots \cup \{X_n\}) = P(X_1, \ldots, X_n) = P(X_1 \wedge \ldots \wedge X_n),$$

$$P(X_i \mid C(X_i)) = P(X_i \mid Z_1, \ldots, Z_m) = P(X_i \mid Z_1 \wedge \ldots \wedge Z_m).$$

Pomocí uvedené symboliky můžeme definici bayesovské sítě přepsat takto: Řekneme, že (V, E, P) je bayesovská síť, jestliže pro všechna $X_i \in V$ a všechna $W \subseteq A(X_i)$, $P(W \cup C(X_i)) \neq 0$, platí, že

$$P(X_i \mid W \cup C(X_i)) = P(X_i \mid C(X_i)).$$

Nechť (V, E, P) je bayesovská síť. Pak platí:

$$P(V) = \prod_{\substack{X \in V \\ P(C(X)) \neq 0}} P(X \mid C(X))$$

Nechť (V, E) je orientovaný acyklický graf, kde $V = \{X_1, \ldots, X_n\}$, přičemž X_i jsou proměnné s obory hodnot $O(X_i)$. Nechť $f(X \mid C(X))$ je nezáporná reálná funkce taková, že

$$\sum_{x \in O(X)} f(x \mid C(X)) = 1$$

pro všechny kombinace hodnot proměnných z $C(X)$. Pak

$$\Omega = O(X_1) \times \ldots \times O(X_n) \qquad \text{a} \qquad P(V) = \prod_{X \in V} f(X \mid C(X))$$

definují pravděpodobnostní prostor, pro nějž (V, E, P) je bayesovská síť. Přitom $P(X \mid C(X))$ je buď 0 nebo $f(X \mid C(X))$.

Konstrukce bayesovské sítě probíhá v těchto krocích:

1. Specifikujeme veličiny $X_1, \ldots, X_n$ a jejich obory hodnot $O(X_i)$.
2. Zkonstruujeme orientovaný acyklický graf (V, E), kde $V = \{X_1, \ldots, X_n\}$, vyjadřující kauzální závislosti mezi veličinami.
3. Odhadneme pravděpodobnost P tak, že odhadneme $P(X \mid C(X))$ pro všechna X, všechny hodnoty X a všechny kombinace hodnot proměnných z $C(X)$. Podle předchozí věty je nezbytné splnění pouze těchto podmínek:

$$0 \leq P(X \mid C(X)) \leq 1$$

$$\sum_{x \in O(X)} P(x \mid C(X)) = 1$$

Problém řešený pomocí bayesovské sítě je možno zjednodušeně formulovat následujícím způsobem. Nechť je dána bayesovská síť (V, E, P) a množiny $U \subseteq V$, $W \subseteq V$, $U \cap W = \emptyset$. Jsou-li zadány hodnoty proměnných z množiny U, je třeba zjistit $P(W \mid U)$.

Po zadání hodnot některých proměnných se provádí *inference*, což znamená přepočet podmíněných pravděpodobností pro ostatní proměnné. Inference v bayesovské síti je založena na Bayesových vzorcích.

Obecně je výše zmíněný problém NP-těžký, což znamená, že pro něj neexistují algoritmy s polynomiální časovou složitostí. Pokud bayesovská síť nemá speciální strukturu, je nutné použít aproximační techniky, které jsou obvykle založeny na nějakých transformacích bayesovské sítě na jednodušší tvar.

Řekneme, že bayesovská síť je ***jednoduše souvislá***, jestliže mezi každými dvěma uzly existuje právě jedna neorientovaná cesta. Jednoduše souvislá síť se také nazývá *polystrom* nebo *les*. Zvláštním případem polystromu je *strom*, což je graf, kde každý uzel má nejvýše jednoho rodiče. Pro polystromovou bayesovskou síť existují algoritmy pro inferenci, které mají polynomiální časovou složitost.

5. Tvorba expertního systému

Problematikou tvorby expertních (znalostních) systémů se zabývá *znalostní inženýrství* (*knowledge engineering*). Znalostní inženýrství má mnoho společných rysů se softwarovým inženýrstvím. Odlišnosti se týkají typu, povahy a množství reprezentovaných znalostí. U softwarového inženýrství se jedná o dobře definované algoritmické znalosti. Povahu a množství těchto znalostí potřebných pro řešení daného problému lze předem dobře odhadnout. U znalostního inženýrství se jedná o extenzivní, nepřesné a špatně definované znalosti, jejichž povahu a množství lze předem velmi špatně odhadnout. To způsobuje potíže v počátečních etapách vývoje ES při odhadu potřebného úsilí a při tvorbě návrhu.

5.1 Životní cyklus expertního systému

Model životního cyklu ES kombinuje *rychlé prototypování* a *inkrementální vývoj* a obsahuje tyto etapy:

1. Analýza problému.
2. Specifikace požadavků.
3. Předběžný návrh.
4. Počáteční (rychlé) prototypování a vyhodnocování.
5. Konečný návrh.
6. Implementace (získávání a reprezentace znalostí).
7. Validace a verifikace (testování).
8. Změny návrhu.
9. Údržba.

Etapy 6, 7 a 8 se iteračně opakují pro jednotlivé části (subsystémy) expertního systému.

5.1.1 Analýza problému

Cílem analýzy je posoudit vhodnost aplikace znalostních technik pro řešení daného problému. Kritéria pro toto posouzení mohou být rozdělena do dvou skupin:

- Vhodnost aplikace
- Dostupnost zdrojů

Vhodnost aplikace

Při posuzování vhodnosti aplikace si musíme klást následující otázky:

1. Problém skutečně existuje?
2. Jsou pro něj vhodné znalostní techniky? Tato otázka se rozpadá na následující dílčí otázky:
   - Mohou být replikovány lidské znalosti řešení problému?
   - Znalosti jsou převážně heuristické?

- Jsou tyto znalosti dobře chápány a akceptovány?
- Expertízy se často mění (nejsou konstantní)?
- Vstupní data jsou nekompletní nebo nepřesná?
- Znalostní přístup k řešení je lepší než jiné prostředky?

Odpovědi na tyto otázky mají různou váhu a nemusejí být všechny kladné. Musejí být posuzovány jako celek s ohledem na konkrétní podmínky).

3. Je znalostní přístup oprávněn z hlediska nákladů a přínosů?

Dostupnost zdrojů

Posuzování dostupnosti zdrojů vede k následujícím otázkám:

1. Má projekt manažerskou podporu? Jedná se o:
   - dostatek času
   - potřebné prostředky a školení
   - disponibilitu expertů
2. Je podpora ze strany expertů?
3. Jsou experti kompetentní?
4. Jsou experti komunikativní?
5. Jsou experti fyzicky dostupní?
6. Jsou k dispozici jiné zdroje znalostí?

5.1.2 Specifikace požadavků

Specifikace požadavků může mít následující strukturu:

1. Úvod.

   Charakteristika problému, profil uživatelů, cíle projektu.

2. Funkce expertního systému.

   Vstupy a výstupy systému, pomocné funkce, implementační priority.

3. Omezení.

   Hardwarová omezení, externí rozhraní, kompatibilita s předchozími produkty, rychlost, spolehlivost, udržovatelnost, bezpečnost, identifikace chyb.

4. Závěrečné požadavky

   Metody validace a verifikace, požadavky na dokumentaci, jiné požadavky.

5.1.3 Předběžný návrh

Předběžný návrh by měl zahrnovat následující body:

1. Výběr paradigmatu reprezentace znalostí.
   - Pravidla nebo logika – vhodné pro mělké znalosti.
   - Struktury (rámce objekty, sémantické sítě) – vhodné pro hluboké a strukturálně provázané znalosti.
   - Hybridní systémy – spojení strukturálních znalostí se schopností inference.
2. Výběr metod usuzování (souvisí s volbou reprezentace).
3. Výběr nástrojů (komerční nebo zákaznický systém?).
4. Výběr lidských zdrojů (znalostní inženýři, vedoucí týmu, experti).
5. Požadavky na vývojový tým (dány zejména složitostí a rozsahem systému).

Kritéria pro výběr komerčního shellu

1. Paradigma reprezentace znalostí a usuzování.
2. Flexibilita.

   Uživatelsky definované funkce, externí rutiny, vestavěné funkce, podpora datových struktur.
3. Speciální požadavky.

   Časové usuzování, operace v reálném čase, zpracování neurčitosti, přístup k externímu softwaru, grafika, okna.
4. Pomocné funkce.

   Editor znalostní báze, trasování, vysvětlování, testovací a verifikační pomůcky, grafická prezentace znalostní báze.
5. Výkon.
6. Podpora výrobce.

   Dokumentace, on-line help, podpora horkou linkou, školení, konzultace.
7. Náklady.

5.1.4 Rychlé prototypování

Rychlé prototypování (*rapid prototyping*) využívá prostředky jako Lisp, Prolog a/nebo komerčně dostupné prázdné ES (*shells*) s cílem rychle vytvořit fungující prototyp finálního systému. Na základě vyhodnocení prototypu musejí být všechna předběžná rozhodnutí potvrzena nebo změněna.

Počáteční prototyp by měl mít dobré uživatelské rozhraní a rozumně robustní podmnožinu znalostí, aby zamýšlení uživatelé mohli posoudit jeho aplikovatelnost. Prototyp může být sice po

vyhodnocení dále modifikován, ale doporučuje se jeho opuštění a započetí implementace ES na základě konečného návrhu od počátku.

5.2 Získávání znalostí

Získávání znalostí (*knowledge acquisition*) je klíčovou operací implementace ES a představuje nejdelší a nejpracnější část vývoje ES. Akvizice znalostí je proces zjišťování (*elicitation*) znalostí ze zdrojů (expertů, textů, dat, obrázků, …) a jejich reprezentace v bázi znalostí.

Proces naplňování báze znalostí probíhá inkrementálně (*incremental development*). Postupně jsou implementovány zvládnutelné a relativně ucelené části znalostí (subsystémy). Po implementaci každé části probíhá testování, na jehož základě mohou být provedeny případné změny v návrhu.

Existují dva základní způsoby získávání znalostí:

1. Získávání znalostí od expertů formou spolupráce mezi znalostními inženýry a experty
   - 1 : 1 (nejčastější případ)
   - 1 : *n*
   - *m* : 1
   - *m* : *n*
2. Automatizované získávání znalostí (strojové učení)
   - od expertů
   - z textů
   - z dat (*data mining*)

Proces získávání znalostí od expertů se obvykle dělí do tří fází:

1. Seznámení s problémem, získání základních znalostí (spolupráce nejen s expertem ale také se zadavatelem a uživatelem)
2. Získávání obecných znalostí.
3. Získávání specifických znalostí.

Práce s jedním expertem probíhá obvykle formou *interview*. Je nutné přitom brát v úvahu nebezpečí zavlečení zavlečení chybné expertízy. Toto nebezpečí je nižší při práci se skupinou expertů, která je obvykle založena na panelové diskusi nebo *brainstormingu*. Práce se skupinou expertů je ovšem náročnější na přípravu a průběh a hrozí při ní nebezpečí konfliktů mezi experty.

Příprava interview zahrnuje optimalizaci a plánování. Optimalizace interview spočívá v pečlivém naplánování a efektivním řízení průběhu. Plán interview obsahuje:

- místo konání – v počáteční fázi na pracovišti experta, později (je-li to možné) na pracovišti znalostního inženýra
- doba trvání – kolem 2 hodin, rozhodně ne více než 3

- cíle interview – stanoveny na základě přehledu výsledků předchozího sezení

S plánem interview je třeba experta předem seznámit.

Při získávání znalostí od experta se používají tyto techniky:

- Nestrukturované interview (běžný rozhovor, vhodné pro počáteční fázi)
- Strukturované interview (kladení cílených dotazů, získání detailního pohledu)
- Myšlení nahlas (expert popisuje svá myšlenkové pochody a chování při řešení problému)
- Pokus o řešení problému pod dohledem experta s cílem vcítit se do jeho myšlenkových pochodů
- Metoda repertoárové tabulky (*repertory grid*)

  Sloupce této tabulky odpovídají objektům z dané oblasti a řádky odpovídají tzv. konstruktům. Každý konstrukt je tvořen dvěma mezními (nejlépe protikladnými) vlastnostmi objektů. Políčka tabulky obsahují číselná ohodnocení příslušnosti objektu k jednomu či druhému pólu.

Při práci s experty je třeba se vyrovnat s řadou problémů. Jedním z nich je známý paradox znalostního inženýrství: Čím více se experti stávají kompetentními, tím méně jsou schopni popsat znalost, kterou používají při řešení problémů. Další možné problémy vyplývají z následujícího přehledu typů problémových expertů:

- expert obávající se ztráty postavení po zavedení ES
- cynický expert
- velekněz oboru
- paternalistický expert
- nekomunikativní expert
- lhostejný expert
- pseudovzdělanec v umělé inteligenci

6. Strojové učení

Metody strojového učení můžeme rozdělit např. takto (Mitchell 1997):

- Prohledávání prostoru verzí (*version space search*)
- Techniky rozhodovacích stromů (*decision tree learning*)
- Techniky rozhodovacích pravidel (*learning sets of rules*)
- Techniky konceptuálního shlukování (*conceptual clustering techniques*)
- Posilované učení (reinforcement learning)
- Učení založené na vysvětlování (*explanation based learning*)
- Usuzování na základě analogií (*analogical reasoning*)
- Bayesovské učení (*Bayesian learning*)
- Evoluční techniky (*evolution techniques*)
- Učení neuronových sítí (*artificial neural networks*)

6.1 Prohledávání prostoru verzí

Prostor verzí (*version space*) je množina všech popisů konceptů, konzistentních s trénovacími daty. Existují tři algoritmy prohledávání prostoru verzí (Mitchell):

- Algoritmus generalizace (*specific to general search*)
- Algoritmus specializace (*general to specific search*)
- Algoritmus eliminace kandidátů (*candidate elimination algorithm*). Tento algoritmus udržuje dvě množiny kandidujících konceptů. *G* je množina maximálně obecných konceptů, *S* je množina maximálně specifických konceptů. Algoritmus specializuje *G* a generalizuje *S*.

Uvedené algoritmy používají jak pozitivní, tak i negativní příklady cílového konceptu. Je sice možné zobecňovat pouze z pozitivních příkladů, ale negativní příklady jsou důležité v prevenci přehnaného zobecnění.

Algoritmus eliminace kandidátů:

1. Vlož do *G* nejobecnější koncept a do *S* prvý pozitivní příklad.
2. Pro každý pozitivní příklad *p*
   a) Odstraň z G všechny prvky, které nepokrývají *p*.
   b) Každý prvek *S*, který nepokrývá *p*, nahraď jeho nejmenší generalizací, která pokrývá *p*.
   c) Odstraň z *S* všechny prvky, které jsou obecnější než nějaké jiné prvky *S* nebo obecnější než nějaké prvky *G*.
3. Pro každý negativní příklad *n*
   a) Odstraň z *S* všechny prvky, které pokrývají *n*.
   b) Každý prvek *G*, který pokrývá *n*, nahraď jeho nejmenší specializací, která nepokrývá *n*.

c) Odstraň z *G* všechny prvky, které jsou specifičtější než nějaké jiné prvky *G* nebo specifičtější než nějaké prvky *S*.

4. Jsou-li *G* a *S* prázdné, postup končí (neexistuje žádný koncept, který by pokrýval všechny pozitivní příklady a žádný negativní). Je-li *G* = *S* a obě množiny jsou jednoprvkové, postup rovněž končí (cílový koncept byl nalezen). V ostatních případech se vrať na krok 2.

6.2 Techniky rozhodovacích stromů

Tyto techniky vycházejí se z příkladů popsaných vektory atributů a indexy tříd. Vytváří se rozhodovací strom, jehož nelistové uzly odpovídají atributům a listové uzly jsou ohodnoceny indexy tříd (každému atributu i každé třídě může odpovídat několik uzlů). Hrany vycházející z nelistového uzlu jsou ohodnoceny hodnotami příslušného atributu.

Základní algoritmy jsou tyto:

- ID3 (Iterative Dichotomizer) – Quinlan
- TDITD (Top-Down Induction of Decision Trees)

Rozšířením algoritmu ID3 je systém C4.5.

Rozhodovací strom můžeme převést na soubor pravidel tak, že každé cestě z kořenového do listového uzlu odpovídá jedno pravidlo.

Algoritmus ID3

Vytvoření rozhodovacího stromu algoritmem ID3 probíhá v následujících krocích:

1. Vybere se jeden atribut jako kořen stromu.
2. Množina trénovacích příkladů se rozdělí na podmnožiny podle hodnot tohoto atributu.
3. Postupně se zpracují všechny tyto podmnožiny takto:
   a) Obsahuje-li podmnožina pouze příklady z téže třídy, vytvoří se pro tuto podmnožinu listový uzel a ohodnotí se indexem příslušné třídy.
   b) V opačném případě se vybere další atribut jako atribut podstromu. Tento atribut se pak zpracuje podle bodů 2 a 3.

Jako nejvhodnější může být vybrán např. atribut, který má

- nejmenší *entropii*
- největší informační zisk nebo největší poměrný informační zisk

Nechť A je atribut, a_1, a_2, …, a_n jsou jeho hodnoty a c_1, c_2, …, c_m jsou klasifikační třídy. Pak ***entropie*** atributu A se vypočte podle vztahů

$$H(A) = \sum_{j=1}^{n} P(a_j) H(a_j)$$

$$H(a_j) = -\sum_{i=1}^{m} P(c_i \mid a_j) \log\left(P(c_i \mid a_j)\right)$$

kde P znamená pravděpodobnost, logaritmus má základ větší než 1 (obvykle je základ roven 2) a pokud je pravděpodobnost rovna nule, pak se hodnota výrazu $P \cdot \log(P)$ považuje za nulovou.

Informační zisk je míra odvozená z entropie:

$$Zisk(A) = H(C) - H(A)$$

kde

$$H(C) = -\sum_{i=1}^{m} P(c_i) \log\left(P(c_i)\right)$$

Nevýhodou entropie a informačního zisku může být skutečnost, že neberou v úvahu počet hodnot atributu. Proto se někdy používá ***poměrný informační zisk***:

$$PoměrnýZisk(A) = \frac{Zisk(A)}{Větvení(A)}$$

kde

$$Větvení(A) = -\sum_{j=1}^{n} P(a_j) \log\left(P(a_j)\right)$$

Problémy rozhodovacích stromů a možnosti jejich odstranění

Vytvořený rozhodovací strom může být příliš rozsáhlý, což snižuje jeho srozumitelnost. Může také dojít k „přeučení“ (*overfitting*) rozhodovacího stromu, tj. k dosažení neúměrné přesnosti stromu v situaci, že trénovací data jsou zatížena šumem.

Strom je možno zjednodušit tak, že místo toho, aby listovému uzlu odpovídaly pouze příklady jedné třídy, se spokojíme s tím, že příklady jedné třídy budou v listovém uzlu převažovat. Zjednodušení (redukci) stromu je možno provést dvěma způsoby:

- Původní algoritmus se modifikuje doplněním nějakého kritéria, které indikuje, zda má uzel dále expandovat. Tímto způsobem se redukovaný strom vytvoří přímo.
- Vytvoří se úplný strom a následně se provede jeho prořezání (*post-prunning*). Postupuje se zdola nahoru a u každého podstromu se podle nějakého kritéria rozhoduje, zda se má podstrom nahradit listovým uzlem. Tento způsob je obvyklejší.

Zdokonalení technik rozhodovacích stromů

- Práce s numerickými atributy.

  V případě spojitých hodnot nebo velkého počtu diskrétních hodnot se obor hodnot rozdělí na intervaly, které se pak považují za diskrétní hodnoty atributu.

- Chybějící hodnoty atributů.

  Jednou z možností je uvažovat místo chybějící hodnoty nejčastější hodnotu příslušného atributu. Jinou možností je uvažovat všechny hodnoty atributu s vahami danými relativními četnostmi jejich výskytu v trénovacích datech.

- Ceny atributů.

  V některých aplikacích může hrát roli i cena zjištění hodnoty atributu. V takovém případě můžeme při tvorbě rozhodovacího stromu např. použít modifikované kritérium informačního zisku

$$ZC(A) = \frac{Zisk(A)^2}{Cena(A)}$$

6.3 Techniky rozhodovacích pravidel

Množiny rozhodovacích pravidel můžeme odvodit z rozhodovacích stromů. Jiný způsob představují algoritmy pokrývání množin, jako např.:

- algoritmus AQ (Michalski)
- algoritmy CN2 (Clark, Niblet), CN4 (Bruha, Kočková)

U výše uvedených technik vytvářená pravidla neobsahují proměnné. Naproti tomu v *induktivním logické programování* pravidla mohou obsahovat proměnné a mají tvar Hornových klauzulí 1.řádu (prologovských klauzulí). K systémům induktivního logického programování ptří např.:

- FOIL (Quinlan)
- PROGOL (Muggleton)
- TILDE (Blockeel, De Raedt)

Algoritmus AQ

Algoritmus AQ je určen rovněž pro učení z klasifikovaných příkladů popsaných pomocí atributů. Jeho výstupem je soubor pravidel popisujících všechny pozitivní příklady z trénovací množiny.

Algoritmus lze shrnout do následujících bodů:

1. Rozděl množinu příkladů na dvě podmnožiny: podmnožinu *P* pozitivních příkladů a podmnožinu *N* negativních příkladů.
2. Vyber z množiny *P* jeden příklad a označ jej *s* (*seed* neboli *jádro*).
3. Nalezni všechny maximální generalizace popisu jádra *s* tak, že jimi nesmí být pokryt žádný negativní příklad.
4. Podle vhodného preferenčního kritéria vyber nejlepší z těchto popisů a zařaď jej do množiny popisů. Odstraň z množiny *P* všechny příklady pokryté tímto popisem.
5. Je-li množina *P* prázdná, ukonči práci (výsledným popisem je disjunkce všech nalezených popisů). V opačném případě se vrať na bod 2.

Možná zdokonalení algoritmu AQ:

- Případ více tříd.

  Základní podoba algoritmu uvažuje dvě třídy. Rozšíření se obvykle provede tak, že pro každou třídu c se za pozitivní považují příklady této třídy a za negativní všechny ostatní. Jinou možností je vytvářet pravidla ke všem třídám současně (tak pracují algoritmy CN2, CN4).

- Data zatížená šumem.

  Algoritmus můžeme upravit tak, že v kroku 3 nepožadujeme, aby pravidlo pokrývalo příklady pouze jedné třídy.

- Rozhodovací seznam.

  Algoritmus AQ vytváří tzv. neuspořádaný soubor pravidel. Opakem je uspořádaný soubor pravidel, neboli rozhodovací seznam (*decision list*), ve kterém jsou pravidla propojena pomocí ELSE. Podmínka za ELSE IF tedy implicitně obsahuje negaci podmínek všech předcházejících pravidel. Algoritmy CN2 a CN4 umožňují tvorbu uspořádaného i neuspořádaného souboru pravidel.

6.4 Konceptuální shlukování

Algoritmus konceptuálního shlukování sestává z těchto kroků:

1. Z množiny všech příkladů vyber k jader $s_1, \ldots, s_k$ (hodnotu k zadává uživatel). Výběr může být proveden náhodně nebo pomocí nějaké selekční funkce.
2. Pro každé jádro vytvoř maximálně obecný popis, který by je odlišil od všech ostatních jader. Každý takový popis je nazýván *protokonceptem*. Vytvořené protokoncepty nemusejí být disjunktní.
3. Uprav protokoncepty tak, aby byly disjunktní. Zpravidla existuje několik různých řešení (množin konceptů), z nichž se pomocí vyhodnocovací funkce vybere to nejlepší.
4. Není-li splněno zvolené kritérium ukončení, vyber nová jádra (z každého stávajícího protokonceptu se vybere jedno) a přejdi zpět ke kroku 2. V opačném případě vyber nejlépe ohodnocené řešení a proces ukonči.

Disjunktní koncepty mohou být vytvořeny např. takto:

1. Vytvoří se seznam L příkladů, pokrytých více protokoncepty.
2. Jako počáteční hodnoty vytvářených disjunktních konceptů C_i se položí $C_i = \{s_i\}$ ($i = 1, \ldots, k$).
3. Vezme se první příklad ze seznamu L a odstraní se z tohoto seznamu.
4. Pro každý z protokonceptů P_i pokrývajících vybraný příklad e se pokusně vytvoří nový co nejobecnější popis tak, aby pokrýval příklad e a shluk C_i a nepřekrýval se s žádným z ostatních konceptů.

5. Pomocí vyhodnocovací funkce vybereme nejlepší z těchto popisů. Nechť je to popis odpovídající protokonceptu P_j. Příklad e přidáme ke shluku C_j . Je-li seznam L prázdný, postup končí. V opačném případě se vracíme na krok 3.

Poznámky ke konceptuálnímu shlukování

- Vyhodnocovací funkce

  Může zahrnovat řadu kritérií, jako např. míru rozdílnosti mezi shluky, počet proměnných umožňujících diskriminovat mezi shluky, jednoduchost popisu shluků, apod.

- Ukončovací kritérium

  Může to být např. počet iterací, po které nedošlo k významnému zlepšení vyhodnocovací funkce.

- Volba jader ze stávajících shluků

  V algoritmu CLUSTER/2 (Michalski) se s využitím vhodné metriky vybírá prvek nejbližší středu každého shluku. V případě neuspokojivých shluků, které se po několik iterací nezlepšují, se jako nová jádra vybírají prvky nejbližší k hranici shluku.

6.5 Učení založené na vysvětlování

Učení založené na vysvětlování obsahuje tyto složky:

- Cílový koncept, jehož definici je třeba vytvořit.
- Trénovací příklad jako instance cíle.
- Doménová teorie (množina pravidel a faktů, které jsou používány k vysvětlení, proč je trénovací příklad instancí cílového konceptu).
- Kritéria popisující tvar, který definice cíle mohou nabývat.

V prvém kroku se pomocí doménové teorie konstruuje vysvětlení trénovacího příkladu. Obvykle se jedná o důkaz, že příklad logicky vyplývá z teorie, přičemž tento důkaz může být reprezentován např. pomocí stromové struktury.

V dalším kroku je zobecněním tohoto vysvětlení získána definice cílového konceptu. Zobecnění spočívá v substituci proměnných za ty konstanty v důkazním stromu, které závisejí pouze na trénovacím příkladu. Na základě zobecněného důkazního stromu se definuje pravidlo, jehož závěr je kořen stromu a předpoklad je konjunkcí listů.

Existuje řada způsobů konstrukce zobecněného důkazního stromu. Jednou z možností je (Mitchell et al.) nejprve zkonstruovat důkazní strom, který je specifický pro trénovací příklad, a následně tento důkaz zobecnit tzv. cílovou regresí (*goal regression*). Cílová regrese unifikuje zobecněný cíl s kořenem důkazního stromu pomocí potřebných substitucí. Algoritmus rekurzívně aplikuje tyto substituce na další uzly stromu, dokud všechny vhodné konstanty nejsou zobecněny.

DeJong and Mooney navrhují alternativní přístup, který v podstatě spočívá v paralelním vytváření zobecněného i specifického stromu. To je uskutečněno pomocí tzv. vysvětlovací struktury (*explanation structure*), která reprezentuje abstraktní strukturu důkazu. Při vytváření

této struktury se konstruují dva substituční seznamy, jeden pro specifický strom a druhý pro obecný strom.

6.6 Metody založené na analogii

Učení založené na analogii znamená učení nových konceptů nebo odvozování nových řešení prostřednictvím použití podobných konceptů a jejich řešení. K základním metodám patří:

- Případové usuzování (*Case-Based Reasoning*, CBR)
- Pravidlo nejbližšího souseda (*nearest neigbour rule*)
- Učení založené na instancích (*Instance-Based Learning*, IBL)
- Líné učení (*lazy learning*)
- Paměťové učení (Memory-Based Learning)

Znalosti jsou reprezentovány v podobě báze již vyřešených problémů. Při usuzování se k danému problému v této bázi hledá nejpodobnější případ a jeho řešení se adaptuje na novou situaci. Fáze učení pak představuje zapamatování znalosti z vyřešeného problému pro budoucí použití. Při učení se tedy neprovádí generalizace z příkladů (proto název *líné učení*).

Případové usuzování

CBR je založeno na vyhledávání a přizpůsobování starých řešení novým problémům. Obecný CBR cyklus lze popsat pomocí následujících čtyř kroků:

1. Vyhledání (*retrieving*) nejvíce podobného případu nebo případů.
2. Použití (*reusing*) informací a znalostí z tohoto případu k vyřešení daného problému.
3. Revize (*revising*) navrženého řešení.
4. Uchování (*retaining*) částí těchto zkušeností tak, aby byly použitelné pro řešení budoucích problémů.

Kromě specifických znalostí reprezentovaných případy hrají určitou roli v tomto procesu i obecné znalosti z příslušného oboru.

V nejjednodušší podobě je případ reprezentován pomocí uspořádané dvojice (*problém*, *řešení*), kde problém je obvykle popsán vektorem hodnot atributů $(a_1, \ldots, a_n)$. Tato forma odpovídá pravidlu $a_1, \ldots, a_n \rightarrow$ *řešení* . Atribut může být číselný (diskrétní nebo spojitý), ostrý symbolický (jeho hodnotami jsou řetězce znaků s určitými významy) nebo vágní lingvistický (každá lingvistická hodnota je reprezentována pomocí odpovídající fuzzy množiny). V jistých situacích může být řešením akce s účinky, které nejsou a priori známé a dokonce ani určené zadanými daty. To vede k rozšíření konceptu případu na trojici (*problém*, *řešení*, *účinky*).

Existují dva hlavní přístupy k vyhledávání podobných případů:

- První je výpočetní přístup, kde se podobnost mezi případy počítá z hodnot atributů tvořících případ.

- Druhý je indexační nebo reprezentační přístup. Podobnost mezi případy je zakódována přímo do struktury báze případů. Případy jsou propojeny indexovými strukturami, které umožňují vyhledání podobného případu.

Poznamenejme, že předchozí dva přístupy mohou být použity i v kombinaci.

Prvním krokem v CBR je vyhledávání podobných případů a velmi důležitou roli při něm hraje podobnostní míra. Pojetí *relace podobnosti* je zobecněním pojmu relace ekvivalence, která je reflexivní symetrická a tranzitivní. Relace podobnosti může být definována jako zobrazení *sim*: $U \times U \rightarrow [0, 1]$, splňující tyto tři následující vlastnosti:

$$\forall\, x \in U,\ sim(x, x) = 1, \qquad \text{(reflexivita)}$$

$$\forall\, x, y \in U,\ sim(x, y) = sim(y, x), \qquad \text{(symetrie)}$$

$$\forall\, x, y, z \in U,\ sim(x, y) \otimes sim(y, z) \leq sim(x, z), \qquad (\otimes \text{ - tranzitivita})$$

kde $\otimes$ je binární operace na jednotkovém intervalu s některými doplňujícími vlastnostmi. Tato operace je obvykle *t-norma*, t.j. $\otimes$ je operace na [0, 1] neklesající v obou argumentech, která je asociativní, komutativní a pro niž platí $1 \otimes a = a$ (1 je neutrální prvek), $0 \otimes a = 0$ (0 je absorbující prvek).

Podobnostní míra na množině hodnot atributu se nazývá *lokální míra* zatímco míra na celém objektu se nazývá *globální míra*. Jedním ze základních úkolů v CBR je spojit vhodným způsobem lokální podobnostní míry do globální podobnostní míry. Přehled podobnostních měr je podán v práci (Dvořák a Hodál 2001). Práce (Dvořák a Šeda 2004) obsahuje srovnání existujících a nově navržených podobnostních měr pro fuzzy čísla).

Případové usuzování může být použito např. pro navigaci mobilních robotů. Existující přístupy jsou popsány v (Dvořák a Hodál 2003). Základní přístup při globální navigaci robota je takový, že případy jsou tvořeny celými cestami, které robot již úspěšně absolvoval. Je-li zadán nový problém počátečním a cílovým bodem pak za podobný případ se považuje cesta, jejíž jeden krajní bod leží v blízkosti zadaného počátku a druhý v blízkosti zadaného cíle. Adaptace pak spočívá v doplnění úseků spojujících zadané body s krajními body vybrané cesty. Lepšího využití dosud získaných zkušeností robota se dosáhne, když je možné jako podobné případy použít nejen celé existující cesty, ale také jejich části. Takové metody jsou navrženy v práci (Dvořák a kol. 2004).

6.7 Bayesovské učení

Bayesovské uvažování je jednak základem pro algoritmy učení, které přímo manipulují s pravděpodobnostmi, a jednak je také je nástrojem pro analýzu a pochopení ostatních algoritmů strojového učení. Výhodou metod bayesovského učení je schopnost klasifikovat příklady do tříd s určitou pravděpodobností.

Metody Bayesovského učení zahrnují:

- Bayesovský optimální klasifikátor
- Gibbsův algoritmus
- Naivní bayesovský klasifikátor

- Učení bayesovských sítí

Naivní bayesovský klasifikátor vychází z předpokladu, že jednotlivé evidence E_1, … E_K jsou podmíněně nezávislé při platnosti hypotézy H. Pak platí

$$P(H \mid E_1,...,E_K) = \frac{P(H)}{P(E_1,...,E_K)} \prod_{i=1}^{K} P(E_i \mid H)$$

Naivní bayesovský klasifikátor je pak dán vztahem:

$$H_{NB} = \arg\max_{H_j} P(H_j) \prod_{i=1}^{K} P(E_i \mid H_j)$$

7. Systém CLIPS

Zkratka CLIPS znamená *C Language Integrated Production System*. CLIPS je prostředí pro vývoj expertních systémů. Znalosti v CLIPSu mohou být reprezentovány pomocí *pravidel*, *funkcí* a *objektů*. Vestavěný inferenční mechanismus rozhoduje o tom, která pravidla mají být použita a kdy. Tento mechanismus je založen na dopředném řetězení, porovnávání se vzorem a algoritmu Rete. Formát CLIPSu je podobný jazyku LISP. Na rozdíl od LISPu však CLIPS rozlišuje malá a velká písmena.

Hlavními syntaktickými kategoriemi v CLIPSu jsou:

- Konstrukty (pro definici faktů, pravidel, šablon, funkcí, tříd, objektů, globálních proměnných, …)
- Příkazy (pro ladění, práci s konstrukty, fakty, prostředím, …)
- Funkce (matematické, predikátové, vstupní a výstupní, procedurální, řetězcové, pro třídy, objekty, pravidla, šablony, fakty, …)

Na nejvyšší úrovni se program v CLIPSu se skládá z:

- konstruktů (konstrukty dále mohou obsahovat příkazy, volání funkcí a poznámky)
- poznámek (poznámka je libovolný text začínající středníkem a končící znakem řádku).

7.1 Fakty

Fakt (*fact*) je jedno nebo více polí (*fields*) uzavřených do kulatých závorek. Pole mohou být pojmenovaná nebo nepojmenovaná (jména polí se zavádějí v definici šablony). U nepojmenovaných polí záleží na jejich pořadí a prvé pole obvykle popisuje vztah mezi ostatními fakty. Pole se oddělují mezerami, tabulátory nebo řádky. Pojmenovaná pole (*rubriky, sloty*) mají tvar:

```
(jméno_rubriky hodnota_1 … hodnota_n)
```

Případ $n > 1$ se týká tzv. vícepolové rubriky.

Není dovoleno vnořovat fakty do jiných faktů a čísla nemohou vystupovat jako samostatné fakty.

Hodnoty polí ve faktu mohou mohou mít tyto typy: `FLOAT, INTEGER, SYMBOL, STRING, FACT-ADRESS`. *Symbol* začíná tisknutelným ASCII znakem, za nímž následuje jeden nebo více tisknutelných ASCII znaků. CLIPS rozlišuje velká a malá písmena. *String* musí začínat a končit uvozovkami.

Pokud pole nemá žádnou hodnotu, zapisuje se na jeho pozici speciální symbol *nil*. Logické hodnoty jsou v CLIPSu reprezentovány pomocí symbolů TRUE a FALSE.

Fakty jsou organizovány v bázi (seznamu) faktů, přičemž jsou jim automaticky přidělovány identifikátory. Fakt je v bázi faktů označen identifikátorem faktu

```
f-index_faktu
```

Index faktu je celé číslo bez znaménka. Číslování začíná od nuly a zvyšuje se s krokem 1. Nula je rezervována pro speciální fakt (*initial-fact*).

Vkládání faktů do báze faktů se provádí pomocí funkce *assert* nebo pomocí konstruktu *deffacts* ve spojení s příkazem *reset*. Rušení faktů se provádí pomocí funkce *retrac*t nebo pomocí konstruktu *undeffacts* ve spojení s příkazem *reset*.

Je možné najednou definovat celou skupinu faktů pomocí konstruktu

```
(deffacts jméno_skupiny_faktů
          "nepovinná poznámka"
          fakt_1
          ...
          fakt_n)
```

Definované fakty jsou do báze faktů vloženy příkazem *reset*. Odstranění skupiny faktů daného jména se zajistí pomocí konstruktu

```
(undeffacts jméno_skupiny_faktů)
```

ve spolupráci s příkazem *reset*.

7.2 Šablony

Šablona je seznam pojmenovaných polí, zvaných *rubriky* (*sloty*). Šablony umožňují používat fakty, na jejichž pole se můžeme odvolávat pomocí jejich jmen namísto pomocí specifikace jejich pozic. Šablony se definují pomocí konstruktu *deftemplate*:

```
(deftemplate jméno_šablony
             "nepovinná poznámka"
             definice_rubriky_1
             ...
             definice_rubriky_n)
```

Rubriky mohou být *jednopolové* (*slots*) nebo *vícepolové* (*multislots*). Jednopolová rubrika obsahuje právě jedno pole, kdežto vícepolová rubrika může obsahovat jedno, více nebo žádné pole.

- Definice jednopolové rubriky:

```
(slot jméno_rubriky atribut_1 … atribut_m)
```

- Definice vícepolové rubriky:

```
(multislot jméno_rubriky atribut_1 … atribut_m)
```

Atributem může být buď atribut *default* nebo atribut omezení.

Atribut *default* určuje implicitní (defaultní) hodnotu, která se při vytváření faktu vloží do slotu, jehož hodnota nebyla zadána.

```
(default výraz_1 … výraz_n)
```

Jednotlivé výrazy předepisují implicitní hodnoty. Více hodnot se objevuje v případě multislotu.

Atribut default může být zadán také těmito způsoby:

```
(default ?DERIVE)
(default ?NONE)
```

Klíčové slovo ?DERIVE určuje, že do příslušného slotu bude automaticky dosazena vhodná hodnota. Pro typ SYMBOL je to *nil*, pro typ STRING je to prázdný řetězec, pro typ FLOAT je to dolní mez rozsahu povolených hodnot (pokud rozsah není zadán, tak je to 0.0), pro typ INTEGER je to prvá hodnota ze seznamu povolených hodnot resp. dolní mez rozsahu (jinak je to 0). Klíčové slovo ?NONE znamená, že se žádná implicitní hodnota nedosazuje a že hodnota slotu musí být zadána.

Atributy omezení:

- Atribut typu:

  ```
  (type specifikace_typu)
  ```

  Možné jsou specifikace: SYMBOL, STRING, LEXEME (*symbol* nebo *string*), INTEGER, FLOAT, NUMBER, FACT-ADDRESS.

- Atribut rozsahu (meze jsou numerické konstanty):

  ```
  (range mez_1 mez_2)
  ```

- Atribut kardinality (meze jsou celočíselné konstanty):.

  ```
  (cardinality mez_1 mez_2)
  ```

  Tento atribut se používá u vícepolových rubrik. Počet polí vícepolové rubriky se musí pohybovat v zadaných mezích.

- Atribut povolených hodnot:

  ```
  (allowed-values hodnota_1 … hodnota_n)
  ```

7.3 Pravidla

Definice pravidla má následující tvar:

```
(defrule jméno_pravidla "nepovinná poznámka"
          vzor_1     ;levá strana pravidla
          ...
          vzor_n
          =>
          akce_1     ;pravá strana pravidla
          ...
          akce_m )
```

Vzor je vlastně jakási podmínka, která se testuje pro fakty v bázi faktů. Akcí může být volání funkce nebo příkaz.

Pravidlu může být explicitně přidělena priorita deklarací

```
(declare (salience hodnota))
```

která se zapisuje bezprostředně před prvý vzor v levé straně pravidla. Možné hodnoty priority se mohou nacházet v rozsahu od –10000 do 10000. Jestliže priorita není explicitně určena, CLIPS ji implicitně nastavuje na hodnotu nula.

7.4 Proces inference

CLIPS zkouší ztotožnit vzory pravidel s fakty v seznamu faktů. Jestliže se podařilo ztotožnit všechny vzory v levé straně pravidla s fakty, pravidlo je aktivováno a uloženo do *agendy*. Agenda může obsahovat jedno nebo více aktivních pravidel.

Je-li v agendě více aktivních pravidel, CLIPS automaticky určuje, které aktivní pravidlo je vhodné ke spuštění. Aktivní pravidla jsou v agendě uspořádána podle rostoucí priority (*salience*) a vybíráno je pravidlo s nejvyšší prioritou. Jestliže existuje více pravidel se stejnou prioritou, pak se uplatňuje metoda LIFO.

CLIPS provede akce pravé strany vybraného pravidla. Toto pravidlo je pak odstraněno z agendy. V každém prováděcím cyklu je provedeno jedno vybrané pravidlo. Tento proces pokračuje tak dlouho, dokud agenda není prázdná nebo dokud není dán příkaz k zastavení.

Opětovná aktivace a deaktivace pravidla se řídí těmito zásadami:

- Žádné pravidlo se nepoužije vícekrát se stejnou množinou faktů.
- Použité pravidlo je opět aktivováno, jestliže jeho vzory jsou ztotožněny:
  - s novými fakty, které předtím neexistovaly, nebo
  - s fakty, které předtím existovaly, ale byly odstraněny a znovu vloženy.

Nepoužité aktivní pravidlo je z agendy odstraněno, jestliže se změnily podmínky nutné k jeho aktivaci. Tento účinek má např. příkaz *reset*, který odstraní existující fakty ze seznamu faktů.

7.5 Vzory

Možné podoby vzorů:

- Uspořádaný vzor:

  (*symbol omezení_1 … omezení_n*), $n \geq 0$

  Uspořádaný vzor se používá pro porovnávání s fakty, jejichž pole nejsou pojmenována a tudíž záleží na jejich pořadí.

- Vzor šablony:

  (*jméno_šablony vzor_1_rubriky … vzor_n_rubriky*)

Vzor jednopolové rubriky:

(*jméno_rubriky omezení*)

Vzor vícepolové rubriky:

(*jméno_rubriky omezení_1 … omezení_m*)

- Přiřazení vzoru:

jednopolová_proměnná <- jednoduchý_vzor

Na proměnnou se naváže adresa faktu, který se ztotožní se vzorem (tato adresa může být pak v pravé část pravidla využita např. k odstranění pravidla pomocí příkazu *retract*).

- Testovací podmínka:

(test *volání_funkce*)

- Negace:

(not *vzor*)

Dojde ke ztotožnění, právě když se v bázi faktů nevyskytuje žádný fakt, který by vyhovoval danému vzoru.

- Konjunkce:

(and *vzor_1 … vzor_n*), *n*≥1

Musejí být ztotožněny všechny uvedené vzory.

- Disjunkce:

(or *vzor_1 … vzor_n*), *n*≥1

Stačí ztotožnění alespoň jednoho z uvedených vzorů.

Typy omezení polí ve vzoru:

- Wildcard:

? (zastupuje právě jedno pole)

$? (zastupuje žádné, jedno nebo více polí)

Tyto zástupné znaky používáme, pokud nás nezajímá hodnota pole, ale pouze jeho přítomnost.

- Jednoduché omezení:

term

~*term* (symbol ~ zakazuje bezprostředně následující hodnotu)

Možné tvary termu:

konstanta,

proměnná,

:*volání_funkce*,

```
=volání_funkce
```

(znaky : a = způsobí vyhodnocení následujícího výrazu).

- Složené omezení z jednoduchých omezení a infixových spojek:

  | … povoluje kteroukoli hodnotu ze spojených omezení

  & … způsobí současné použití spojených omezení

7.5 Proměnné

Proměnné mohou být lokální a globální, jednopolové a vícepolové. Identifikátor lokální jednopolové proměnné má tvar:

```
?jméno_proměnné
```

Je-li porovnání vzoru obsahujícího proměnnou s faktem úspěšné, je na proměnnou navázána jí odpovídající hodnota z faktu. Tato hodnota proměnné zůstává jak na levé, tak i na pravé straně pravidla, dokud není změněna na pravé straně pravidla. Navázání hodnoty k proměnné na pravé straně pravidla je možné pomocí funkce *bind*. Teprve poté, co je na proměnnou navázána nějaká hodnota, je možné tuto proměnnou použít.

Vazba na hodnotu je pro lokální proměnné platná pouze v rámci prováděného pravidla (pouze globální proměnné jsou vázány ve všech pravidlech).

Použijeme-li v identifikátoru proměnné znak $, jedná se o vícepolovou proměnnou, na kterou může být navázána žádná, jedna nebo více hodnot:

```
$?jméno_proměnné
```

Je-li vícepolová proměnná uvedena na pravé straně pravidla, může se znak $ vynechat. Vícepolové hodnoty systém vrací uzavřené do závorek.

Identifikátor globální proměnné vypadá takto:

```
?*jméno_proměnné*
```

Globální proměnné spolu s jejich výchozími hodnotami se definují pomocí konstruktu *defglobal*:

```
(defglobal přiřazení_1 … přiřazení_n)
```

Přiřazení výchozí hodnoty globální proměnné:

```
globální_proměnná = výraz
```

Hodnotu globální proměnné je možno změnit pomocí funkce *bind*.

7.6 Funkce

Volání funkce má následující podoby:

- Volání funkce která nemá argumenty:

```
(jméno_funkce)
```

- Volání funkce s argumenty:

```
(jméno_funkce arg_1 … arg_n)
```

U většiny funkcí argumenty jsou výrazy, které se nejprve všechny vyhodnotí a na získané hodnoty se pak aplikuje funkce. Výjimkou jsou např. procedurální funkce, kde se s argumenty zachází jiným způsobem.

V Clipsu se vyskytují následující předdefinované funkce:

- Matematické funkce
- Predikátové funkce
- Vstupní a výstupní funkce
- Pro zpracování sekvencí a vícepolových hodnot
- Řetězcové funkce
- Procedurální funkce
- Pro získání informací o definovaných faktech, šablonách, pravidlech, funkcích, modulech.
- Pro práci s fakty
- Pro práci s agendou
- Pro práci se třídami a objekty
- Pro práci s generickými funkcemi
- Různé speciální funkce

Uživatel také může definovat svoje funkce pomocí konstruktu *deffunction*:

```
(deffunction jméno_funkce "nepovinná poznámka"
          (par_1 … par_n);seznam parametrů
          akce_1
          ...
          akce_m)
```

Seznam parametrů může být i prázdný. Parametry se zapisují jako jednopolové proměnné, pouze poslední z nich může být vícepolová proměnná. Pokud není výsledná hodnota funkce určena akcí *return*, je určena závěrečnou akcí (pokud tato akce má hodnotu).

Z matematických funkcí jsou k dispozici např. tyto:

+, –, *, /, *div* (celočíselné dělení), *mod* (zbytek po celočíselném dělení), ** (mocnina),

max, min, abs, exp, *log*, *log10*, *sqrt* (druhá odmocnina), *round* (zaokrouhlení).

Predikátové funkce nabývají hodnot TRUE nebo FALSE. Mezi tyto funkce patří:

- srovnávací funkce: *eq* (rovno pro lib.typy), *neq* (nerovno pro lib. typy), = (numericky rovno), <> (numericky nerovno), >= , > , <= , <

- logické funkce: *not*, *and*, *or*
- funkce pro testování typu: *floatp*, *integerp*, *numberp*, *symbolp*, *stringp*, *lexemep* (symbol nebo string)

7.6.1 Vstupní a výstupní funkce

Vstup z klávesnice zajišťují funkce *read* a *readline*:

```
(read)
(readline)
```

Hodnotou těchto funkcí jsou přečtené údaje. Funkce *read* přečte jeden údaj a pak přejde na nový řádek. Funkce *readline* přečte všechny údaje až do konce řádku a vytvoří z nich řetězec.

Výstup na obrazovku zajišťuje funkce *printout*

```
(printout t výraz_1 … výraz_n)
```

Parametr *t* určuje, že se jedná o obrazovku.

Vstup a výstup při práci se soubory zajišťují opět funkce *read*, *readline* a *printout*. V tom případě musí být jako první argument těchto funkcí uvedeno jako logické jméno souboru.

```
(read logické_jméno)
(readline logické_jméno)
(printout logické_jméno výraz_1 … výraz_n)
```

Před čtením/zapisováním ze/do souboru musí být soubor otevřen pomocí funkce *open* a po skončení práce uzavřen pomocí funkce *close*.

```
(open jméno_souboru logicke_jméno)
(close logicke_jméno)
```

7.6.2 Vícepolové funkce

Vícepolové funkce jsou určeny pro vytváření nebo rozklad vícepolových hodnot. Jména těchto funkcí končí znakem *$* (výjimkou jsou predikátové funkce, které končí písmenem *p*).

- Funkce *create$*:

  ```
  (create$ výraz_1 … výraz_n)
  ```

  Hodnoty argumentů se spojí do vícepolové hodnoty. Volání této funkce bez argumentů vytvoří prázdnou vícepolovou hodnotu `()`.

- Funkce *insert$*:

  ```
  (insert$ vícepolový_výraz
           poziční_výraz výraz_1 … výraz_n)
  ```

  Jedna nebo více hodnot se vloží do vícepolové hodnoty na místo určené pozičním výrazem.

- Funkce *replace$*:

```
(replace$ vícepolový_výraz
          pozice_1 pozice_2 výraz_1 … výraz_n)
```

  Část vícepolové hodnoty specifikovaná pozičními výrazy se nahradí hodnotami zadaných výrazů.

- Funkce *delete$*:

```
(delete$ vícepolový_výraz pozice_1 pozice_2)
```

  Zruší se část vícepolové hodnoty specifikovaná pozičními výrazy.

- Funkce *explode$*:

```
(explode$ řetězcový_výraz)
```

  Z řetězce se vytvoří vícepolová hodnota.

- Funkce *implode$*:

```
(implode$ vícepolový_výraz)
```

  Z vícepolové hodnoty se vytvoří řetězec.

- Funkce *first$*:

```
(first$ vícepolový_výraz)
```

  Pomocí této funkce se získá hodnota prvého pole vícepolové hodnoty.

- Funkce *rest$*:

```
(rest$ vícepolový_výraz)
```

  Tato funkce vrátí zbytek vícepolové hodnoty bez prvého pole.

- Funkce *nth$*:

```
(nth$ poziční_výraz vícepolový_výraz)
```

  Získá se hodnota pole specifikovaná pozičním výrazem.

- Funkce *subseq$*:

```
(subseq$ vícepolový_výraz pozice_1 pozice_2)
```

  Z vícepolové hodnoty se extrahuje část specifikovaná pozičními výrazy.

- Funkce *length$*:

```
(length$ vícepolový_výraz)
```

  Hodnotou této funkce je počet polí zadané vícepolové hodnoty.

- Funkce *member$*:

```
(member$ jednopolový_výraz vícepolový_výraz)
```

  tato funkce určí pozici zadané jednopolové hodnoty ve vícepolové hodnotě.

- Funkce *subsetp*:

  ```
  (subsetp vícepol_výraz_1 vícepol_výraz_2)
  ```

  Tato predikátová funkce nabývá hodnotu TRUE, je-li hodnota prvého argumentu podmnožinou hodnot druhého argumentu.

7.6.3 Procedurální funkce

Procedurální funkce vlastně představují příkazy. Můžeme je rozdělit na:

- Jednoduché procedurální funkce:

  Tyto funkce zajišťují provedení jediné specializované akce. Jedná se např. o funkce *bind*, *return*, *break*.

- Strukturované procedurální funkce:

  Tyto funkce organizují provádění akcí. Může jít o:

  - provedení skupiny akcí (funkce *progn* a *progn$*)
  - podmíněné provedení akcí (funkce *if* a *switch*)
  - opakování akcí (funkce *loop-for-count* a *while*)

  Uvedené funkce jsou obdobou strukturovaných příkazů známých z imperativních programovacích jazyků.

Vybrané jednoduché procedurální funkce:

- Funkce *bind* přiřadí proměnné novou hodnotu.

  ```
  (bind proměnná výraz_1 … výraz_n)
  ```

- Funkce *return* ukončí vykonávání funkce definované pomocí konstrukce *deffunction* a v případě specifikované hodnoty je výsledkem vykonání tato hodnota.

  ```
  (return)
  (return výraz)
  ```

- Funkce *break* okamžitě ukončí právě prováděný cyklus.

  ```
  (break)
  ```

Provedení skupiny akcí:

- Funkce *progn* zajistí provedení skupiny akcí. Výslednou hodnotou je hodnota poslední akce.

  ```
  (progn akce_1 … akce_n)
  ```

- Funkce *progn$* provede skupinu akcí pro každé pole vícepolové hodnoty.

  ```
  (progn$ vícepolový_výraz akce_1 … akce_n)
  ```

Podmíněné akce:

- Funkce *if* zajistí podmíněné provedení skupiny akcí.

```
(if podminka then akce_1 … akce_m)
(if podmínka  then akce_1 … akce_m
              else akce_1 … akce_n)
```

- Funkce *switch* umožní větvení výpočtu do více větví.

  ```
  (switch testový_výraz případ_1 … případ_m)
  ```

  Případy mají tvar:

  ```
  (case srovnávací_výraz then akce_1 … akce_n)
  ```

  Poslední případ může mít také tento tvar:

  ```
  (default akce_1 … akce_n)
  ```

Opakované akce:

- Příkaz *loop-for-count* umožní iterativní vykonávání akcí v cyklu se specifikovaným počtem opakování.

  ```
  (loop-for-count specifikace_rozsahu do akce_1 … akce_n)
  ```

  Specifikace rozsahu může mít tyto podoby:

  ```
  ukončovací_výraz
  (řídicí_proměnná_cyklu  počáteční_výraz
                          ukončovací_výraz)
  ```

- Příkaz *while* umožní iterativní vykonávání akcí v cyklu bez předem specifikovaného počtu opakování.

  ```
  (while podmínka do akce_1 … akce_n)
  ```

7.6.4 Vybrané funkce pro práci s fakty

- Funkce *assert* zajistí přidání faktu do seznamu faktů.

  ```
  (assert fakt)
  ```

- Funkce *retract* zajistí odstranění faktu ze seznamu faktů (index odstraněného faktu zůstává nevyužit).

  ```
  (retract specifikátor_faktu)
  ```

  Specifikátor faktu je buď index faktu nebo proměnná, jejíž hodnotou je adresa faktu v bázi faktů.

- Funkce *modify* modifikuje obsah specifikovaných rubrik. Provedení této akce znamená odstranění původního faktu z báze faktů a vložení nového faktu obsahující modifikované hodnoty.

  ```
  (modify specifikátor_faktu rubrika_1 … rubrika_m)
  ```

Specifikace rubriky má tvar:

```
(jméno_rubriky výraz_1 … výraz_n)
```

7.6.5 Vybrané funkce pro práci s agendou

- Funkce *agenda* vypíše seznam aktivovaných pravidel.

```
(agenda)
```

- Funkce *halt* ukončí provádění pravidla.

```
(halt)
```

- Funkce *get-strategy* vypíše aktuální strategii řešení konfliktu.

```
(get-strategy)
```

- Funkce *set-strategy* nastavuje strategii řešení konfliktu (strategii výběru pravidla z agendy).

```
(set-strategy typ_strategie)
```

Jednoduché strategie řešení konfliktu (týkají se pravidel se stejnou prioritou):

- Strategie ***depth***:

  Přednost se dává pravidlům, aktivovaným s novějšími daty. Tato strategie je implicitní.

- Strategie ***breadth***:

  Přednost se dává pravidlům aktivovaným se staršími daty.

- Strategie ***simplicity***:

  Přednost se dává nově aktivovaným jednodušším (obecnějším) pravidlům (pravidlům s menším počtem testovaných podmínek).

- Strategie ***complexity***:

  Přednost se dává nově aktivovaným složitějším (specifičtějším) pravidlům (pravidlům s větším počtem testovaných podmínek).

- Strategie ***random***:

  Pravidla se vybírají náhodně.

Složené strategie řešení konfliktu:

- Strategie ***lex***:

  Pravidla se stejnou prioritou se setřídí podle aktuálnosti dat s nimiž byla aktivována od nejaktuálnějších k nejméně aktuálním (strategie *depth*). Jestliže dvě pravidla mají stejnou aktuálnost, vybere se specifičtější pravidlo (strategie *complexity*).

- Strategie ***mea***:

  Pravidla se stejnou prioritou se setřídí podle aktuálnosti dat odpovídajících první testované podmínce. Jestliže neexistuje jasný vítěz, uplatní se strategie *lex*.

7.6.6 Vybrané funkce (příkazy) pro práci s prostředím

- Příkaz *reset* zruší fakty v bázi faktů, vloží fakt (*initial-fact*) s identifikátorem f-0 do báze faktů, vloží fakty definované konstruktem *deffacts* do báze faktů, odstraní fakty definované konstruktem *undeffacts* z báze faktů.

```
(reset)
```

- Příkaz *batch* provádí příkazy ze souboru.

```
(batch jméno_souboru)
```

- Příkaz *clear* vymaže prostředí CLIPSu.

```
(clear)
```

- Příkaz *exit* opustí prostředí CLIPSu.

```
(exit)
```

- Příkaz *load* zavede konstrukty ze souboru.

```
(load jméno_souboru)
```

- Příkaz *save* uloží konstrukty do souboru.

```
(save jméno_souboru)
```

7.7 Objekty v CLIPSu

7.7.1 Třídy a objekty

Základním kamenem objektové technologie je *třída*. Třída v CLIPSu je šablona, která popisuje společné atributy a chování objektů. *Objekt* je instance třídy. Aby byla definována třída, musí být specifikována jedna nebo více přímých rodičovských tříd (*nadtříd*). Opakem nadtřídy je *podtřída* (dceřinná třída). Podtřída dědí atributy a chování od všech svých nadtříd (přímých i nepřímých). *Atributy* objektů třídy jsou popsány pomocí *rubrik* (*slotů*). *Chování* objektů třídy je definováno pomocí *obsluh zpráv*.

Hierarchii předdefinovaných tříd v CLIPSu ukazuje obr. 7.1. Je doporučeno, aby uživatelské třídy byly definovány jako podtřídy třídy USER.

Definice třídy:

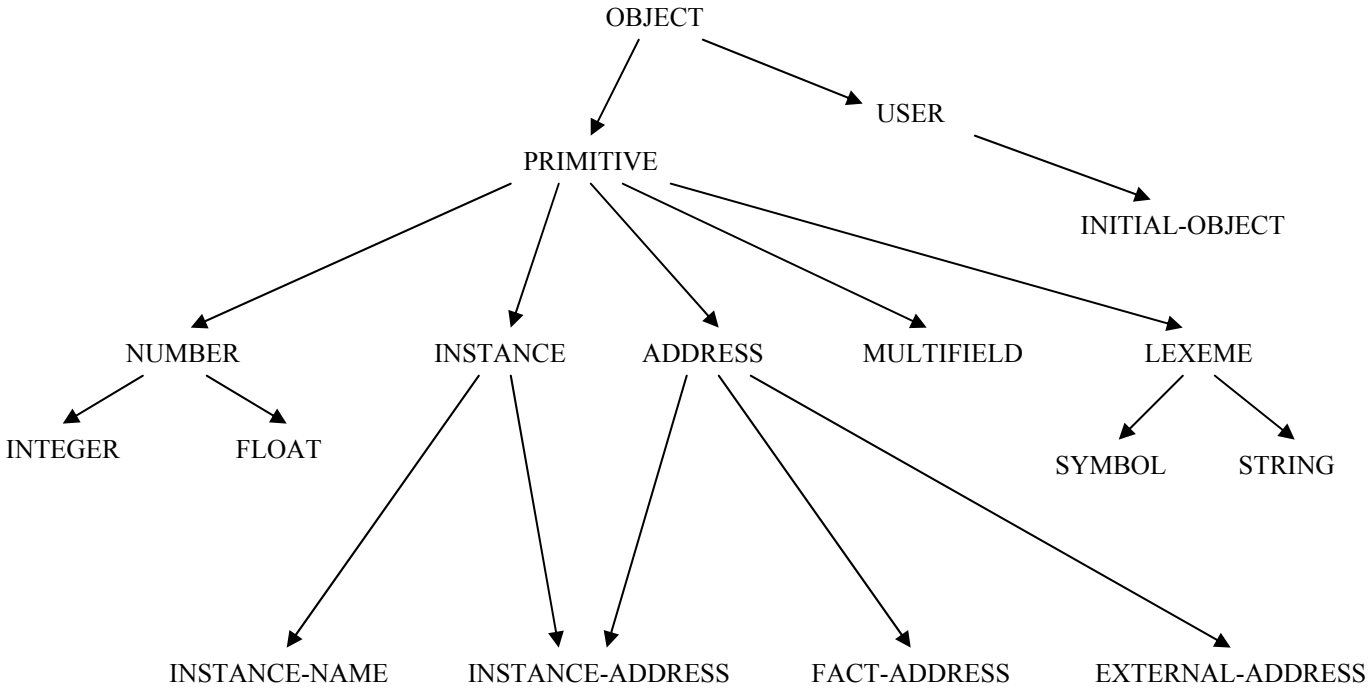
```
(defclass jméno_třídy "nepovinná poznámka"
      (is-a jméno_nadtřídy_1 … jméno_nadtřídy_m)
      (role typ_role)
      (pattern-match typ_role_při_porovnávání)
      definice_rubriky_1
      ...
      definice_rubriky_n
```

dokumentace_obsluhy_zpracování_zprávy_1

...

dokumentace_obsluhy_zpracování_zprávy_k)

Specifikace typů rolí jsou nepovinné a definice třídy nemusí obsahovat žádné rubriky ani dokumentaci obsluhy zpráv.



Obr 7.1. Hierarchie předdefinovaných tříd

Role třídy mohou být dvojího typu:

```
(role abstract)
(role concrete)
```

Pro abstraktní třídu nejdou vytvořit instance. Třída je implicitně abstraktní. Pro konkrétní třídu je možné vytvořit instance (objekty).

Role při porovnávání mohou být rovněž dvojího typu:

```
(pattern-match reactive)
(pattern-match non-reactive)
```

Objekt třídy může nebo nemusí reagovat na porovnání se vzorem. Implicitní je druhá možnost.

Instance třídy

Instance třídy jsou definovány pomocí konstruktu

```
(definstances jméno_skupiny_instancí
          "nepovinná poznámka"
          definice_instance_1
          ...
          definice_instance_n)
```

Definované instance jsou vytvořeny po provedení příkazu *reset*. Specifikovaná definice instancí může být zrušena pomocí konstruktu

```
(undefinstances jméno_skupiny_instancí)
```

ve spolupráci s příkazem *reset*.

Definice jednotlivých instancí se zapisují následovně:

```
(jméno_instance of jméno_třídy
          rubrika_1
          ...
          rubrika_n)
```

Jednotlivé rubriky jsou zadávány takto:

```
(jméno_rubriky hodnota_1 … hodnota_n)
```

V definici instance nemusí být uvedena žádná rubrika. Vytvořená instance obsahuje v nezadaných rubrikách implicitní hodnoty určené atributem *default*.

Jednotlivé instance mohou být vytvářeny a rušeny pomocí příkazů *make* a *unmake*:

```
(make-instance
          jméno_instance of jméno_třídy
          rubrika_1
          ...
          rubrika_n)

(unmake-instance jméno_instance)
```

Příkaz *unmake* ruší specifikovanou instanci zasláním zprávy *delete*. To můžeme zajistit také pomocí funkce *send* (viz § 7.7.4).

Objekty a porovnání se vzorem

Mezi vzory v levé straně pravidla se může objevit také vzor objektu:

```
(object vzor_1_atributu … vzor_n_atributu)
```

Vzory atributů mohou mít tyto tvary:

```
(is-a omezení)
(name omezení)
(jméno_rubriky omezení_1 … omezení_m)
```

7.7.2 Rubriky třídy

Definice rubrik třídy v definici třídy mohou mít následující podoby:

```
(slot jméno_rubriky fazeta_1 … fazeta_m)
(single-slot jméno_rubriky fazeta_1 … fazeta_m)
(multislot jméno_rubriky fazeta_1 … fazeta_m)
```

Fazety specifikují charakteristiky rubrik (způsob uložení hodnot, typ přístupu, způsob dědění, chování při porovnávání se vzorem, viditelnost, …).

Typy fazet:

- Fazeta *storage* určuje uložení hodnoty rubriky:

  ```
  (storage local)
  ```

  Hodnota je uložena v instanci (tato možnost je implicitní).

  ```
  (storage shared)
  ```

  Hodnota je uložena ve třídě a je sdílena objekty třídy..

- Fazeta *propagation* určuje způsob dědění:

  ```
  (propagation inherit)
  ```

  Je možné neomezené dědění (tento způsob je implicitní)

  ```
  (propagation no-inherit)
  ```

  Při tomto způsobu je možné pouze přímé dědění.

- Fazeta *source* určuje zdroj fazet při dědění:

  ```
  (source exclusive)
  ```

  Fazety se dědí pouze od nejbližšího předka (tato možnost je implicitní).

  ```
  (source composite
  ```

  Fazety, které nejsou explicitně specifikovány u nejbližšího předka, se hledají u dalšího nejbližšího předka.

- Fazeta *pattern-match* určuje chování rubriky při porovnávání se vzorem:

```
(pattern-match reactive)
(pattern-match non-reactive)
```

  Samotná rubrika tedy může nebo nemusí reagovat na porovnání se vzorem. Implicitní je druhá možnost.

- Fazeta *access* určuje typ přístupu k rubrice:

```
(access read-write)
(access read-only)
(access initialize-only)
```

  Implicitně je nastavena možnost čtení i zápisu.

- Fazeta *create-accessor* určuje, zda mají být vytvořeny obsluhy *put-* (vložení hodnoty) a/nebo *get-* (získání hodnoty); implicitně je tato fazeta určena nastavením fazety *access*.

```
(create-accessor ?NONE)
(create-accessor read)
(create-accessor write)
(create-accessor read-write)
```

- Fazeta *visibility* určuje viditelnost rubriky pro předky a potomky:

```
(visibility private)
(visibility public)
```

  U varianty *private* mají přímý přístup k rubrice pouze obsluhy definované třídy (tato možnost je implicitní).

- Fazeta *override-message* indikuje zprávu pro přepsání rubriky v případě, že uživatel nechce používat standardní zprávu *put-*.

```
(override-message ?DEFAULT)
(override-message <message-name>)
```

- Další typy fazet jsou stejné jako u šablon:
  - atribut *default* (slouží k nastavení počáteční hodnoty rubriky)
  - atributy omezení

7.7.3 Obsluha zpráv

Dokumentace obsluhy zpráv v definici třídy se může zadat takto:

```
(message-handler jméno_obsluhy)
(message-handler jméno_obsluhy typ_obsluhy)
```

Existují následující typy obsluhy: *primary*, *around*, *before*, *after*. Implicitní je typ *primary*.

Obsluha zpráv se definuje pomocí konstruktu:

```
(defmessage-handler jméno_třídy jméno_zprávy
      typ_obsluhy
      "nepovinná poznámka"
      (par_1 … par_n)  ;seznam parametrů
      akce_1
      ...
      akce_m)
```

Typ obsluhy nemusí být uveden (implicitní je *primary*). Seznam parametrů může být prázdný. Parametry se zapisují jako jednopolové proměnné, pouze poslední z nich může být vícepolová proměnná. Všechny typy obsluhy vztahující se k určité zprávě musejí mít stejný počet parametrů. Výsledná hodnota obsluhy je určena poslední akcí.

Typy obsluhy

- Obsluha *primary*

  Tato obluhaje určena pro realizaci hlavního úkolu. Vrací hodnotu. Z hierarchie *primary* obsluh třídy aktivního objektu a všech jejích předků se provede pouze ta nejspecifičtější obsluha (ta nejbližší aktivnímu objektu). *Primary* obsluha ovšem může volat *primary* obsluhu přímého předka pomocí funkcí *call-next-handler* a *override-next-handler*.

- Obsluha *before*

  Provede se před jakoukoli obsluhou typu *primary*. Nevrací hodnotu. Provedou se všechny *before* obsluhy objektu a jeho předků v pořadí od nejspecifičtější k nejobecnější.

- Obsluha *after*

  Provede se po všech obsluhách typu *primary*. Nevrací hodnotu. Provedou se všechny *after* obsluhy objektu a jeho předků v pořadí od nejobecnější k nejspecifičtější.

- Obsluha *around*

  Vytváří prostředí pro ostatní obsluhy (zastíněné obsluhy), které jsou volány pomocí funkcí *call-next-handler* a *override-next-handler*. Vrací hodnotu.

Funkce *call-next-handler* volá další zastíněnou obsluhu.

```
(call-next-handler)
```

Funkce *override-next-handler* volá další zastíněnou obsluhu a mění její argumenty.

```
(override-next-handler výraz_1 … výraz_n)
```

Funkce *next-handlerp* vrací hodnotu TRUE, jestliže existuje další proveditelná obsluha. V opačném případě vrací hodnotu FALSE.

```
(next-handlerp)
```

7.7.4 Posílání zpráv

Objekty (instance) mezi sebou komunikují pomocí posílání zpráv. Poslání zprávy zajišťuje funkce *send* kde místo výrazu [*jméno_instance*] může být proměnná.

```
(send [jméno_instance] jméno_zprávy)
(send [jméno_instance] jméno_zprávy arg_1 ... arg_n)
```

Jestliže obsluha zprávy není definována ve třídě objektu, CLIPS zkouší obsluhy předků této třídy. Z hlediska efektivnosti je dobré definovat obsluhy co nejblíže třídě, pro niž jsou určeny.

Funkci *send* můžeme využít i při práci s rubrikami:

```
(send [jméno_instance]
      put-jméno_rubriky výraz_1 … výraz_n)
```

Do rubriky specifikované instance se vloží hodnota ($n > 1$ v případě vícepolové rubriky).

```
(send [jméno_instance] get-jméno_rubriky)
```

Získá se hodnota rubriky specifikované instance.

Specifickým způsobem se pracuje s rubrikami aktivní instance. Aktivní instance je instance, které byla zaslána zpráva. Aktivní instance je uložena ve speciální proměnné *?self*. Tato proměnná se nesmí explicitně vyskytovat v argumentu obsluhy zprávy.

```
?self:jméno_rubriky
```

Tento výraz zpřístupňuje hodnotu rubriky. Vyhodnocuje se však staticky, takže dojde k selhání, pokud se obsluha nadtřídy takto pokusí získat hodnotu rubriky podtřídy.

```
(dynamic-put výraz_jméno_rubriky výraz_1 … výraz_n)
```

Tímto způsobem se vloží hodnota do specifikované rubriky aktivní instance ($n > 1$ v případě vícepolové rubriky).

```
(dynamic-get výraz_jméno_rubriky)
```

Takto se získá hodnota specifikované rubriky aktivní instance.

Pomocí funkce *send* můžeme také rušit instance:

```
(send [jméno_instance] delete)
```

Zruší se specifikovaná instance zasláním zprávy *delete*. Zrušení aktivní instance (tj. instance, které byla zaslána zpráva) můžeme zajistit také tak, že v těle obsluhy zprávy napíšeme

```
(delete-instance)
```

7.8 Příklady

Příklad 7.1

Příklad je jednoduchým programem pro řízení stavby věže z kostek. Na počátku se kostky nacházejí na hromadě. Při stavbě věže nesmí být položena větší kostka na menší. Program přepíná mezi dvěma úkoly:

- nalezení další volné kostky,
- položení kostky na věž.

```
; STAVBA VĚŽE
; ===========

; ŠABLONY

; Kostka je určena tak, že má barvu, velikost a polohu.
(deftemplate kostka
  (slot cislo (type INTEGER))
  (slot barva (type SYMBOL))
  (slot velikost (type INTEGER))
  (slot misto (type SYMBOL) (default hromada)))

; V případě, že místem je věž, pak vztah NA určuje pro dvě
; sousední kostky, která je nahoře a která dole.
(deftemplate na
  (slot nahore (type INTEGER))
  (slot dole (type INTEGER))
  (slot misto (type SYMBOL) (default hromada)))

; INICIALIZACE

; Máme 3 kostky různých barev a velikostí.
; Předpokládáme, že se nacházejí na hromadě.
(deffacts poc-stav
  (kostka (cislo 1) (barva cervena) (velikost 10))
  (kostka (cislo 2) (barva zluta) (velikost 20))
  (kostka (cislo 3) (barva modra) (velikost 30)))

; PRAVIDLA

;Nastavení prvého úkolu: najít kostku.
(defrule start
  (initial-fact)
  =>
  (assert (ukol najdi)))

; Zvednutí největší kostky z hromady.
(defrule zvedni
  ?ukol <- (ukol najdi)
  ?kostka <- (kostka (velikost ?v1) (misto hromada))
  (not (kostka (velikost ?v2&:(> ?v2 ?v1)) (misto hromada)))
  =>
  (modify ?kostka (misto ruka))
```

```
  (retract ?ukol)
  (assert (ukol stav)))

; Položení základní kostky věže.
(defrule poloz-zaklad
  ?ukol <- (ukol stav)
  ?kostka <- (kostka (misto ruka))
  (not (kostka (misto vez)))
  =>
  (modify ?kostka (misto vez))
  (retract ?ukol)
  (assert (ukol najdi)))

; Položení další kostky na nejvyšší kostku věže.
(defrule poloz-dalsi
  ?ukol <- (ukol stav)
  ?kostka <- (kostka (cislo ?c0) (misto ruka))
  (kostka (cislo ?c1) (misto vez))
  (not (na (nahore ?c2) (dole ?c1) (misto vez)))
  =>
  (modify ?kostka (misto vez))
  (assert (na (nahore ?c0) (dole ?c1) (misto vez)))
  (retract ?ukol)
  (assert (ukol najdi)))

; Zastavení v případě, že na hromadě se již nenachází
; žádná kostka.
(defrule stop
  ?ukol <- (ukol najdi)
  (not (kostka (misto hromada)))
  =>
  (retract ?ukol))
```

Příklad 7.2

Tento příklad představuje jednoduchý systém pro plánování akcí robota.

```
; ŘÍZENÍ ROBOTA
; =============

; ŠABLONY

; "Cíl" je vektor se 4 vlastnostmi:
;  - akce, která má být provedena,
;  - objekt, kterého se akce týká
;  - počáteční pozice
;  - koncová pozice
```

```
 (deftemplate Cil
  (slot akce (type SYMBOL))
  (slot objekt (type SYMBOL))
  (slot z (type SYMBOL))
  (slot do (type SYMBOL)))

; "Kde" zaznamenává pozici objektu
(deftemplate Kde
  (slot objekt (type SYMBOL))
  (slot pozice (type SYMBOL)))

; FAKTY

; Počáteční stav:
; - robot je v místě A,
; - bedna je v místě B,
; - cílem je přenést bednu do místa A
(deffacts PocStav
   (Kde (objekt Robot) (pozice MistoA))
   (Kde (objekt Bedna) (pozice MistoB))
   (Cil (akce Prenes) (objekt Bedna) (z MistoB) (do MistoA)))

; PRAVIDLA

; Jestli bylo dosaženo cíle, pak konec
(defrule Stop
  (Cil (objekt ?x) (do ?y))
  (Kde (objekt ?x) (pozice ?y))
  =>
  (halt))

; Jestli robot není na stejném místě, jako objekt, který
; má být přenesen, pak přesuň robota na pozici objektu
(defrule Presun-robota
  (Cil (objekt ?x) (z ?y))
  (Kde (objekt ?x) (pozice ?y))
  ?pozice-robota <- (Kde (objekt Robot) (pozice ?z&~?y))
  =>
  (modify ?pozice-robota (pozice ?y)))
```

```
; Jestliže je robot na správném místě, pak přesuň robota
; a objekt na cílové místo
(defrule Prenes
  (Cil (objekt ?x) (z ?y) (do ?z))
  ?pozice-objektu <- (Kde (objekt ?x) (pozice ?y))
  ?pozice-robota <- (Kde (objekt Robot) (pozice ?y))
  =>
  (modify ?pozice-robota (pozice ?z))
  (modify ?pozice-objektu (pozice ?z)))
```

Příklad 7.3

Příklad ilustruje práci s objekty. Jedná se o simulaci soutěže, kde soutěžícím jsou přidělovány náhodně generované body. V každém kole je vyřazen soutěžící s nejmenším počtem bodů. Soutěž končí, když zbývá jeden soutěžící.

```
; SOUTĚŽ
; ======

; TŘÍDY

 (defclass osoba
     (is-a USER))

(defclass soutezici
     (is-a osoba)
     (role concrete)
     (pattern-match reactive)
     (slot kolo (type INTEGER))
     (slot body (type INTEGER)))

; INSTANCE

; Definice skupiny soutěžících.
(definstances soutez
     (pepa of soutezici)
     (karel of soutezici)
     (robert of soutezici))

; OBSLUHY ZPRÁV

(defmessage-handler soutezici pridej-body (?zmena)
     (dynamic-put body (+ ?self:body ?zmena)))

(defmessage-handler soutezici zvys-kolo ()
     (dynamic-put kolo (+ ?self:kolo 1)))
```

```
; PRAVIDLA

; Výchozí kolo má číslo 1.
(defrule priprava
     (initial-fact)
     =>
     (assert (kolo 1)))

; Jestliže ještě existuje soutěžící, který neabsolvoval
; aktuální kolo, přidělí se mu body a zvýší se u něj číslo
; kola.
(defrule bodovani
     (kolo ?m)
     (object (name ?x) (kolo ?n&:(> ?m ?n)))
     =>
     (send ?x pridej-body (div (random) 1000))
     (send ?x zvys-kolo))

; Jestli už všichni soutěžící absolvovali aktuální kolo, určí
; se soutěžící s nejmenším počtem bodů a vyřadí se.
(defrule vyrazeni
     ?kolo <- (kolo ?m)
     (not (object (kolo ?n&:(> ?m ?n))))
     (object (name ?x) (body ?y))
     (not (object (body ?z&:(< ?z ?y))))
     =>
     (printout t ?m ". kolo: " ?x " ma padaka" crlf)
     (send ?x delete)
     (retract ?kolo)
     (assert (kolo (+ ?m 1))))

; Jestliže zbyl jediný soutěžící, je vítězem.
(defrule vitez
     (declare (salience 100))
     ?kolo <- (kolo ?)
     (object (is-a osoba) (name ?x))
     (not (object (is-a osoba) (name ?y&:(neq ?x ?y))))
     =>
     (printout t ?x " je vitez!!!" crlf)
     (retract ?kolo))
```

Příklad 7.4

Tento příklad ilustruje kombinace různých typů obsluhy zpráv.

```
; NÁZORY
; ======

; TŘÍDY

(defclass osoba
     (is-a USER)
     (slot jmeno (type SYMBOL) (create-accessor read-write)))

(defclass optimista
     (is-a osoba)
     (role concrete)
     (pattern-match reactive))

(defclass pesimista
     (is-a osoba)
     (role concrete)
     (pattern-match reactive))

(defclass optimisticky-pesimista
     (is-a pesimista)
     (role concrete)
     (pattern-match reactive))

(defclass pesimisticky-optimista
     (is-a optimista)
     (role concrete)
     (pattern-match reactive))

(defclass skeptik
     (is-a pesimisticky-optimista optimisticky-pesimista)
     (role concrete)
     (pattern-match reactive))

(defclass pijak
     (is-a osoba)
     (role concrete)
     (pattern-match reactive))

(defclass pijak-optimista
     (is-a pijak optimista)
     (role concrete)
     (pattern-match reactive))
```

```
(defclass pijak-pesimista
     (is-a pijak pesimista)
     (role concrete)
     (pattern-match reactive))

; OBSLUHY ZPRÁV

(defmessage-handler pijak rika before ()
     (printout t "lahev je "))

(defmessage-handler pijak rika after ()
     (printout t " a bude prazdna"))

(defmessage-handler pijak-pesimista rika primary ()
     (printout t "z poloviny prazdna"))

(defmessage-handler pijak-pesimista rika after ()
     (printout t " bohuzel uz velmi brzy" crlf))

(defmessage-handler pijak-optimista rika primary ()
     (printout t "z poloviny plna"))

(defmessage-handler pijak-optimista rika after ()
     (printout t " az za nejakou dobu" crlf))

(defmessage-handler pesimista rika ()
     (printout t " vsechno dopadne spatne" crlf))

(defmessage-handler optimista rika ()
     (printout t " vsechno dopadne dobre" crlf))

(defmessage-handler optimisticky-pesimista rika before ()
     (printout t "ne"))

 (defmessage-handler pesimisticky-optimista rika before ()
     (printout t "ne"))

(defmessage-handler skeptik rika before ()
     (printout t "ale "))

(defmessage-handler skeptik rika primary ()
     (printout t ", vsechno dopadne jinak, nez si
predstavujeme" crlf))
```

```
(defmessage-handler osoba rika  around ()
     (printout t crlf ?self:jmeno " rika: " crlf)
     (if (next-handlerp) then (call-next-handler)))

; INSTANCE

(definstances lide
     (tomas of skeptik
          (jmeno Tomas))
     (honza of pesimisticky-optimista
          (jmeno Honza))
     (karel of optimisticky-pesimista
          (jmeno Karel))
     (xaver of pesimista
          (jmeno Xaver))
     (radek of optimista
          (jmeno Radek))
     (josef of pijak-optimista
          (jmeno Josef))
     (jarda of pijak-pesimista
          (jmeno Jarda)))

; PRAVIDLA

(defrule nazor
     (object (is-a osoba) (name ?x))
     =>
     (send ?x rika))
```

Při použití uvedeného programu obdržíme následující výpis:

```
CLIPS> (reset)
CLIPS> (run)

Jarda rika:
lahev je z poloviny prazdna a bude prazdna bohuzel uz velmi brzy

Josef rika:
lahev je z poloviny plna a bude prazdna az za nejakou dobu

Radek rika:
vsechno dopadne dobre

Xaver rika:
vsechno dopadne spatne

Karel rika:
ne vsechno dopadne spatne
```

```
Honza rika:
ne vsechno dopadne dobre

Tomas rika:
ale nene, vsechno dopadne jinak, nez si predstavujeme

CLIPS>
```

Příklad 7.5

Tento příklad je ukázkou jednoduchého expertního systému, který diagnostikuje příčiny mokra v domě. Zkoumány jsou tři místnosti: hala, koupelna a kuchyně. Z haly vedou dveře do koupelny a do kuchyně. V kuchyni se nachází okno. Během dialogu s uživatelem systém klade dotazy a zaznamenává odpovědi uživatele do báze faktů. Na základě obsahu báze faktů a pravidel v bázi znalostí systém odvozuje nová fakta a zaznamenává je do báze faktů.

```
; Expertní systém diagnostikující příčiny mokra v domě
; ====================================================

; FUNKCE

 (deffunction dialog (?dotaz)
     (printout t ?dotaz)
     (bind ?odpoved (read))
     ?odpoved)

; PRAVIDLA (BÁZE ZNALOSTÍ)

(defrule R0
     (declare (salience 10))
     (initial-fact)
     =>
     (bind ?x (dialog "hala je (mokra/sucha)?"))
     (assert (hala ?x))
     (bind ?x (dialog "koupelna je (mokra/sucha)?"))
     (assert (koupelna ?x))
     (bind ?x (dialog "kuchyne je (mokra/sucha)?"))
     (assert (kuchyne ?x)))

(defrule R1
     (hala sucha)
     (koupelna mokra)
     =>
     (assert (unik-vody koupelna)))
```

```
(defrule R2
     (hala mokra)
     (kuchyne sucha)
     (koupelna mokra)
     =>
     (assert (unik-vody koupelna)))

(defrule R3
     (kuchyne mokra)
     =>
     (bind ?x (dialog "okno je (otevrene/zavrene)?"))
     (assert (okno ?x)))

(defrule R4
     (okno otevrene)
     =>
     (bind ?x (dialog "prsi (ano/ne)?"))
     (assert (dest ?x)))

(defrule R5
     (okno otevrene)
     (dest ano)
     =>
     (assert (voda-zvenku ano)))

(defrule R6
     (or (okno zavrene)
         (dest ne))
     =>
     (assert (voda-zvenku ne)))

(defrule R7
     (kuchyne mokra)
     (voda-zvenku ano)
     =>
     (printout t "Zavri okno v kuchyni, vysus podlahu a cekej"
      crlf))

(defrule R8
     (kuchyne mokra)
     (voda-zvenku ne)
     (not (and (hala mokra) (koupelna mokra)))
     =>
     (assert (unik-vody kuchyne)))
```

```
(defrule R9
     (koupelna mokra)
     (hala mokra)
     (kuchyne mokra)
     (voda-zvenku ne)
     =>
     (assert (unik-vody koupelna-nebo-kuchyne)))

(defrule R10
     (declare (salience -10))
     (unik-vody ?kde)
     =>
     (printout t "Misto uniku vody: " ?kde crlf))

(defrule R11
     (declare (salience -10))
     (hala mokra)
     (koupelna sucha)
     (kuchyne sucha)
     =>
     (bind ?x (dialog "prsi (ano/ne)?"))
     (if (eq ?x ano)
          then (printout t "Mozna zateka stropem nebo dvermi.
                Jinak nevim." crlf)
          else (printout t "Opravdu, ale opravdu nevim, kde se tu
                ta voda mohla vzit." crlf)))

(defrule R12
     (declare (salience -10))
     (hala sucha)
     (koupelna sucha)
     (kuchyne sucha)
     =>
     (printout t "Dej si panaka, at to tu neni jako u Suchanku."
      crlf))
```

Literatura

Berka, P. *Dobývání znalostí z databází*. Praha, Academia 2003.

Berka, P. a kol. *Expertní systémy*. Skripta. Praha, VŠE 1998.

Berka, P. *Tvorba znalostních systémů*. Skripta. Praha, VŠE 1994.

Berka, P. *Vybrané znalostní systémy SAK, SAZE, KEX*. Skripta. Praha, VŠE 1994.

Csontó, J, Sabol, T. *Umelá inteligencia*. Skripta. Košice, TU 1991.

Druckmüller, M. *Technické aplikace vícehodnotové logiky*. Skripta. Brno, PC-DIR Real 1998.

Druckmüller, M. *Linguistic Model Processing System. Průvodce systémem LMPS*. Brno, Janes 1991.

Dvořák, J., Šeda, M. Comparison of Fuzzy Similarity Measures. *In Proceedings of the 3rd International Conference Aplimat*, Slovak University of Technology in Bratislava, 2004, pp. 387-392.

Dvořák, J., Hodál, J. Case-Based Reasoning Approaches to Robot Navigation. In *Proceedings of the 9th International Conference on Soft Computing MENDEL 2003*, Brno, Czech Republic, 2003, pp.326-333.

Dvořák, J., Hodál, J. Similarity Measures in Case-Based Reasoning. In *Proceedings of the 9th Zittau Fuzzy Colloquium 2001*, Zittau, Germany, 2001, pp. 21-28.

Dvořák, J., Krček, P., Samohýl, P. Using Case-Based Reasoning and Genetic Algorithms for Mobile Robot Path Planning. In *Proceedings of XXVIth International Autumn Colloquium Advanced Simulation of Systems ASIS 2004*, Sv. Hostýn, 2004, pp. 185-190.

Expertní systémy pro podporu konstrukčních prací ve strojírenství. Sborník. Praha DT ČSVTS 1988.

Geyer-Schulz A. *Fuzy Rule-Based Expert Systems and Genetic Machine Learning*. Heidelberg, Physica-Verlag 1995.

Giarratano, J., Riley, G. *Expert Systems. Principles and Programming*. Boston, PWS Publishing Company 1998.

Gonzalez, A. J., Dankel, D. D.: *The Engineering of Knowledge-based Systems. Theory and Practice*. New York, Prentice Hall 1993.

Gosman, S. a kol. Umělá inteligence a expertní systémy. *Výběr informací z organizační a výpočetní techniky*. Mimořádné číslo. Kancelářské stroje 1990.

Hand, D., Mannila, H. Smyth, P. *Principles of Data Mining*. Cambridge, MIT Press 2001.

Jackson, P. *Introduction to Expert Systems*. Harlow, Addison-Wesley 1999.

Kelemen J. a kol.: *Tvorba expertních systémů v prostředí CLIPS*. Praha, Grada 1999.

Krček, P. *Využití expertního systému pro plánování dráhy robota*. Diplomová práce. Brno, FSI VUT 2003.

Krček, P., Dvořák, J.: Mobile Robot Motion Control by Means of Fuzzy Rules (in Czech). In *Book of Extended Abstracts of the National Conference with International Participation*

Engineering Mechanics 2004. Association for Engineering Mechanics, Svratka, 2004, pp. 157-158 + 10 pp. on CD-ROM.

Lenz, M., Bartsch-Spörl, B., Burkhard, H.-D., Wess, S. (Eds.): *Case-Based Reasoning Technology: From Foundations to Applications.* Berlin, Springer-Verlag 1998.

Liebowitz, J., De Salvo, D.A. *Structuring Expert Systems. Domain, Design and Development*. Englewood Cliffs, Prentice Hall 1989.

Luger, G.F. *Artificial Intelligence. Structures and Strategies for Complex Problems Solv*ing. Harlow, Addison-Wesley 2002.

Mařík, V. a kol.: *Umělá inteligence* 1. Praha, Academia 1993.

Mařík, V. a kol.: *Umělá inteligence* 2. Praha, Academia 1997.

Mital, A., Anand, S. (Eds.) *Handbook of Expert Systems Applications in Manufacturing. Structures and Rules*. London, Chapman & Hall 1994.

Mitchell, T.M. *Machine Learning*. New York, McGraw Hill 1997.

Nikolopoulos, C. *Expert Systems*. *Introduction to First and Second Generation and Hybrid Knowledge Based Systems*. New York. Marcel Dekker 1997.

Novák, M. *Fuzzy množiny a jejich aplikace*. Praha, SNTL 1986.

Payne, E.C., Arthur, R.C. *Developing Expert Systems. A Knowledge Engineer's Handbook for Rules & Objects*. John Wiley & Sons 1990.

Pedersen, K. *Expert Systems Programminng. Practical Techniques for Rule-Based Systems*. New York, John Wiley & Sons 1989.

Pokorný, M. *Řídicí systémy se znalostní bází*. Skripta. Ostrava, VŠB – TU 1999.

Popper, K., Kelemen, J. *Expertné systémy*. Bratislava, Alfa 1988.

Vysoký, P. *Fuzzy řízení*. Skripta. Praha ČVUT 1996.

Waterman, D.A. *A Guide to Expert Systems*. Reading, Addison-Wesley 1986.

Winstanley, G. *Program Design for Knowledge Based Systems*. Wilmslow, Sigma Press 1987.

Zbořil, F., Hanáček, P. *Umělá inteligence*. Skripta. Brno, FE VUT 1991.