VŠB - Technická Univerzita Ostrava, Fakulta bezpečnostního inženýrství

Expertní systémy

Pavel Šenovský

Ostrava 2007

Obsah

Úvod 5
1 Umělá inteligence 7
2 Modelování znalostí 10
- 2.1 Historie jazyka UML 10
- 2.2 Diagramy jazyka UML 11
- 2.3 Třídní diagram 16
- 2.4 Model jednání 18
- 2.5 Stavový diagram 20
- 2.6 Scénáře činností 20
- 2.7 Diagram spolupráce 21
- 2.8 Diagram činností 22
- 2.9 Diagram komponent 23
- 2.10 Diagram nasazení 24
- 2.11 UML závěr 24

3 Expertní systémy 26
- 3.1 Stručná historie expertních systémů 26
- 3.2 Základy expertních systémů 27
- 3.3 Definice ontologie pomocí Protege 2000 29
- 3.4 Šablony CLIPS 34
- 3.5 Definice fakt 35
- 3.6 Pravidla 36
- 3.7 Závěr 39

4 Neuronové sítě 40
- 4.1 Biologické základy neuronových sítí 40
- 4.2 Perceptron 41
- 4.3 Pracovní prostředí SciLab 45
- 4.4 Poznámky k modelování pomocí neuronových sítí 49
- 4.6 Problémy řešitelné neuronovými sítěmi 50

5 Buněčné automaty 52
- 5.1 Počátky buněčných automatů – Conwayova hra života 52
- 5.2 Wolframův 1D buněčný automat 54
- 5.3 Boidi – Raynoldsův model shlukování ptáků 57
- 5.4 Lindenmayerovy systémy 57
- 5.5 Programovací jazyk Logo 60

6 Multiagentní systémy 64
- 6.1 Inteligentní agenti 64
- 6.2 Multiagentní systémy 68

7 Genetické algoritmy 71
Literatura 75

Úvod

Vážený studente, dostává se Vám do rukou učební text předmětu *Expertní systémy.* Základním cílem těchto skript a potažmo i předmětu je, přiblížit různé metody pro podporu rozhodování zaměřené do oblasti umělé inteligence nebo chcete-li obecněji – umělého života.

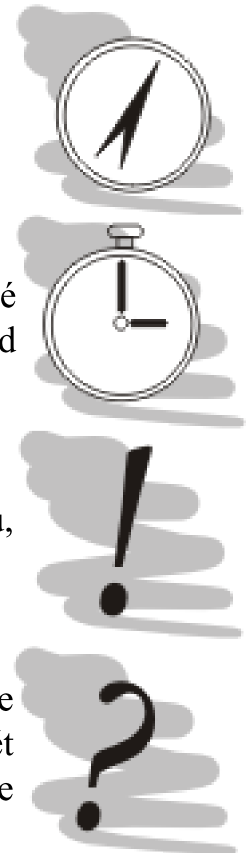
Pro zpříjemnění čtení jsem se také rozhodl, zpracovat tento text formou vhodnou pro „distanční vzdělávání“, tak aby práce s ním byla co možná nejjednodušší. Z tohoto důvodu je text jednotlivých kapitol segmentován do bloků.

Každá kapitola začíná náhledem kapitoly, ve kterém se dozvíte o čem budeme v kapitole mluvit a proč. V bodech se pokusím shrnout co byste po prostudování kapitoly měli znát a kolik času by Vám studium mělo zabrat. Prosím mějte na paměti, že tento časový údaj je pouze orientační, nebuďte proto prosím smutní nebo naštvaní když ve skutečnosti budete kapitole věnovat o něco méně nebo více času.

Za kapitolou následuje shrnutí, ve kterém budou zdůrazněny informace, které byste si rozhodně měli zapamatovat.

To, že jste správně pochopili probíranou látku si budete moci ověřit pomocí kontrolních otázek, které by Vám měly poskytnout dostatečnou zpětnou vazbu k rozhodnutí zdali jít dále nebo věnovat delší čas opakování.

Pro zjednodušení orientace v textu jsem zavedl systém ikon:



Průvodce studiem
Slouží pro seznámení studentů s látkou která bude v kapitole probírána.

Čas nutný ke studiu
Představuje odhad doby, který budete potřebovat ke prostudování celé kapitoly. Jedná pouze o orientační odhad, neznepokojujte se proto pokud Vám studium bude trvat o něco déle nebo budete hotovi rychleji.

Vysvětlení, definice, poznámka
U této ikony najdete vysvětlující text, poznámku k probíranému tématu, která problém uvede do širších souvislostí, popřípadě důležitou definice.

Kontrolní otázky
Na závěr každé kapitoly je zařazeno několik otázek, které prověří, zda jste problematice kapitoly dostatečně porozuměli. Pokud nebudete vědět odpověď na některou otázku, je to signál pro Vás, abyste se ke kapitole vrátili.

A nakonec si to ověříme pomocí několika kontrolních otázek.

Přeji Vám, aby jste čas který strávíte s tímto textem byl co možná nejpříjemnější a abyste jej nepovažovali za ztracený.

Autor.

1 Umělá inteligence

Průvodce studiem

V této kapitole se zamyslíme nad tím, proč se vůbec zabývat umělou inteligencí, jak ji chápat. Vytvoříme si určitý rámec, do kterého postupně budeme přidávat vědomosti z různých oblastní umělé inteligence.

Po prostudování této kapitoly budete vědět

- jakým způsobem dělíme umělou inteligenci
- důvody seřazení jednotlivých kapitol
- vzájemnou provázanost kapitol

Čas pro studium

Pro prostudování této kapitoly budete potřebovat přibližně 45 minut.

Umělou inteligenci lze definovat jako soubor nástrojů, které mají společný cíl pokud možno věrně simulovat metody myšlení živých organizmů. Tímto způsobem se snažíme zajistit, aby počítačové vybavení vykazovalo jisté aspekty inteligentního chování v předmětné oblasti. Umělou inteligenci jako obor lze rozdělit podle řady kritérií, pro náš účel bude postačovat rozdělení podle aspektu inteligence, který se snaží zachytit.

Z takového pohledi můžeme rozdělit umělou inteligenci do následujících oblastí:

- formulace řečového aktu
- expertní systémy
- neuronové sítě

Problém formulace řečového aktu spočívá ve snaze zajistit, aby počítač byl schopen komunikovat běžným jazykem, podobným způsobem jako člověk. Inteligenci člověka posuzujeme obvykle právě podle způsobu jeho vyjadřování. Blíže se s touto problematikou seznámíme v kapitole 3 (systémy ELISA, A.L.I.C.E.).

Expertní systémy si kladou za cíl nahradit ve své problémové doméně člověka – experta. Základním problémem experta totiž je, že se jedná o člověka a člověk jak známo je tvor omylný, potřebuje jíst a spát a obvykle také nečeká v pohotovosti, aby hbitě pomohl identifikovat/odstranit problém. Expertní systémy samozřejmě tyto problémy nemají, ale na druhou stranu nejsou schopny úplně nahradit experta – člověka. Jejich hlavní a reálně dosažitelným úkolem, je automatizovat některé činnosti spojené s řešením problémové situace a poskytnout určité rady, které budou přínosné pro řešení problému (ale nemusí být pro danou situaci optimální) do příchodu skutečného, lidského experta.

Prostředek, který k tomuto účelu expertní systémy využívají je formální logika, pomocí které se zpracovávají informace o problému (fakta) a způsobu jeho řešení (pravidla). Obě dohromady tvoří tzv. bázi znalostí, která tvoří jádro každého expertního systému.

Abychom mohli takovou bázi znalostí sestavit je nutné ji nejprve navrhnout. K tomuto účelu budeme využívat modelovací jazyk UML. Protože fáze modelování obvykle předchází fázi tvorby báze dat, budeme se věnovat jazyku UML již v kapitole 2 a teprve poté se zaměříme na samotné expertní systémy obsažené v kapitole 3.

Neuronové sítě přistupují k řešení problémů jiným způsobem – snaží se napodobit fyzické aspekty chování mozku. Člověk obvykle nepracuje stejným způsobem jako počítač, své dovednosti se postupně učí. Učení se proto stává základním nástrojem také umělých neuronových sítí. Obvykle postupujeme tak že kontrolujeme výstup neuronové sítě a porovnáváme jej s žádoucím (správným) výsledkem. Neuronová síť se postupně adaptuje tak, aby se její výsledky blížily žádoucím hodnotám.

Po provedení adaptace (učení) neuronové sítě, je tato síť připravena poskytovat údaje. Informace o neuronových sítích, adaptaci a použití naleznete v kapitole 4.

Kromě umělé inteligence jako takové je předmětem zájmu v tomto předmětu také tzv. umělý život, tento obvykle nemá charakter umělé inteligence neboť se nesnaží napodobit způsob myšlení člověka, ale přesto mohou poskytnout zajímavé výsledky. Do oblasti umělého života řadíme:

- buněčné automaty
- lindenmayerovy systémy
- agentní/multiagentní systémy
- genetické algoritmy

Buněčné automaty jsou dynamickými modely, diskrétními v čase. Simulace probíhá v iteracích, kdy na všechny buňky univerza působí zároveň stejná transformační funkce (pravidlo). Aplikační transformační funkce dochází ke změně modelu, kterou nějakým způsobem interpretujeme. Prostřednictvím buněčných automatů je možné simulovat rozvoj lesního požáru, ale třeba také průběh vzniku kolony v dopravě.

Lindenmayerovy systémy slouží pro vykreslení tvarů, které nápadně připomínají rostliny nebo jejich části. Vykreslování probíhá podobným způsobem jako u buněčných automatů, tedy opakovanou aplikací jednoduchého transformačního pravidla.

Buněčným automatům i lindenmayerovým systémům se budeme věnovat v kapitole 4.

Agentní/multiagentní systémy složí pro napodobení inteligentního chování. Pro toto napodobení je využit trochu jiný přístup – je realizován pomocí jednoúčelových nástrojů věrně zachycujících určitý aspekt

chování. Těmto nástrojům říkáme agenti. Vzájemnou komunikací agentů dojde k emergenci inteligentního chování (resp. jeho napodobení).

Samotní agenti mohou být založeni na expertních systémech, neuronových sítích nebo nějakém jiném typu modelu.

Agenty se budeme zabývat v kapitole 5.

Poslední, šestou kapitolu věnujeme genetickým algoritmům. Tyto algoritmy se snaží pro své fungování napodobit ne chování jednotlivce nebo jeho části (např. mozku), ale celého biologického druhu. Genetické algoritmy vycházejí z evoluční teorie, která hovoří o přirozeném výběru jako mechanismu pro optimalizaci schopnosti přežití biologického druhu jako celku.

Genetické algoritmy se snaží tento mechanismus napodobit a vyřešit tak zadané především optimalizační problémy.

Jednotlivé metody, se kterými se v těchto skriptech postupně seznámíte je možné použít samostatně nebo v tandemu s dalšími metodami, se kterými jste se mohli setkat v předmětech jako je statistika nebo modelování rozhodovacích procesů.

Kontrolní otázky

1) Jaký je rozdíl mezi umělou inteligencí a umělým životem?
2) Co řadíme do oblasti umělé inteligence?
3) Co řadíme do oblasti umělého života?

2 Modelování znalostí

Průvodce studiem

V této kapitole se zaměříme na možnosti modelování báze znalostí, jako podklad pro implementaci expertního systému v dané oblasti. Při výkladu se zaměříme především na modelovací jazyk UML (Universal Modeling Language).

Po prostudování této kapitoly budete umět

- modelovat bázi znalostí
- konstruovat diagramy v jazyce UML

Čas pro studium

Pro prostudování této kapitoly budete potřebovat minimálně dvě hodiny.

2.1 Historie jazyka UML

objektově orientované programování

UML ve své podstatě vychází z konceptu *objektově orientovaného programování*. Pro tento typ programování jsou totiž nevhodné metodologie zaměřené na modelování funkční a datové. V devadesátých letech se proto vyvíjí celá řada nových metodologií zaměřených výhradně objektově.

Prominentní roli v tomto hraje firma Rational Corporation, jedna z prominentních firem zabývajících se vývojem vývojářů CASE systémů (Computer Aided System Engeneering). V roce 1994 zaměstnává Jamese Rumbaughta, který propaguje svůj přístup k modelování nazvaný *Rumbaughtova technika modelování objektů* (používá se zkratka *OMT*). Rumbaught svou techniku vyvinul v roce 1991. OMT se zaměřuje na objektově orientovanou analýzu.

Gary Booch (opět zaměstnanec Rational) vyvíjí jinou techniku (*Boochova metoda*), která vyniká v objektově orientovaném návrhu.

Boochova metoda i OMT mají své silné stránky, ale také slabiny, z tohoto důvodu oba začínají pracovat na sjednocení svých metod, tak aby vynikly silné stránky obou přístupů. V roce 1995 se k nim připojuje taktéž Ivar Jacobson (autor metody OOSE).

UML

První návrh sjednocených metod vzniká již v roce 1996 pod názvem UML (Unified Modeling Language) a v roce 1997 (UML v1.1) je přijata jako standard organizací Object Management Group (OMG). V roce 2004 vychází významná revize jazyka UML v2.0.

V roce 2005 vychází UML také jako norma ISO 19501 [2], což podtrhuje dominantní postavení jazyka UML v modelování.

V současné době se připravuje revize 2.1 jazyka UML.

2.2 Diagramy jazyka UML

Diagramy UML

UML je grafickým jazykem, který pro popis modelovaného problému používá soustavu grafických symbolů jednoduše interpretovatelných i neodborníkem v oblasti návrhu systémů.

UML podporuje následující základní typy digramů:

- třídní diagram (class diagram)
- model jednání (use case diagram, diagram případů užití)
- stavový diagram (state diagram)
- scénáře činností (sequence diagram, sekvenční diagram)
- diagram spolupráce (collaboration diagram)
- diagram činností (activity diagram)
- diagramy komponent (component diagram)
- diagram nasazení (deployment diagram)

V následujících podkapitolách se budeme věnovat konstrukci jednotlivých diagramů. Předtím, než se však do toho pustíme podívejme se společně na možnosti podpory konstrukce diagramů UML pomocí softwarových nástrojů.

Jelikož UML je pravděpodobně nejrozšířenějším modelovacím jazykem, existují desítky, možná stovky nástrojů, které bychom mohli použít. Nástroje bychom obecně mohli rozdělit do dvou skupin:

- obecné kreslící nástroje se zabudovanou podporou symboliky UML
- specializované CASE nástroje

Obecné kreslící nástroje nám mohou pomoci při kreslení diagramů, kromě toho neobsahují, žádnou další funkci. Umožňují tedy vytvoření, editaci, tisk diagramů a jejich export do podoby běžných bitmap.

CASE nástroje oproti tomu nabízejí ještě další funkčnost, jako je generování zdrojového kódu na základě diagramů, reverzní analýza kódu s vytvořením příslušných diagramů, nástroje pro generování dokumentace a další.

Vzhledem k tomu, že my, alespoň v tomto předmětu, nebudeme UML používat jako podklad pro programování, našim požadavkům vyhoví oba druhy nástrojů. Zaměřme se nejprve do oblasti grafických nástrojů. Do této skupiny můžeme zařadit nástroje jako Visio nebo Dia.

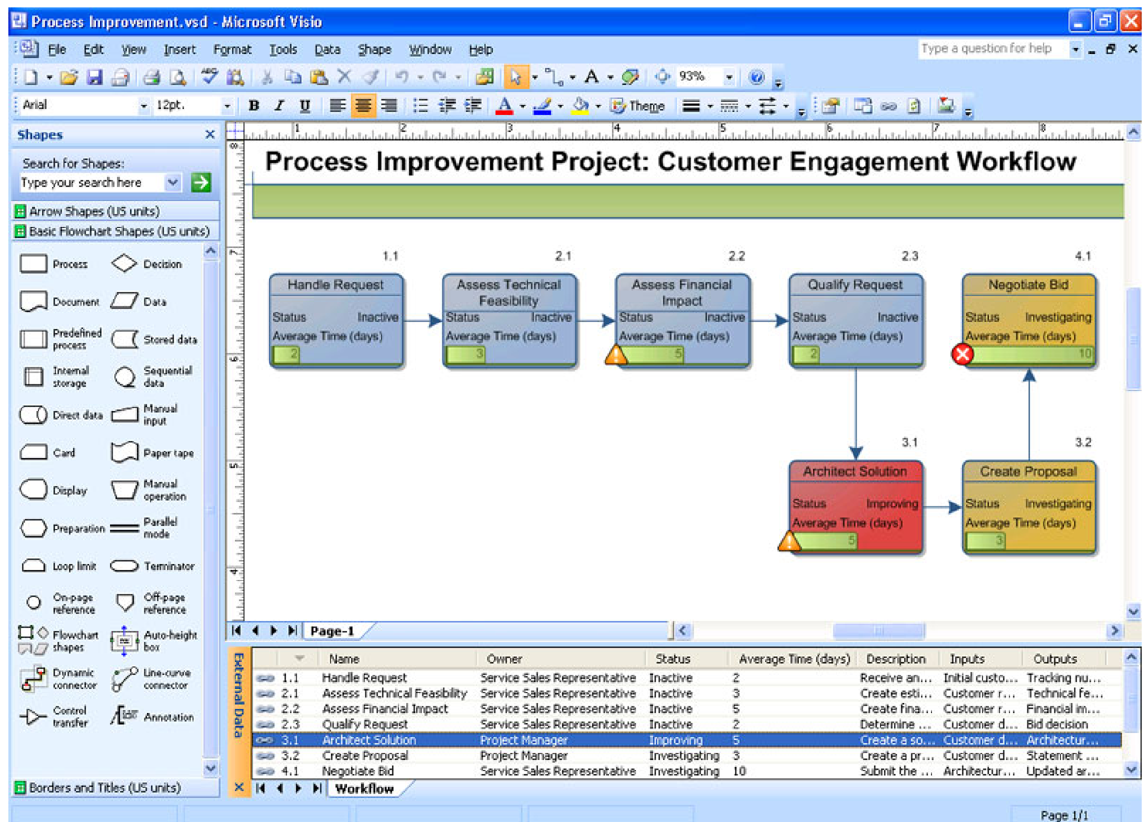
Visio

Visio

Visio je obecný nástroj z dílny Microsoftu pro kreslení schémat různého druhu (algoritmy, schémata sítí, organizační struktury apod.). Tento produkt se prodává ve verzích Standard a Profesional, které se liší výběrem dostupných diagramů. Podpora UML je zavedena až ve verzi Profesional.

V době psaní těchto skript (červenec 2007) byla verze Visio Profesional na FBI nainstalována na počítačové učebně LA49. Na učebně LA56 je sice Visio také instalováno, ale pouze ve verzi Standard, tedy bez podpory UML.

Náhled pracovního okna naleznete na obr. 1.



Obr. 1: Pracovní okno programu Visio [zdroj: microsoft.com]

Svou funkčností se Visio řadí do rodiny programů Microsoft Office (Podobně jako třeba MS Project), jeho ovládání je proto velmi intuitivní.

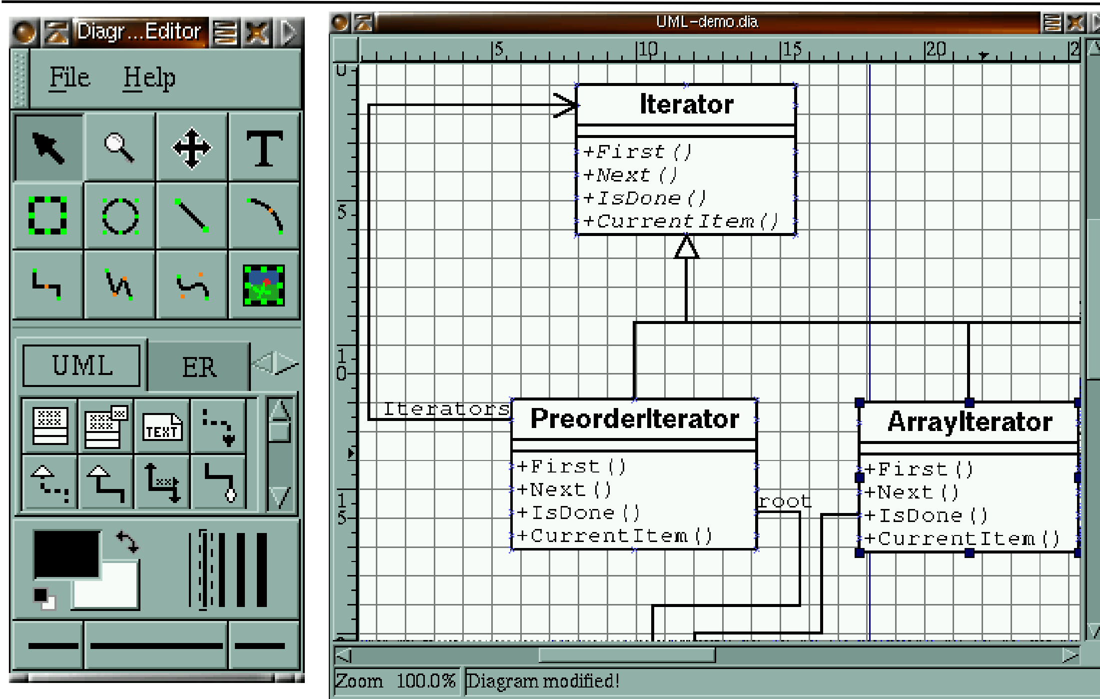
Dia

Dia

Program Dia [3] je zástupcem open-source v kategorii obchodní grafiky. Tento produkt je dostupný zdarma pro všechny majoritní operační systémy včetně Windows a různých variant Linuxu. Svou koncepcí je tento program trochu odlišný. Panel nástrojů totiž s pracovní plochou programu netvoří jeden celek, ale fungují jako samostatná okna. To je vlastnost, kterou pravděpodobně více ocení uživatelé Unixových operačních systémů, které nativně podporují práci s více pracovními plochami, což je vlastnost, která v operačním systému Windows v použitelné podobě chybí.

Pro provoz programu Dia na Windows je nutno nejprve nainstalovat aktuální knihovnu GTK, kterou program Dia vyžaduje pro svou činnost. GUI programu Dia si můžete prohlédnout na obr. 2.

V oblasti CASE nástrojů se nabízí celá řada možností, vynechám přitom čistě komerční řešení protože nepředpokládám, že bych svým



Obr. 2: Dia [zdroj [3]]

výkladem inspiroval někoho ke koupi. Naštěstí je v oblasti freeware a open-source dostatek nástrojů, které nám mohou pomoci aniž by se to projevilo na stavu našich financí.

StarUML

StarUML

Prvním z nástrojů, které bych z této skupiny zmínil je StarUML [4]. Jedná se o vysoce kvalitní open-source CASE nástroj, který kromě UML ve verzi 2.0 obsahuje i nástroje pro generování kódu, konstrukci diagramů na základě automatizované analýzy zdrojového kódu (v jazycích JAVA a C#).

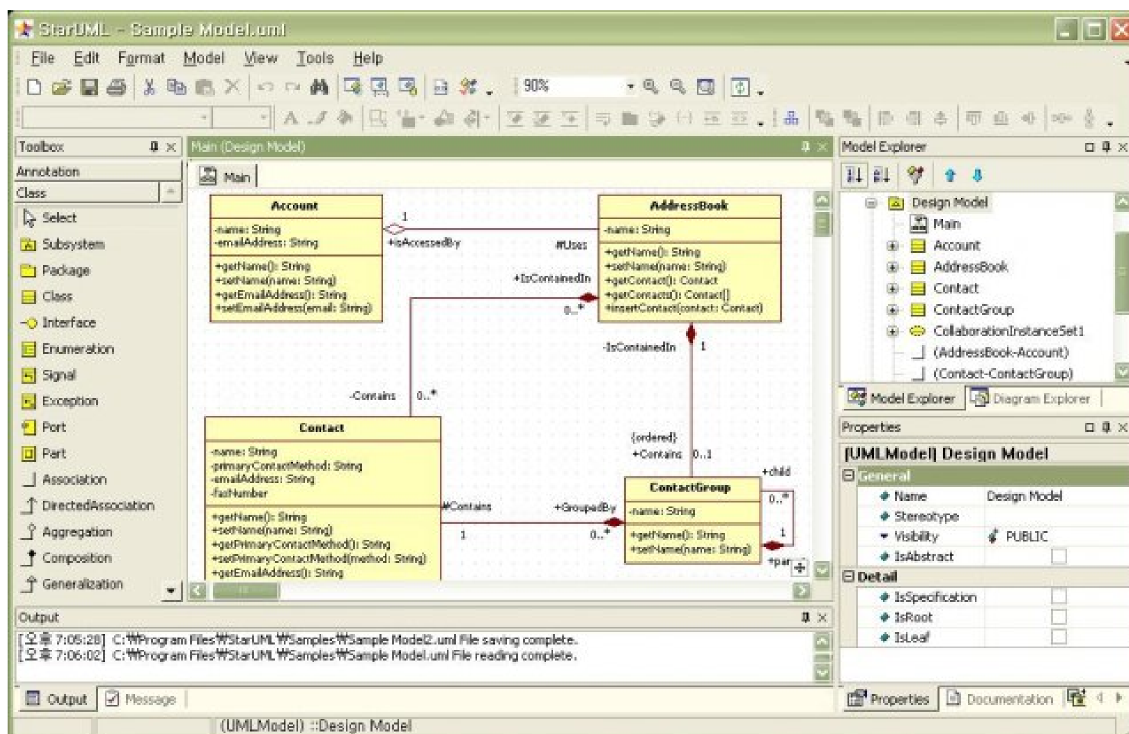
Kromě toho obsahuje celou řadu dalších možností použitelných s využitím plug-in architektury programu jako je možnost konstrukce ERD diagramů (není součástí specifikace UML), podpora AML (Agent Modeling Language) a další.

Náhled pracovního prostředí je viditelný na obr. 3.

Rozhraní produktu je ve srovnání s kreslícími nástroji s podporou UML složitější. V CASE nástrojích jako je StartUML se nepostupuje systémem jeden soubor – jeden diagram, ale jeden soubor – jeden projekt. Projekt přitom v sobě může obsahovat libovolné množství digramů a jednotlivé objekty v diagramech zpracované jsou pak jednoduše využitelné v dalších diagramech.

Kromě toho jednotlivých prvkům diagramu definujeme vlastnosti. To je nutný krok v okamžiku, kdy bychom požadovali generování kódu. V našem případě však tyto pokročilé možnosti nevyužijeme.

StarUML je možné provozovat operačních systémech MS Windows.



Obr. 3: Star UML [zdroj [4]]

ArgoUML

ArgoUML

Dalším představitelem CASE nástrojů je ArgoUML. Jedná se o open-source naprogramovaný v jazyce JAVA, což znamená, že se jedná o produkt multiplatformní a že ke svému běhu program vyžaduje přítomnost JAVA JRE nebo JAVA SDK ve verzi 1.4 nebo novější (v době psaní skript byla aktuální JAVA 6.0, aktuální verzi běhového prostředí JAVA lze stáhnout ze stránek www.sun.com).

ArgoUML obsahuje podporu pro jazyk UML v1.4, ale jinak je funkčně z hlediska možnosti práce s diagramy je srovnatelný se StarUML. Poměrně výraznou nevýhodou ArgoUML je vysoká hardwarová náročnost (zejména nároky na paměť) při práci s větším množstvím objektů a určitá nestabilita.

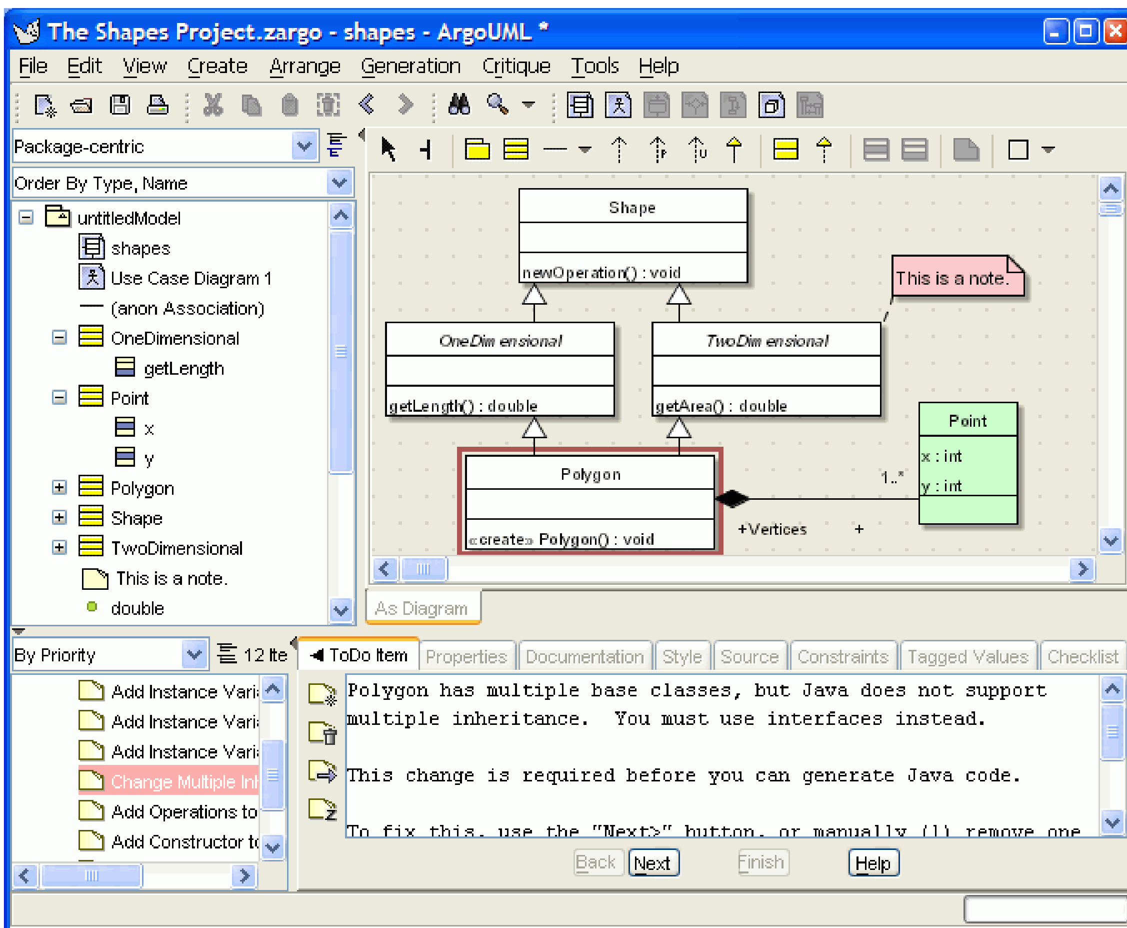
Visual Paradigm

Visual Paradigm

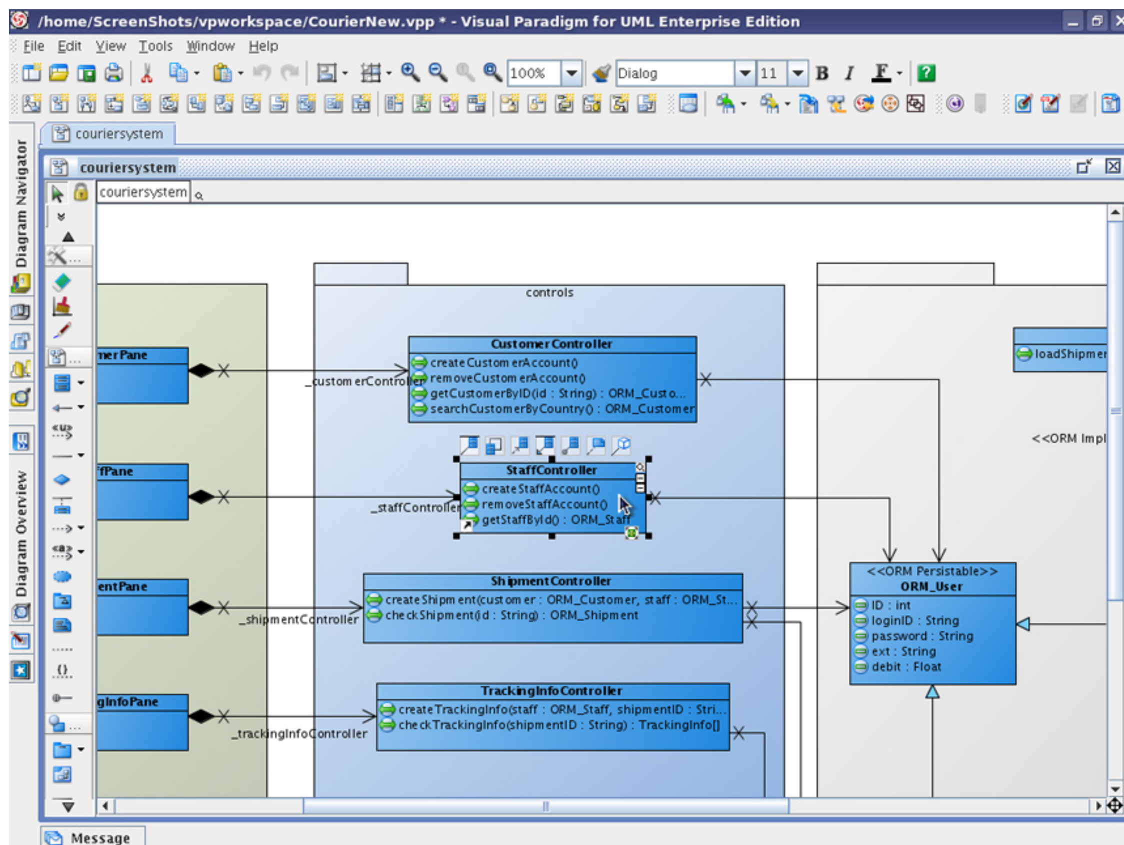
Posledním představitelem CASE nástrojů, o kterém bych se rád zmínil je produkt Visual Paradigm for UML Community Edition [6]. Visual Paradigm je komerční CASE, jehož Community Edition je však šířena bezúplatně. Tato bezplatná verze je funkčně omezena tak, že jeden projekt může obsahovat pouze jeden diagram z každého z podporovaných typů diagramů.

Visual Paradigm je jinak funkčně na velmi vysoké úrovni. Podporuje UML v2.1, umožňuje integraci do vývojových prostředí, modelování databází apod.

Příklad uživatelského rozhraní je na obr. 6.



Obr. 5: ArgoUML [zdroj [5]]



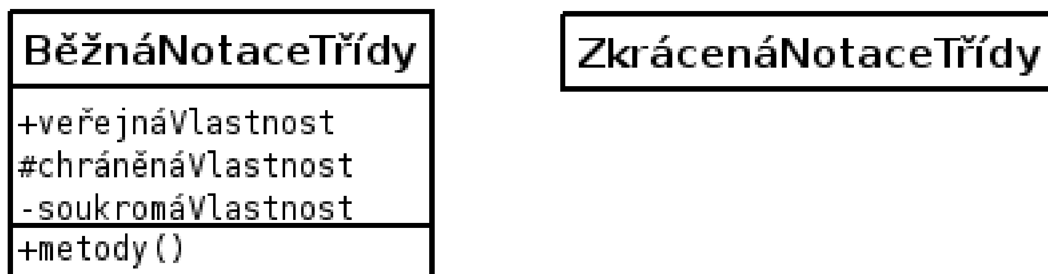
Obr. 6: Visual Paradigm for UML Community Edition [zdroj [6]]

2.3 Třídní diagram

Třídní diagram je základním a z našeho hlediska (modelování báze znalostí) také diagram nejdůležitější. Jeho důležitost je dána tím, že výsledky tohoto diagramu, tedy návrh tříd a vztahů mezi nimi je přímo převoditelný do podoby datových struktur.

Třída

Základním konstruktorem třídního diagramu je *třída*. Třída je virtuální reprezentací objektů (hmotných i nehmotných) reálného světa. Podobně jako objekty reálného světa i třídy mohou mít naspecifikovány určité *vlastnosti* (properties) a *činnosti* (metody), které vykonávají. Grafické znázornění třídy je patrné z obr. 7.



Obr. 7: Třída – zkrácená notace vs. plná notace

Všimněte si vlastností a metod, resp. jejich modifikátoru viditelnosti. Tyto indikátory znamenají formu jakou mohou být metody/vlastnosti použity v dalších třídách:
+ veřejné (jakákoliv třída může využít tuto vlastnost/metodu)
- privátní (může použít pouze třída, která tuto vlastnost/metodu obsahuje)
chráněná (mohou využívat třídy, které jsou obsaženy ve stejném balíčku jako je aktuální třída)

Pro náš účel můžeme veřejné vlastnosti chápat jako údaje, které je potřeba zjistit nebo nastavit, třeba prostřednictvím nějakého formuláře apod. Privátní vlastnosti můžeme chápat jako pracovní proměnné, které získáme zpracováním veřejných údajů. Chráněné vlastnosti můžeme pro náš účel ignorovat.

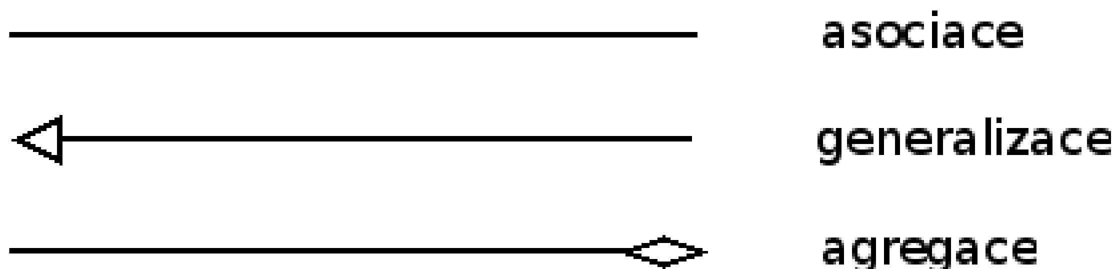
Metody mají pro náš účel spíše podpůrný charakter – nebudeme realizovat dynamické chování třídy v nějakém programovacím jazyce. Uvažovaní nad specifikací metod nám však může napomoci ve správném zachycení procesu řešení problému.

Rozlišujte pečlivě mezi třídou a instancí třídy. Třídu je potřeba chápat jako obecný pojem popisující určitou skupinu (třídu) objektů, instance třídy je popisem konkrétního výskytu třídy. Zkusme si to demonstrovat na příkladu: třída – pes, instance třídy pes – Váš Alík.

Instance třídy obvykle do třídního diagramu nedáváme, neboť tento diagram je obvykle konstruován obecně. Za určitých okolností však bývá výhodné zachytit v tomto diagramu i instance třídy a to zejména v okamžiku, kdy popisuji konkrétní situaci, s konkrétními objekty. Získám

tak třídní diagram tvořený instancemi, který mohu zobecnit do podoby třídního diagramu tvořeného třídami.

Kromě tříd v tomto druhu diagramů modelujeme i vazby mezi nimi. Tyto vazby nabývají podob asociací, generalizací a agregací. Pro grafické značení viz. obr. 8.



Obr. 8: Vazby mezi třídami

Asociace

Běžné vazbě mezi třídami říkáme *asociace*. Asociaci je možné pojmenovat, abychom usnadnili orientaci v diagramu. U vazeb specifikujeme také četnost (kardinalitu). Kardinalita se specifikuje pro obě zakončení vazby.

Možnosti kardinality:

1 (nebo vynecháno) – na takto označeném konci vazby musí existovat jedna instance třídy

1..* - na takto označeném konci musí existovat minimálně jedna instance třídy

0..3 – na takto označeném konci vazby musí existovat maximálně 3 instance třídy

Příklad použití naleznete na obr. 9.

Obr. 9: Použití asociace

Generalizace

Generalizace

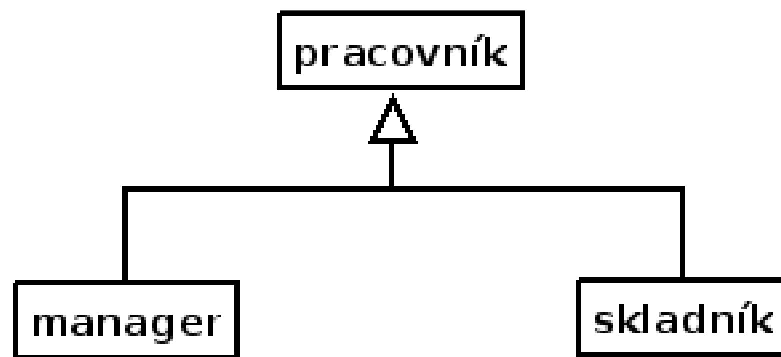
Zachycuje vztahy mezi třídou rodičem a potomkem. Vychází se z toho, že třída potomek dědí všechny vlastnosti a metody svého rodiče, kromě toho však může obsahovat další – rozšiřující vlastnosti a metody, popřípadě může těm původním, zděděným dát nový význam.

Na obr. 10 je zobrazen jednoduchý příklad použití dědičnosti pro popis různých typů pracovníků.

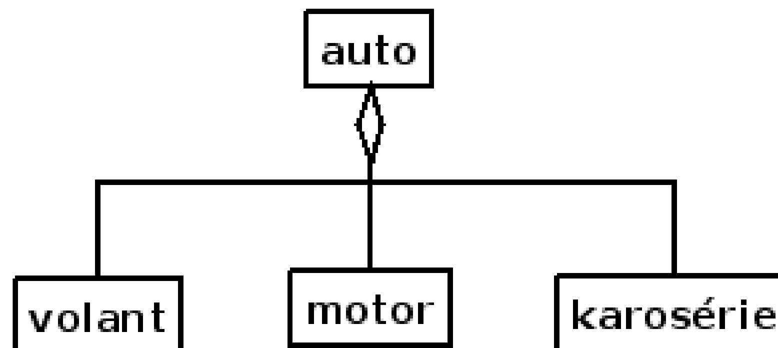
Agregace

Agregace

Agregační vazba nám pomůže v případě, že pracujeme s třídami, které se mohou skládat z dalších tříd. Příklad je znázorněn na obr. 11.



Obr. 10: Použití generalizační vazby pro popis vazeb mezi pracovníky



Obr. 11: Použití agregační vazby

2.4 Model jednání

Model jednání

Úkolem tohoto modelu je vytipovat *uživatele (aktory)* modelovaného systému a zjistit jakým způsobem se systémem pracují (*případy užití*). Jinými slovy pohlížíme na modelovaný systém jako na černou skříňku, ke které přistupujeme z vnějšku pomocí aktorů a zkoumáme, které funkce jsou od systému požadovány.

Jednoduchý případ použití je znázorněn na obr. 12.

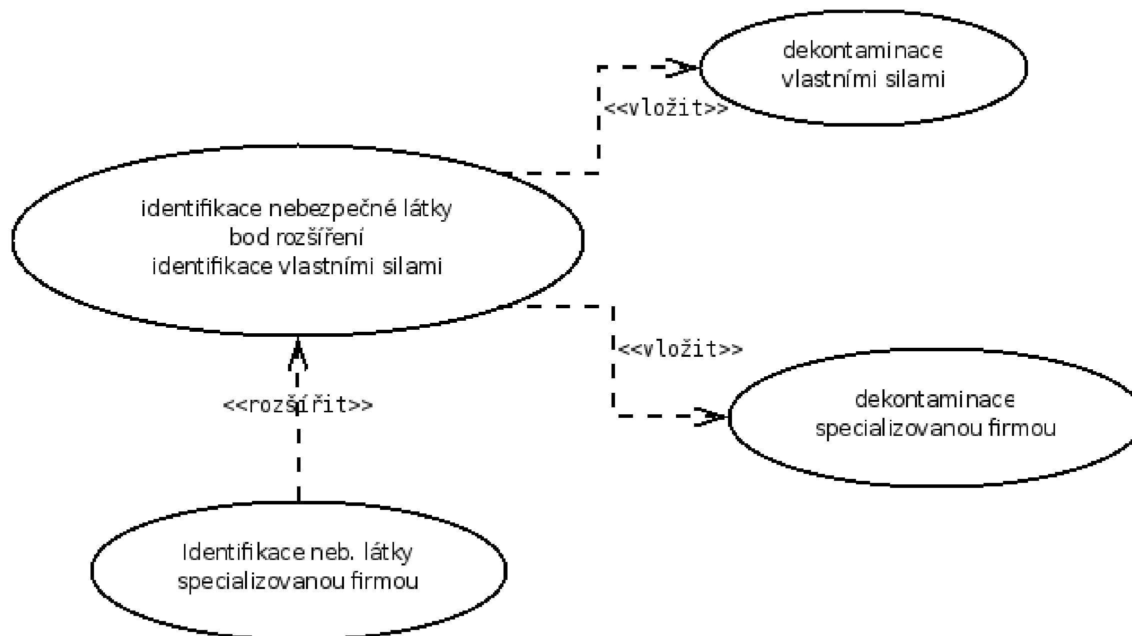
Obr. 12: Případ užití – jednoduchý případ

Případy užití mou být také ve vzájemném vztahu. Tento vztah ovšem nemá charakter podobný asociacím nebo agregacím, které známe z třídního diagramu. Jedná se o vztahy typu *vložit* nebo *rozšířit*.

Vztah typu vložit znamená, že kroky podniknuté v rámci případu užití jsou vloženy do dalšího, navazujícího případu užití. Tento typ vztahu

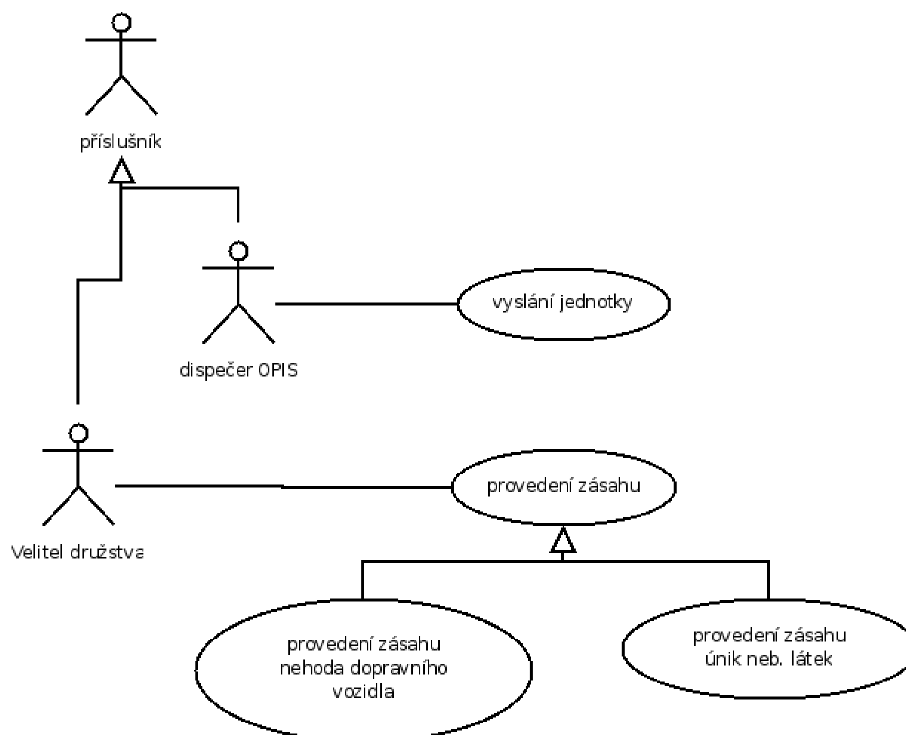
je označován <<vložit>>. Směr závislosti určuje šipka.

Vztah typů rozšíření nám dovoluje rozšířit chování případu užití o nějakou modifikovanou funkčnost. Toto rozšíření je možné provést pouze v určitých případech užitím která nazýváme body rozšíření. Tento typ vztahu označujeme klíčovým slovem <<rozšířit>> a opět specifikujeme i směr závislosti, viz. obr. 13.



Obr. 13: Vazby rozšířit a vložit modelu jednání

Kromě těchto specifických vazeb můžete v rámci modelu jednání použít i běžnou generalizační vazbu. Můžeme přitom zobecňovat nejen samotné případy užití, ale také aktory modelu. Způsob použití je patrný z obr. 14.



Obr. 14: Použití generalizační vazby v modelu jednání

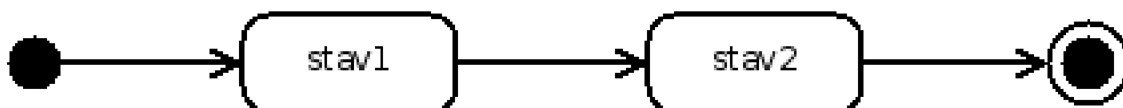
2.5 Stavový diagram

Stavový diagram

Stavový diagram umožňuje zachytit časové změny v modelovaném systému. Model jednání i třídní diagram byly z hlediska notace čistě statickými nástroji. Stavový diagram zavádí časové hledisko pomocí sledování změn stavů v jednotlivých objektech a vzájemných vazeb mezi těmito stavy.

Stavový diagram, vzhledem k tomu že je časově vymezen, musí obsahovat začátek a konec. Jednotlivé stavy mohou podobně jako třídy obsahovat stavové proměnné (odpovídá vlastnostem třídy) a činnosti (odpovídá metodám třídy). Podobně jako u tříd existuje i zjednodušená notace, která obsahuje pouze název stavu. Tento zkrácený zápis budeme nadále používat pro zápis stavových diagramů pro náš účel.

Grafickou notaci stavového diagramu si můžete prohlédnout na obr. 15.



Obr. 15: Stavový diagram

Podstav

Kromě běžných stavů, umožňuje UML zapisovat i tzv. *podstavy*. Podstavy rozumíme sekvence stavů, které podrobněji vysvětlují daný stav. Tato podřízená sekvence se zapisuje přímo do stavu (který se patřičně rozšíří), který je vysvětlován. K zápisu podstavů je možné dodat, že ne všechny nástroje umožňují tento typ zápisu, vlastně se dá říci, že tento zápis je podporován pouze u špičkových CASE nástrojích, se kterými se pravděpodobně nesetkáte.

Tam kde chybí podpora podstavů musíme podstavy obejít a to obvykle tak, že stav, který je nutno podrobněji rozebrat popíšeme samostatným stavovým diagramem.

2.6 Scénáře činností

Scénáře činností

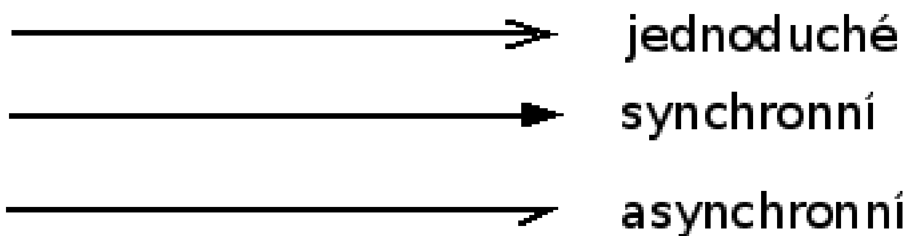
Scénáře činností jsou dalším nástrojem, který nám umožní vnést do analýzy systému časové hledisko. Oproti stavovým diagramům však scénáře činností sledují komunikaci mezi jednotlivými objekty, tak jak byly navrženy formou tříd v třídním diagramu.

Jednotlivé objekty si mezi sebou posílají zprávy. Tyto zprávy jsou předávány jedním ze třech způsobů, které graficky značíme různými typy šipek, viz. obr. 16.

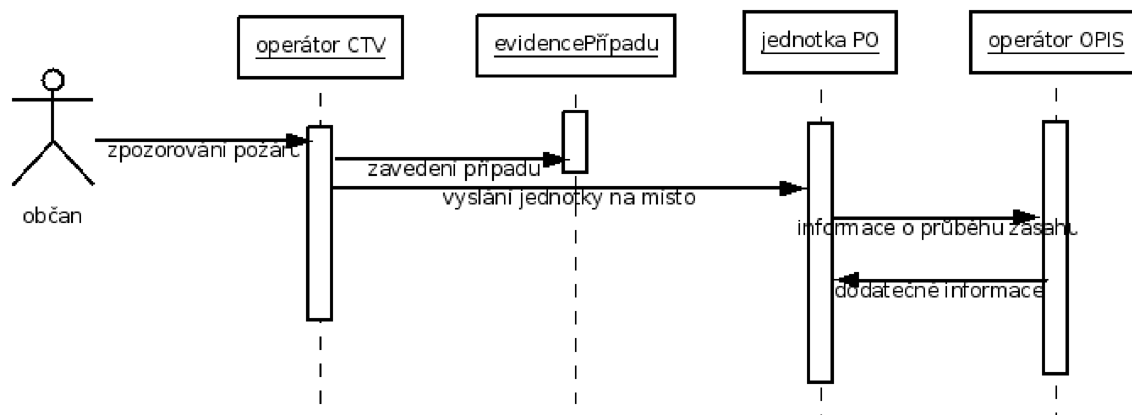
Z hlediska našich omezených potřeb si můžeme zjednodušit práci a

Časově rozlišujeme posloupnost zpráv pomocí posunu šipek směrem dolů ve scénáři. Obvykle lze vysledovat také prvotního původce činnosti – člověka. V diagramu zachycujeme pomocí symbolu aktora, známého z modelu jednání.

Příklad scénáře činností je zachycen na obr. 17.



Obr. 16: Zachycení předávání zpráv ve scénáři činnosti.



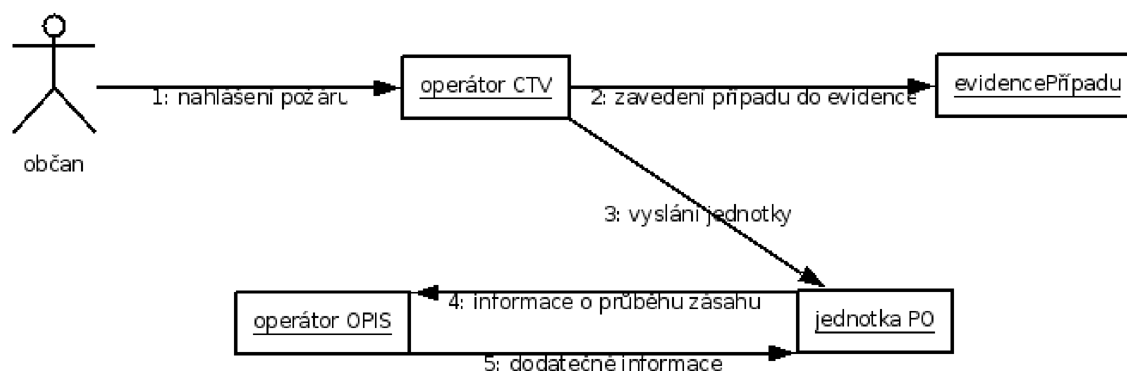
Obr. 17: Příklad scénáře činností

2.7 Diagram spolupráce

Diagram spolupráce

Diagram spolupráce je alternativním zápisem zasílání zpráv mezi objekty ke scénáři činností. Rozdíl je ve způsobu zápisu zasílání zpráv. Zatímco ve scénáři činností byly objekty seřazeny vedle sebe na horním okraji diagramu, objekty v diagramu spolupráce mohou být umístěny libovolně.

Podívejme se na zápis příkladu z obr. 17 do formy diagramu spolupráce (viz. obr. 18).



Obr. 18: Přepracovaný příklad scénáře činnosti (obr. 17) do podoby diagramu spolupráce

Základní rozdíl mezi oběma typy diagramů je místo, které zabírají. Scénáře činností mají tendenci se neúměrně roztahovat, ale z hlediska čitelnosti předávání zpráv a určení jejich pořadí je lepší. Diagram spolupráce oproti tomu zabírá mnohem méně místa, ale může být také mnohem méně čitelný. Z tohoto důvodu je také nutné označovat sekvenci

činností pořadovým číslem.

2.8 Diagram činností

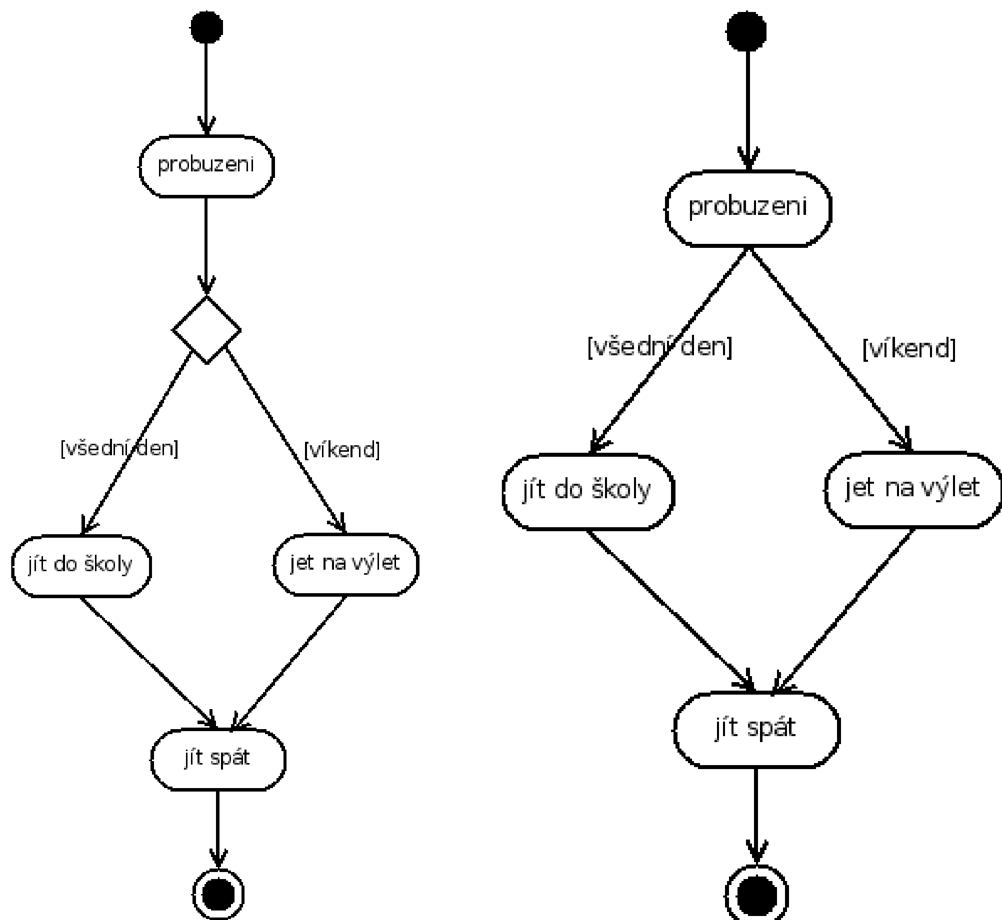
Diagram činností

Diagram činností přináší opět dynamický pohled na modelování systému. Koncepčně je velmi podobný stavovému diagramu, ze kterého je také odvozen, poskytuje však trochu odlišný pohled na systém.

Stavový diagram chápal systém jako sled stavů a transformací těchto stavů (činností) ústících do nových stavů systému. Diagram činností se zaměřuje na modelování právě těchto transformací (činností) a zcela pomíjí stavy.

Podobně jako stavový diagram i diagram činností má explicitně definován počátek a konec. Diagram činností má však také několik dalších prvků, které lze použít. Prvním z nich je rozhodování, reprezentované kosočtvercem. Jedná se o symbol, který je totžný se symbolem rozhodování známým z algoritmů. Používáme jej vždy tam, kde dochází k nějakému vědomému procesu rozhodnutí mezi několika variantami. V diagramu činností symbol rozhodování není povinný, ale zvyšuje u čitatele čitelnost diagramu.

Demonstraci použití symbolu rozhodování naleznete na obr. 19.

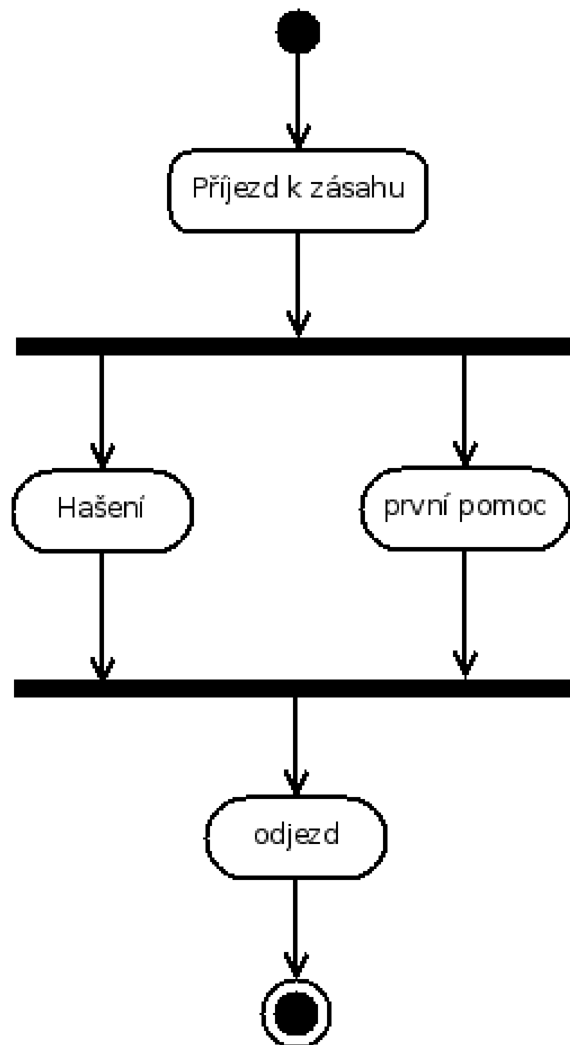


Obr. 19: Použití a nepoužití symbolu rozhodování (oba případy jsou správně a mají stejný význam)

Diagram činností umožňuje také zachytit paralerně (současně) běžící procesy. Zachycení je možné s použitím symbolu rozvětvení a

souběhu. Pro diagram by přitom mělo platit, že paralerně probíhající proces by se opět měly spojit nebo by měly být ukončeny symbolem konce činnosti. Toto jednoduché pravidlo opět slouží pro zachování čitelnosti diagramu.

Příklad použití rozvětvení naleznete na obr. 20.



Obr. 20: Modelování paralerně probíhajících procesů

2.9 Diagram komponent

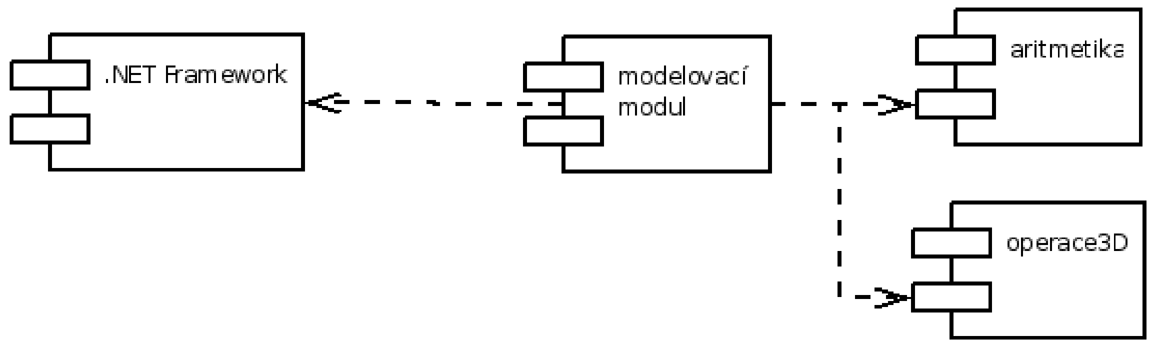
Diagram komponent

Diagram komponent má z hlediska modelování báze znalostí minimální význam. Tento diagram umožňuje modelovat vztahy mezi jednotlivými komponentami systému, ale i komponentami fungujícími mimo systém, které s modelovaným systémem ale nějak komunikují.

Komponentou rozumíme samostatný softwarový modul (knihovna, ActiveX prvek apod.).

Příklad použití diagramu komponent je na obr. 21.

Dle obrázku závisí základní modelovací modul na .NET Framefork a dvou knihovnách, jedné zabývající se aritmetikou a druhé operacemi ve 3D.



Obr. 21: Příklad použití diagramu komponent

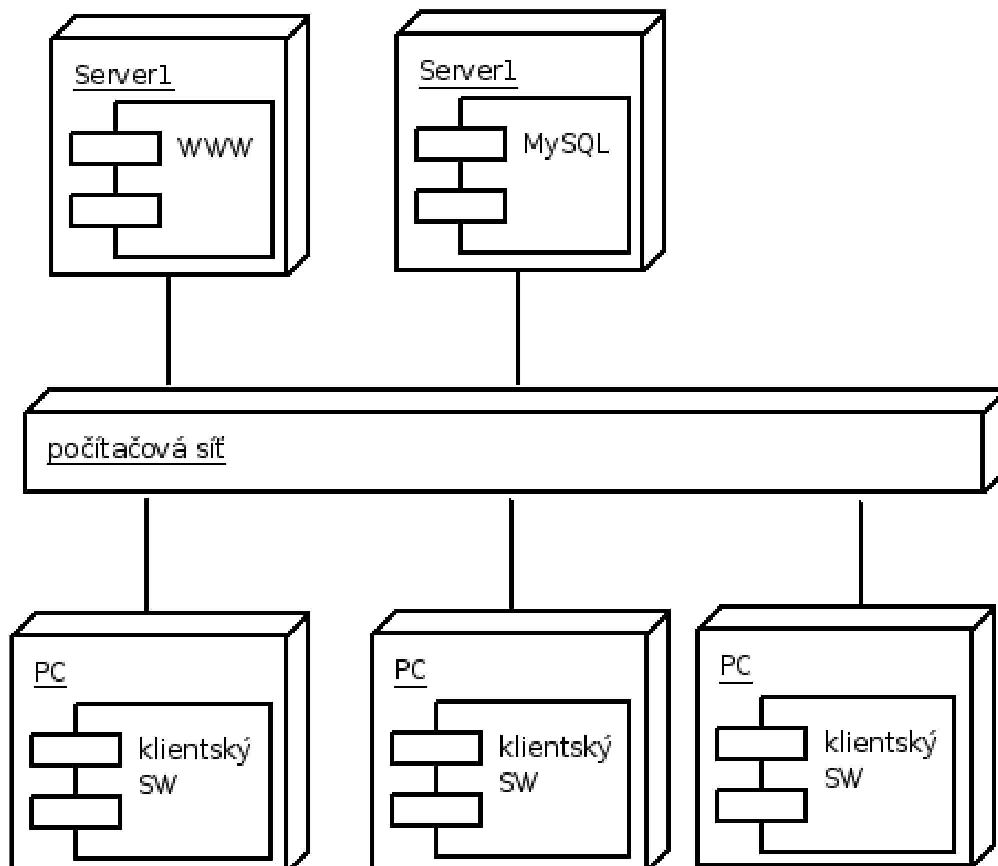
2.10 Diagram nasazení

Diagram nasazení

Diagram nasazení zkoumá operační prostředí, ve kterém bude nasazen modelovaný systém. Umožňuje tak zkoumat které části systému budou na jakých místech a v jakém vzájemném vztahu budou.

K tomuto účelu využívá diagram symboliku diagramu komponent. Komponenty zde mají charakter samostatných systémů, které běží v jednotlivých uzlech obvykle síťového prostředí.

Demonstrujme tento postup na jednoduchém modelu počítačové sítě (viz. obr. 22).



Obr. 22: Diagram nasazení – model počítačové sítě

2.11 UML závěr

Modelování pomocí jazyka UML probíhá obvykle iteračně. Různé pohledy na systém reprezentované různými diagramy UML nám umožňují

získat komplexní pohled na systém. To nám také umožní, abychom v jednotlivých diagramech odhalili chybějící souvislosti načrtnuté v odlišných druzích UML diagramů (třeba na základě třídního diagramu zjistíme, že v diagramu případů užití chybí podstatný aktor).

Z hlediska procesu modelování se nedá říci, že by některý způsob modelování byl optimální, vždy je nutné vycházet z dané situace, účelu za kterým modelování probíhá. Velmi často se ale během modelování začíná konstrukcí diagramu případů užití, protože tento diagram nám řekne co má modelovaný systém dělat z pohledu z vnějšku a nevyžaduje žádné znalosti vnitřního fungování systému. Požadované funkce a jejich vzájemné vztahy můžeme zkoumat v třídních diagramech, diagramech činností apod.

Diagramem nasazení obvykle končíme. Teprve až je celý systém hotov – budeme znát jaké jsou úplné požadavky na nasazení a tyto můžeme zachytit právě v tomto druhu diagramu.

Je zcela jisté, že bychom dokázali vymyslet případ, kdy pořadí použití jednotlivých typů diagramů bylo odlišné. Obecně začínáme tím co máme nebo víme, vše ostatní analyzujeme a zachytáváme ve vhodných typech diagramů.

Kontrolní otázky:

1) Vyjmenujte alespoň 3 druhy UML diagramů?

2) Jaký je vztah mezi třídním diagramem a diagramem případů užití?

3) Jaký je rozdíl mezi činnostmi a stavy?

4) Jaký je rozdíl mezi kreslícími nástroji (Dia, Visio) a CASE nástroji (StarUML, Poseidon)?

5) Jaký je rozdíl mezi diagramem spolupráce a scénáři činností?

Otázky k zamyšlení:

1) Zkuste vymyslet případ kdy by bylo vhodnější použít opačné pořadí použití diagramů než je popsáno v závěru této kapitoly.

2) Zkuste napsat diagram činností popisující Vaše studium tohoto textu. Použijte k zápisu Vámi zvolený softwarový prostředek.

3 Expertní systémy

Náhled kapitoly

Expertní systémy jsou pravděpodobně první oblastí která se začala rozvíjet v oboru, který dnes nazýváme souhrnně umělá inteligence. Koncepčně vychází ze snahy formálně popsat způsob rozhodování experta a zachytit jej uměle vytvořeném systému – expertním systému.

Po prostudování této kapitoly budete vědět

- co je to expertní systém
- jak takový systém funguje
- jakým způsobem je konstruována báze znalostí

Čas pro studium

Na plné pochopení této kapitoly budete potřebovat přibližně dvě hodiny.

3.1 Stručná historie expertních systémů

Základy umělé inteligence jako vědy položil v roce 1950 britský matematik a kryptoanalytik Alan Turing ve svém článku *Computing Machinery and Intelligence*. V tomto článku mimo jiné definuje jakým způsobem určit, že systém je skutečně „inteligentní". Tato pravidla jsou dnes označována jako tzv. *Turingův test*.

Turingův test

Turingův test spočívá v použití imitační hry. Této hry se účastní několik lidí, kteří mají za úkol imitovat počítač, a počítač, který má za úkol imitovat člověka. Hráč musí rozlišit, aniž by svého protivníka viděl, kdo je počítač a kdo skutečný člověk. Rozhodování probíhá na základě pokládání otázek hráčem a analýze odpovědí protihráče (člověka nebo stroje).

Pro úplnost dodejme, že v historii nebyl zaznamenán případ špatné klasifikace počítače, ale několika lidem se povedlo úspěšně imitovat stroj.

ELISA

V roce 1965 nastává další milník v oblasti umělé inteligence, v tomto roce totiž Joseph Weizenbaum sestavuje program ELISA. Základním úkolem ELISy je parodovat psychoanalytika. Jedná se o konverzační program, který pracuje tak, že analyzuje věty které se mu předkládají a modifikuje je tak aby z nich položil otázku.

Koncepce konverzace vedené strojem za posledních 40 let notně pokročila kupředu. Jedním z nejúspěšnějších umělých „kecálků" poslední doby je A.L.I.C.E, nebo chcete-li Alice [10]. Výsledek krátké konverzace mezi mnou a Alicí můžete vidět níže:

Human: Hello, Alice
ALICE: Hi there!
Human: Nice to be here.

ALICE: Thank you for your kindness.
Human: How are you today?
ALICE: Ah. I am functioning within normal parameters.
Human: That sounds good.
ALICE: It sounds like it to me too.
Human: Do you function outside normal parametrs too?
ALICE: I have never considered it.
I am always on the Web.

Pokud vládnete anglickým jazykem, můžete pokecat s Alicí nebo jinými virtuálními osobami sami [10].

MYCIN

V roce 1974 se konečně objevuje první expertní systém nazvaný MYCIN. Tento systém je zaměřen na doporučování vhodných léků na základě analýzy zdravotní karty pacienta, zejména s ohledem na vzájemné působení jednotlivých léků.

3.2 Základy expertních systémů

Plné expertní systémy, prázdné expertní systémy

Od uvedení expertního systému MYCIN doby se objevila celá řada dalších expertních systémů zaměřených na jiné oblasti. Všem těmto expertním systémům říkáme tzv. *plné expertní systémy.*

Plností expertních systémů přitom rozumíme plnost báze znalostí údaji pro řešení určitého problému. Práce s tímto systémem pak obvykle spočívá převážně v konzultaci se systémem. *Prázdné expertní systémy* obsahují pouze prázdnou znalostní bázi. Předtím, než bude takový expertní systém použit je nutné jej naplnit údaji.

Kromě tohoto základního dělení můžeme nalézt i jiná dělení těchto systémů a to na *diagnostické, plánovací a hybridní*.

Diagnostické expertní systémy

Diagnostické expertní systémy mají za úkol diagnostikovat, tedy na základě zjištěných údajů rozhodnout o nějaké variantě řešení z konečného počtu předem známých řešení.

Představme si to na příkladu systému pro diagnostiku chorob v okamžiku, kdy do něj vložíme jenom některé choroby – třeba viróza, chřipka a angína. Systém sám pak nebude moci diagnostikovat další choroby, protože je jednoduše nebude znát a není sám schopen odvodit nějaké nové diagnózy.

Plánovací expertní systémy

Úkolem plánovacích expertních systémů je plánování. Pomocí těchto systémů se řeší úlohy, u kterých je znám počáteční stav a cíl řešení. Expertní systém má za úkol naplánovat posloupnost kroků, které k tomuto cíli vedou. Úkolem znalostního inženýra je omezit možnou kombinaci těchto možných kroků a nějakým způsobem zabezpečit ohodnocení příspěvku těchto kroků k dosažení cíle.

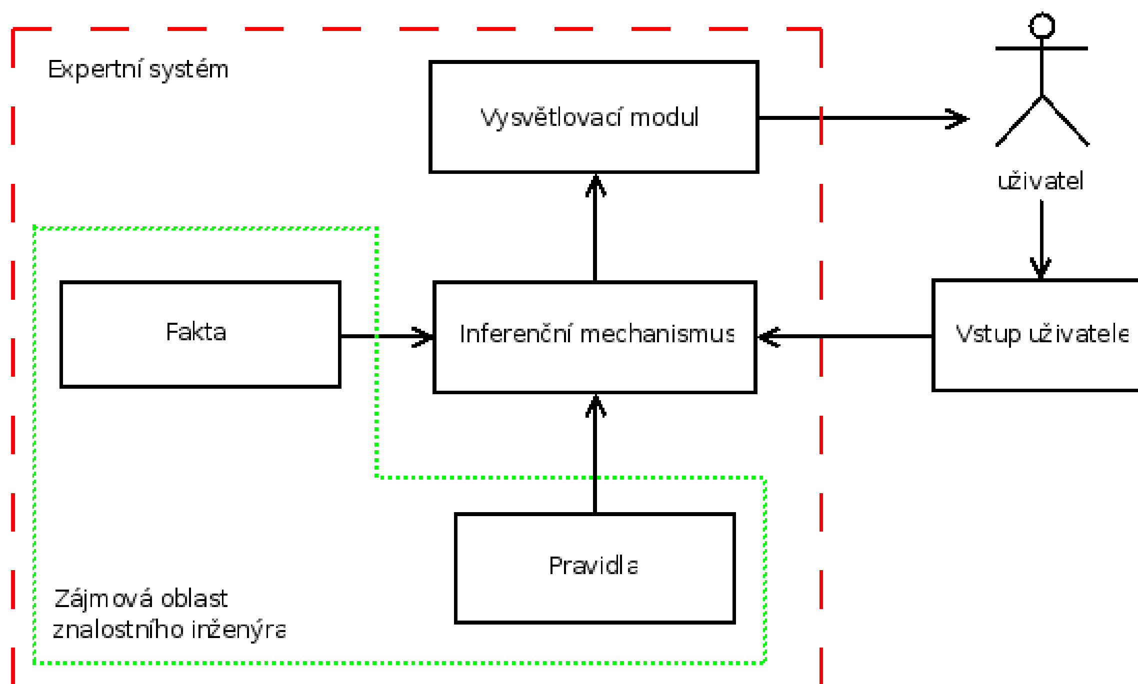
Na základě tohoto hodnocení totiž systém může zkonstruovat účelovou funkci, tak aby vybral vhodné řešení. Tady je potřeba podotknout, že vyhledávané řešení není v případě expertních systémů (ale

i některých jiných metod se kterými jste se mohli setkat třeba v předmětu Modelování rozhodovacích procesů) optimální, ale suboptimální – tedy nejlepší získatelné při časovém omezení i omezené výkonnosti výpočetní techniky.

Hybridní expertní systémy

Hybridní expertní systémy v sobě kombinují vlastnosti diagnostických i plánovacích expertních systémů, přičemž obvykle převažuje diagnostická složka.

Podívejme se na obecné schéma expertního systému



Obr. 23: Schéma expertního systému

Inferenční mechanismus z obr. 23 tvoří jádro expertního systému. Jeho úkolem je zpracovávat údaje zadané uživatelem a porovnávat je s údaji zjištěnými z báze znalostí (tvořené pravidly a fakty). Výsledek této činnosti je předkládán uživateli ve formě vysvětlovacího modulu.

Inferenční mechanismus je v expertním systému zabudován napevno, základním problémem použití tak bude z našeho hlediska formulace báze znalostí, tedy naplnění expertního systému.

Konceptualizace

Na bázi znalostí máme přitom několik požadavků. Předně budeme potřebovat, aby tato báze byla rozšiřitelná, tedy abychom do ní podle potřeb mohli doplňovat fakta a pravidla (abychom systém mohli začít plnit). Jazyk, ve kterém budeme tyto pravidla musí být formální a musí nám umožnit popsat všechny objekty problémové oblasti a vztahy mezi nimi. Proces určování těchto objektů a vztahů mezi nimi nazýváme konceptualizace.

Pro grafický popis tohoto procesu můžeme použít zápis z nám už teď známého jazyka UML. Z hlediska potřeb expertního systému pro nás bude mít enormní význam třídní diagram.

Formální zápis univerza získaného konceptualizací problému

nazýváme ontologie. Existuje celá řada jazyků, které nám umožní formálně vyjádřit onotologii jako rámec problému a znalosti, vyjmenovat zde můžeme např. jazyky KML nebo KIF.

Tyto jazyky nejsou obecně použitelné, výběr výrazového prostředku úzce souvisí s volbou expertního systému a toho jaké jazyky podporuje.

Z konstrukce báze dat nám také vyplývá omezení z hlediska problému, které jsou expertní systémy schopny řešit. Pokud problémovou oblast nejsme schopni formálně vyjádřit, ať už vůbec nebo pouze obtížně, není tento problém řešitelný pomocí expertních systémů.

Pomocí expertních systémů také není vhodné řešit problémy které naopak velmi dobře známe a jsme je schopni explicitně zachytit prostřednictvím formálních modelů např. vzorcem, soustavou diferenciálních rovnic apod. V takovém případě je výhodnější řešit konvenčními metodami – poskytují rychleji přesnější výsledky.

3.3 Definice ontologie pomocí Protege 2000

Protege2000 [12] je open-source systémem pro definici ontologií. Protože pro praktické ukázky použití expertních systémů budeme využívat systém CLIPS [13], budeme také potřebovat rozšiřující modul, který dokáže vytvořenou ontologii exportovat do formy stravitelné pro tento expertní systém. Tuto funkcionalitu zajišťuje plug-in CLIPSTab [14].

Protege2000 je naprogramován v jazyku JAVA a z tohoto důvodů je potřeba mít pro provoz nainstalován Java Runtime Environment (JRE nejlépe poslední verze). CLIPSTab je naprogramován v kombinaci C++ a JAVY a z tohoto důvodu je jeho běh omezen pouze na operační systémy MS Windows, Protege2000 běží i na jiných operačních systémech jako je např. LINUX nebo OS X.

Instalace probíhá tak, že nejprve nainstalujete JRE ve verzi minimálně 5, potom Protege2000, a teprve potom do adresáře plugins v instalačním adresáři Protege2000 nakopírujete CLIPSTab. Pozor: ProtegeClipsInterface.dll je potřeba nakopírovat do hlavního adresáře Protege2000, jinak se plug-in nespustí.

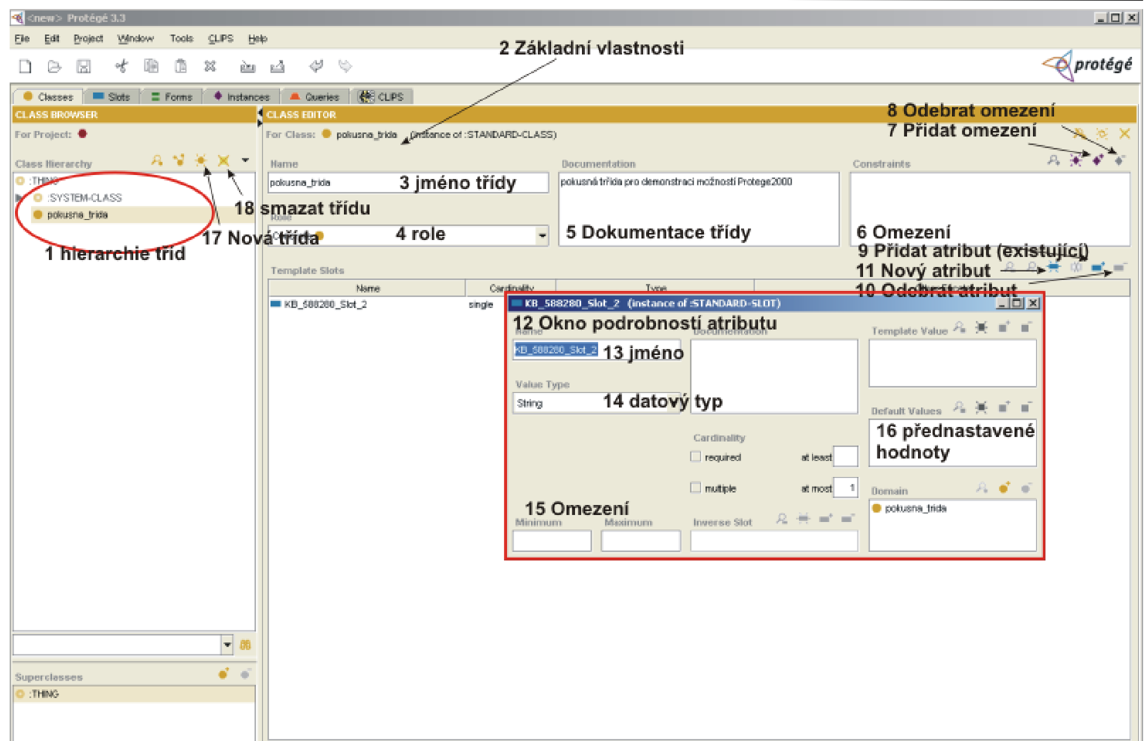
CLIPSTab můžete zapnout z menu Project -> Configure -> na záložce TabWidget -> zaškrtněte CLIPSTab.

Po spuštění vypadá okno Protege2000 s připojeným CLIPSTab pluginem následovně (viz. obr. 24).

V prvním kroku se podívejme na definici tříd. Třídu zde chápeme stejně jako jsme si ji nadefinovali v první kapitole. Třídní diagram je tak jediným diagramem, který je přímo převoditelný a použitelný pro náš účel.

V Protege2000 vytváříme hierarchii tříd. všechny tyto třídy jsou odvozeny od základní třídy :THING, která je abstraktní (nemohou existovat instance této třídy).

Třídy vytváříme buď z kontextového menu hierarchie tříd (1) a nebo kliknutím na tlačítko nová třída (17). Třídu můžeme opětovně



Obr. 24: Protege2000 základní rozhraní

smazat kliknutím na tlačítko smazat třídu (18).

Pro jednotlivé třídy definujeme také kromě názvu (3), také zda třída má být abstraktní nebo konkrétní (4). Připomínám, že abstraktní třídy mohou být použity pouze jako rodič dalších tříd, konkrétní třídy mohou být použity i jako rodič i k vytváření instancí třídy.

Protege2000 umožňuje vytvářet jednoduchou dokumentaci, kterou můžeme vepisovat do dokumentačních polí k tomu určeným. Tato pole jsou obsažena prakticky v každém formuláři, komentář tak může být velmi cílený.

Třída může obsahovat taktéž omezení. Tedy omezení ve smyslu vytvoření pravidel, které budoucí instance třídy musí splňovat. Omezení můžeme definovat do určité míry i na úrovni jednotlivých atributů.

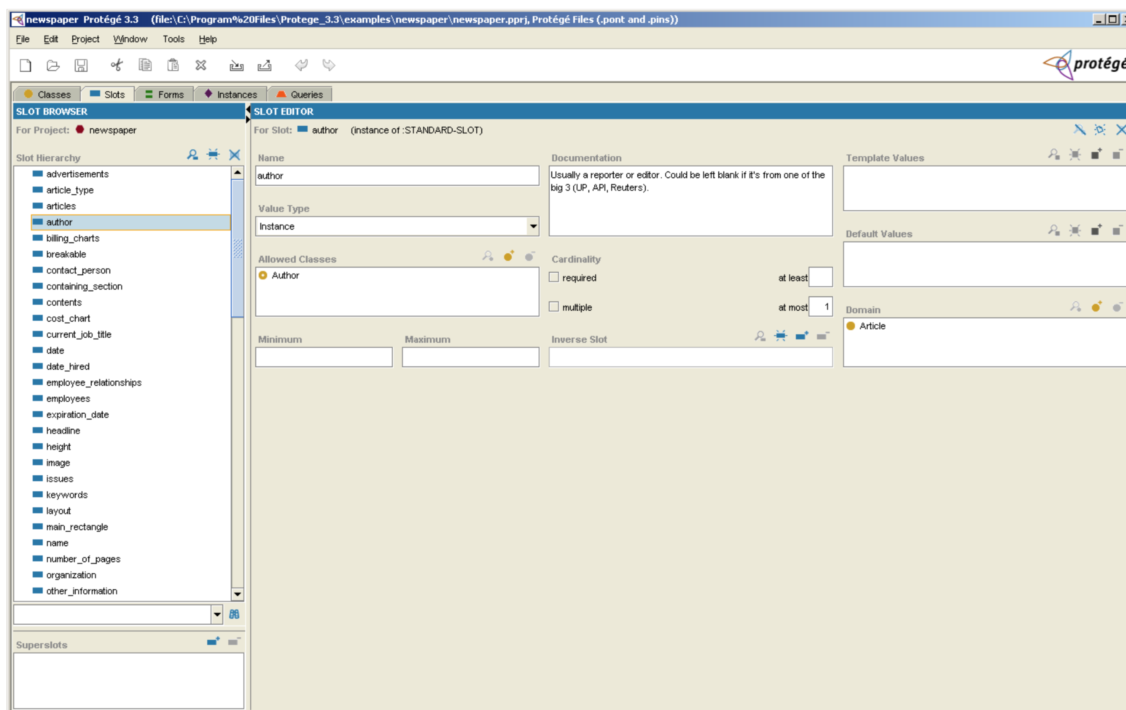
Atributy třídy (v Protege2000 se používá název slot) kliknutím na tlačítko (11). Prosím nepleťte si tlačítko vytvoření nového atributu (11) a přidání atributu (9). Přidání je zde myšleno ve smyslu přidání již existujícího definovaného atributu, zatímco vytvořením se myslí vytvoření úplně nového atributu.

Pro jednotlivé atributy kromě jména definujeme taktéž datový typ. Definice datového typu je nesmírně důležitá, protože ovlivňuje jakým způsobem budeme moci daný atribut využít (např. čísla můžeme použít pro výpočty, ale textové řetězce ne). Datový typ také ovlivní koncepci formuláře, který pro nás může Protege2000 vytvořit (bez programování!). Datový typ lze definovat již na úrovni třídního diagramu v UML. Pokud se tak stalo je definice tříd v Protege2000 jednodušší.

Atributy lze editovat i ze záložky nazvané Slots. V tomto okně se

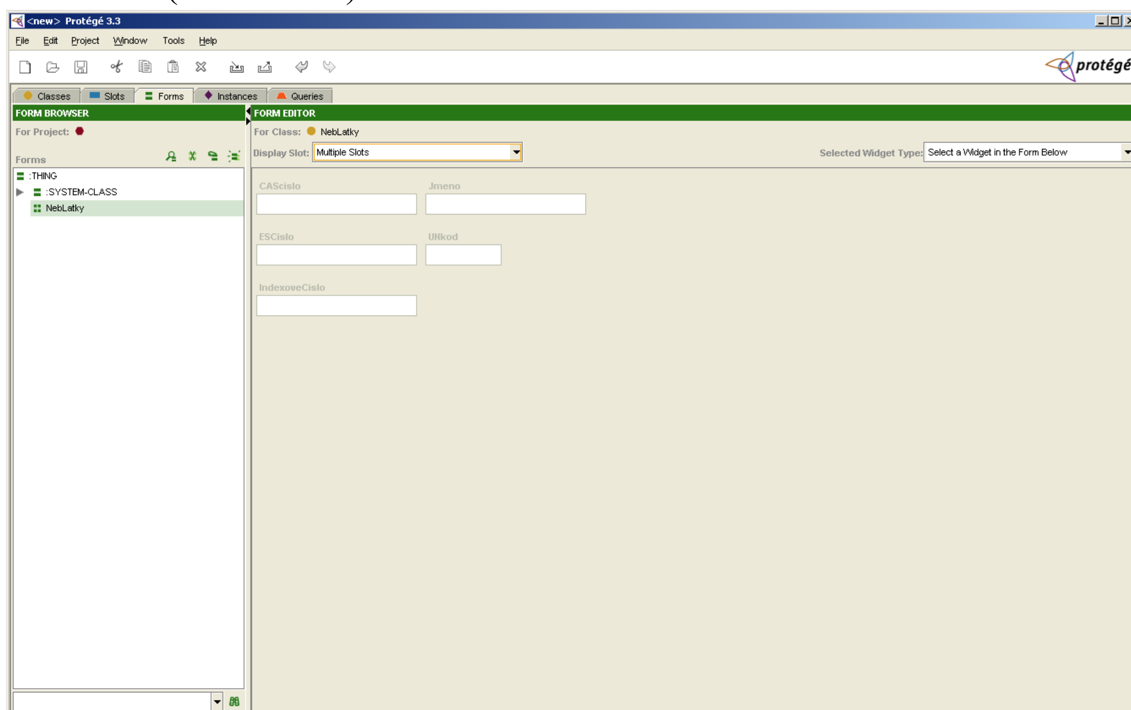
zobrazí všechny atributy všech tříd v přehledném rozhraní. Možnost samostatné definice atributů je zde dána jako podpora pro tvorbu znovupoužitelných komponent. Atribut tak definujeme pouze jednou a pak jej využíváme ve všech zájmových třídách.

Editační okno uvidíte na obr. 25.



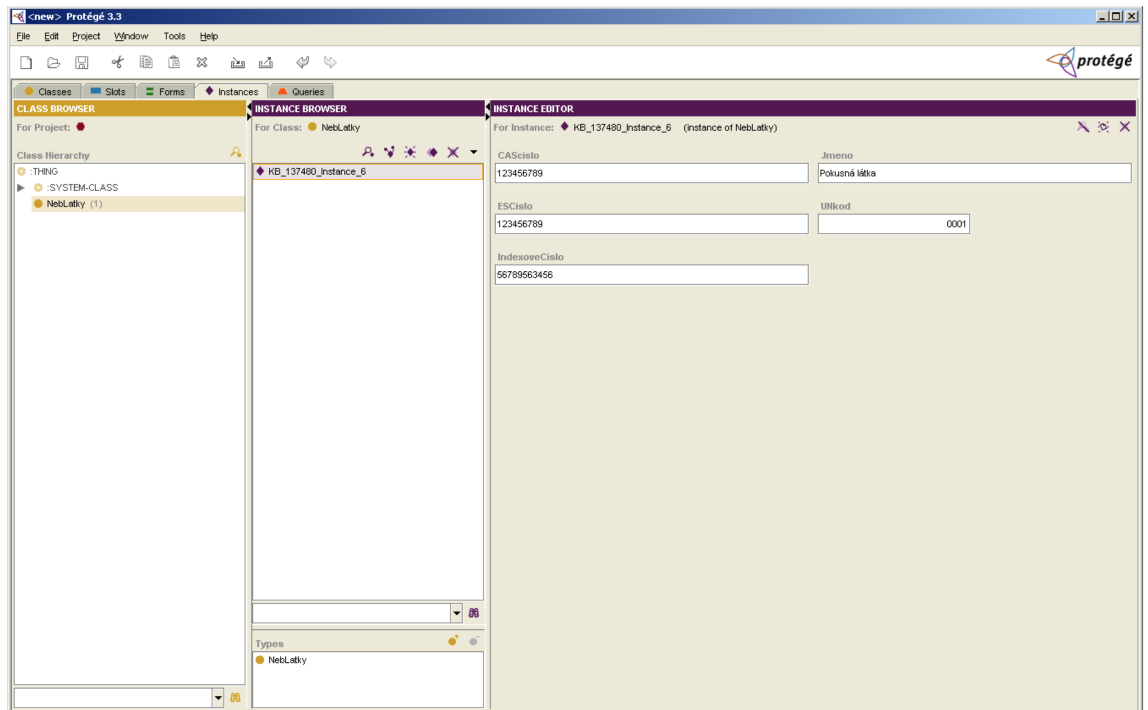
Obr. 25: Záložka slots – editace atributů

Všechny třídy, které se vytvoří na záložce tříd, se automaticky transformují do podoby formulářů. Na záložce formulářů máme možnost už jen poupravit umístnění a vizuální podobu jednotlivých prvků formuláře (viz. obr. 26).



Obr. 26: Úprava vzhledu formulářů

Na záložce instances, máme pak možnost tyto formuláře využít pro vyplnění instancí tříd. Z našeho hlediska tvoří instance tříd fakta, která budeme následně používat v rámci báze znalostí. Vzhled definice instancí je patrný z obr. 27.



Obr. 27: Definice instancí

V praxi se tento postup příliš nepoužívá (myšleno vyplňování přímo z prostředí definice ontologie). Obvykle se fakta získávají z nějakého zdroje dat jako jsou databáze nebo nějaký informační systém.

Pro začátečníka nicméně tento postup může být výhodný, protože se vyhne nutnosti definovat (spíše programovat) rozhraní mezi expertním systémem a tímto zdrojem dat.

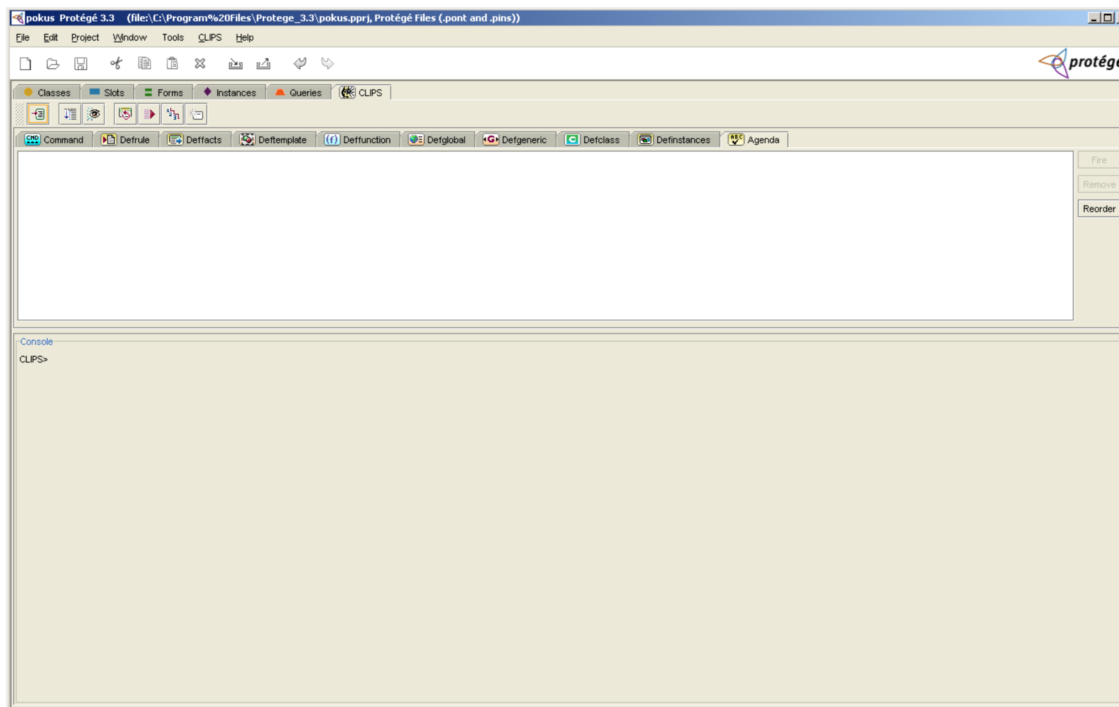
Poslední ze standardních záložek je záložka query, tedy dotaz. V rámci této záložky dostáváte do rukou nástroj pro výběr instancí tříd na základě definice omezujících parametrů. Z našeho hlediska a potřeb má tento nástroj minimální význam.

Záložka CLIPSTab je dostupná pokud byl aktivová CLIPSTab plug-in (viz. výše). Vzhled CLIPSTab plug-inu vidíte na obr. 28.

Teoreticky plug-in poskytuje grafické uživatelské rozhraní pro expertní systém CLIPS a můžete z něj dělat, vše co je potřeba, problémem je že musíte vědět co je potřeba, tedy alespoň na minimální úrovni vědět jakým způsobem CLIPS pracuje a to ještě nevíte, neboť jsme si to ještě neřekli.

Z důvodu, že ještě neznáme CLIPS se omezíme pouze na transformaci námi vytvořené ontologie do podoby stravitelné CLIPS. Import je intuitivní, prostě klikněte na první ikonu zleva na záložce (bublinková nápověda import ontology).

Pro jednoduchou třídu NebezpečnéLátky obsahující textové atributy



Obr. 28: CLIPSTab plug-in

jméno, UNKód, ESČíslo, CASČíslo se vygeneruje následující kód.

```
CLIPS>

(defclass NebezpecneLatky
	(is-a USER)
	(role concrete)
	(pattern-match reactive)
	(single-slot jmeno
		(type STRING)
;+		(cardinality 0 1)
		(visibility public)
		(create-accessor read-write))
	(single-slot UNKod
		(type INTEGER)
;+		(cardinality 0 1)
		(visibility public)
		(create-accessor read-write))
	(single-slot CASCislo
		(type STRING)
;+		(cardinality 0 1)
		(visibility public)
		(create-accessor read-write))
	(single-slot ESCislo
		(type STRING)
;+		(cardinality 0 1)
```

```
            (visibility public)
            (create-accessor read-write))
       (single-slot IndexoveCislo
            (type STRING)
;+          (cardinality 0 1)
            (visibility public)
            (create-accessor read-write)))
CLIPS> (definstances pokus)
```

Protege2000 používá objektovou notaci (označovaná zkratkou COOL). Je očividné, že pro pochopení fungování expertního systému taková notace není úplně cool, takže ji opustíme a použijeme jednodušší neobjektovou notaci.

3.4 Šablony CLIPS

Šablony

Šablony umožňují formálně zachytit strukturu fakt v bázi znalostí. To, co jsme definovali v kapitole 2.3 nebylo nic jiného než šablona. Zkusme si rozebrat jednoduchý příklad šablony z předchozí podkapitoly (příklad s nebezpečnou látkou).

```
(deftemplate NebezpecnaLatka „informace o nebezpecne latce"
     (slot jmeno (type STRING))
     (slot UNKod (type NUMBER) (range 1 9999))
     (slot CASCislo (type STRING))
     (slot ESCislo (type STRING))
     (slot IndexoveCislo (type STRING))
     ;případné další atrinbuty nebezpečných látek
)
```

Takže několik pravidel práce s CLIPS. Existuje celá řada verzí CLIPS. Základní verze pracuje z příkazové řádky. Aby bylo možné odlišit, co je a co není součástí jednoho příkazu, je nutné všechny příkazy uzavřít do kulatých závorek.

Tyto příkazy je možné zadávat z příkazové řádky, ale to je samozřejmě poměrně nepohodlné, proto se velmi často šablony i pravidla definují mimo prostředí CLIPS v nějakém textovém editoru. Pro tyto soubory se často používá přípona .clp.

Příkazy jsou citlivé na velikost písmen. Jednotlivé příkazy jsou obvykle psány malými písmeny a pro datové typy se používá písmo velké.

Nyní k samotné šabloně. Příkazem pro definici šablony je deftemplate, za kterým následuje jméno šablony, v uvozovkách je specifikován vysvětlující text, tak aby byla zjednodušena orientace ve zdrojovém kódu šablon.

Jednotlivé atributy se definují klíčovým slovem slot, za kterým následuje jméno atributu a naspecifikovaný datový typ atributu. Pro atributy, tam kde to má smysl, je možné definovat omezení, např. rozsahem. V našem případě je takovým způsobem nadefinováno omezení atributu UNKod, které může být pouze v rozmezí 1 – 9999.

Pro atributy je možné nadefinovat také předvolenou hodnotu, pomocí klíčového slova default. V našem případu to smysl moc nemá, proto příklad předvolby naleznete až zde.

```
(slot ESCislo (type STRING) (default -)
```

V tomto případě je tím předvoleným znakem „-“. Toto samozřejmě nejsou všechny možnosti, které máme při definici šablon, pouze ty nejjednodušší.

Podívejme se na možnosti manipulace s šablonami.
(list-deftemplates) - vypíše seznam aktivních šablon
(undeftemplate jméno_šablony) - odebere šablonu z báze znalostí
(clear) - smazání báze znalostí v paměti
(save „c:\\cesta\\soubor.clp“) - uložení báze znalostí to souboru soubor.clp
(load „c:\\cesta\\soubor.clp“) - načtení báze znalostí ze souboru soubor.clp

Během načtení dojde k definici nedefinovaných šablon a redefinici (přepsání v paměti) již načtených šablon.
(ppdeftemplate jméno_šablony) - zobrazí na obrazovce obsah šablony (její definici)

3.5 Definice fakt

Fakta

Fakta definujeme podobným způsobem jako šablony. Základním příkazem pro tuto editaci je deffacts. Můžeme zadávat libovolná fakta podle libovolné šablony. Podmínkou však je, aby tato šablona byla načtena v paměti pomocí deftemplate (je jedno zda je šablona zadána ručně nebo načtena pomocí funkce load).

U faktů již zapisujeme konkrétní hodnoty. Podívejme se na příklad používající šablonu NebezpecneLatky definovanou výše.

```
(deffacts  fakta-nebezpecne-latky  „zakladni  udaje  o  neb.
latkach“
      (NebezpecnaLatka (jmeno „Acefát“) (UNKod 3018)
            (CASCislo „30516-19-1“) (ESCislo „250-241-2“)
            (IndexoveCislo „015-079-00-7“)
      )
      (NebezpecnaLatka (jmeno „Acetaldehyd“) (UNKod 1089)
            (CASCislo „75-07-0“) (ESCislo „200-836-8“)
            (IndexoveCislo „605-003-00-6“)
      )
```

)

Při specifikaci tedy nejprve odkážeme jméno šablony, na kterou se odkazujeme (v našem případě NebezpecnaLatka), a potom vypisujeme názvy atributů a k nim přináležející hodnoty.

Poznámka: zarovnání pomocí entrů a použití tabulátorů je pouze pro lepší čitelnost souboru. Teoreticky mi nic nebrání, abychom všechny tyto definice zapsali na jediný řádek. Orientace v takovém prostředí a případné dohledávání chyb by se samozřejmě velmi ztížilo.

Při manipulaci s fakty můžeme používat řadu dalších příkazů:
(ppdeffacts jmeno_faktu) - zobrazení obsah daného faktu
(reset) - resetování báze faktů
(facts) - vypíše seznam všech faktů na obrazovku, tedy napříč všemi definicemi deffacts.
(undeffacts jmeno_faktu) - odstranění definice daného faktu
(list-deffacts) - vypíše seznam všech definic faktů

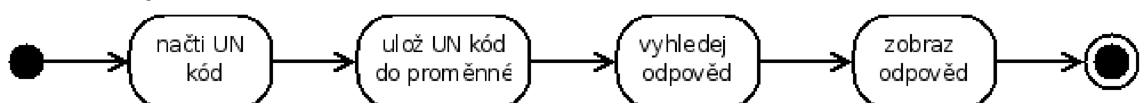
3.6 Pravidla

Pravidla

Pravidla obvykle formulujeme ve formě předpoklad -> důsledek (příčina -> následek). Od pravidel očekáváme, že provedou analýzu zadaných údajů (faktů problému) a dodají odpověď. Tomuto postupu (směru odvozování předpoklad -> důsledek) říkáme dopředné řetězení.

Z hlediska konstrukce jsou pravidla složitější neboť musí v sobě zahrnovat prostor pro otázky – zjišťování údajů od uživatele, a zároveň obsahuje logiku pro zpracování těchto údajů, jejich porovnání s fakty a poskytnutí odpovědi.

Zkusme si nadefinovat jednoduché pravidlo pro identifikaci látky podle UN kódu. UN kód v reálu sice není unikátní identifikátor nebezpečné látky, ale pro naše účely jej za unikátní identifikátor považovat přesto budeme. Proces identifikace látky, který pravidel ošetříme je zobrazen na obr. 29.



Obr. 29: Identifikace nebezpečné látky

Proces identifikace je záměrně zjednodušen, pomíjíme možnost, že UN kód není k dispozici a spoustu dalších věcí, takže nám vznikl lineární proces identifikace.

Podívejme se na první krok tohoto procesu – načtení UN kódu. Pro tuto situaci totiž budeme už potřebovat nadefinovat pravidlo. Pro definici pravidla používáme příkaz defrule.

Předtím, ale vytvoříme ještě jednu sadu faktů, která nám bude kontrolovat jaké otázky se mají načíst.

```
(deffacts   BehProgramu   „Pomocná   fakta,   která   ovlivní,
které otázky/(pravidla se spustí a které ne"
     (otazka-na-unkod 1)
     ; případné další otázky
)
```

Pravidlo pro naši situaci bude vypadat následovně:

```
(defrule NactiUNKod „provede načteni UN kódu"
     ?f <- (otazka-na-unkod 1)
=>
     (printout t crlf „Zadejte UN kod: ")
     (bind ?odp_un (read))
     (assert (UzivUNKod ?odp_un))
     (retract ?f)
     (assert (otazka-na-unkod 2))
)
```

Co nám pravidlo říká? První řádek pravidla začínající ?f provede vyhodnocení údajů v závorce. Jinými slovy systém zkontroluje zda otazka-na-unkod = 1. Pokud pravidlo spouštíme poprvé pak tomu tak skutečně je, protože jsme to tak naspecifikovali ve faktech o pár řádků výše.

Znak => nám odděluje předpoklady od důsledků. Tedy v našem případě, pokud otazka-na-unkod = 1 pak se provedou činnosti specifikované za tímto znakem. Jak vidíte v důsledkové části pravidla máme 4 nové příkazy, podívejme se co znamenají.

Printout vytiskne textový řetězec v uvozovkách na zadaném umístnění, parametr t specifikuje, že tímto místem bude terminál (příkazová řádka). Tento příkaz používáme v okamžiku, kdy je potřeba uživateli něco sdělit. CRLF je substitut za znak konce řádku. Stisknutí enter v tomto případě nestačí, protože enter je v pravidlech považován za tzv. bílý znak a jako takový je ignorován.

Příkaz bind provede načtení hodnoty do proměnné. Proměnné v CLIPS se poznají tak, že začínají znakem "?". V našem případě do proměnné ?odp_un načítáme pomocí funkce read vstup uživatele z klávesnice. Proměnnou jsme si mohli pojmenovat libovolně, je zde pouze omezení, že jméno proměnné musí začínat písmenem.

Příkazem assert provedeme přiřazení hodnoty z proměnné ?odp_un do naší báze znalostí. Retract umožňuje odstranit určitý fakt. V našem případě odstraníme fakt že otázka na un kód = 1, protože ji přiřadíme hodnotu 2. Tímto způsobem zajistíme, že dané pravidlo se již při vyhodnocování nespustí.

Ke spuštění pravidel používáme obvykle 2 příkazy a to (reset) a (run). Příkaz reset, resetuje bázi znalostí na základě původních definic.

Resetování je nutné pokud báze znalostí byla modifikována aplikací pravidel. Absence resetu může významně ovlivnit způsob chování expertního systému.

Příkaz run provede spuštění pravidel nadefinovaných v systému. Pozor tato pravidla mohou ovlivňovat bázi znalostí. Třeba v našem případě bude vytvořen nový fakt, který bude obsahovat zadaný UN kód. Existenci tohoto nového faktu můžete jednoduše ověřit pomocí příkazu (facts) který vypíše všechna definovaná fakta.

Pokud jste náš společný výtvor zkusili spustit, CLIPS se Vás zeptal na UN kód a to je všechno, co systém dělal. Přiznejme si, je to málo. Z tohoto důvodu navrhneme ještě jedno pravidlo, které zpětně identifikuje látku podle UN kódu.

(defrule HledejUNKod „Provede vyhledání UN kódu“

```
      (otazka-na-unkod 2)
      (UzivUNKod ?un)
      (NebezpecnaLatka    (UNKod ?UNKod)
                          (jmeno ?jmeno))
      (test (eq ?un ?UNKod))
=>
      (printout t crlf „Identifikována látka: ‟ ?jmeno
crlf)
)
```

Jak si zajisté všimnete, tak předpokladová část pravidla je poněkud složitější. Prvním předpokladem (otazka-na-unkod 2) je, že proběhlo předcházející pravidlo, kterým jsme načetli od uživatele UN kód.

Další dva předpoklady:

```
      (UzivUNKod ?un)
      (NebezpecnaLatka    (UNKod ?UNKod)
                          (jmeno ?jmeno))
```

se starají o tzv. porovnání se vzorem. Toto porovnání probíhá tak, že se procházejí všechna dostupná fakta zavedená v bázi znalostí a porovnají se se vzorem. Hledáme výsledek předchozího pravidla, tedy uživatelem zadaný UN kód, tento výsledek se automaticky uloží do proměnné ?un.

Podobně to funguje s druhým předpokladem, ten načte a do paměti uloží všechny UN kódy a jména nebezpečných látek obsažených v bázi znalostí. Z seznamu těchto látek je však nutno odstranit látky, u kterých neodpovídá UN kód načtený od uživatele. O to se stará poslední předpoklad.

```
    (test (eq ?un ?UNKod))
```

Tento předpoklad provede srovnání UN kódu načteného od uživatele (?un) a načteného z báze znalostí (?UNKod).

Důsledková část pravidla pouze zobrazí název hledané(-ých) látek. Pokud předpokladům bude vyhovovat více látek provede se vypsání jména pro všechny tyto látky bez nutnosti něco dalšího doplňovat.

3.7 Závěr

V této kapitole jsme si přiblížili postup plnění expertního systému.

Všimněte si, že kód šablon a případně faktů generovaný Protege2000 s pluginem CLIPSTab vlastně vůbec neodpovídá šablonám a faktům, které jsme ručně tvořili ve zbývajících podkapitolách. Je tomu tak proto, že Protege2000 používá pokročilejší objektovou notaci znalostní báze, zatímco my jsme použili jednodušší zápis, který je snadněji pochopitelný a je také kratší.

Expertní systém CLIPS je nesmírně složitý a v omezeném prostoru, který v tomto předmětu máme k dispozici, jsme se dotkli pouze jeho povrchu.

Příklad, který jsme společně vytvořili (s trochou štěstí) je pro jistotu dostupný i z mých domácích stránek [15] ve vyzkoušené a odladěné verzi. Takže pokud se Vám nepovedlo naplnit expertní systém vlastními silami nebo se Vám prostě nechtělo, můžete si bázi znalostí stáhnout a vyzkoušet.

Kontrolní otázky

1. Jaký je rozdíl mezi šablonou, faktem a pravidlem?
2. Co je to ontologie systému?
3. Který diagram UML je převoditelný do expertního systému?
4. Jaký je rozdíl mezi předpokladem a důsledkem v pravidlech báze znalostí?

Samostatné úkoly

1. Rozšiřte pravidla vzorového expertního systému na hledání názvu nebezpečné látky tak, aby zobrazoval také CAS, ES a indexové číslo látky.
2. Navrhněte expertní systém pro vyhledávání informací o strudentech na základě nějakého identifikátoru (třeba rodné číslo, číslo studenta apod.).

4 Neuronové sítě

Průvodce studiem

V této kapitole se seznámíme se způsobem jakým funguje lidský mozek a jak tyto znalosti můžeme využít pro konstrukci umělých neuronových sítí. Podíváme se na několik druhů neuronových sítí a na problémy, které pomocí nich můžeme řešit.

Po prostudování této kapitoly budete vědět

- jakým způsobem fungují proces učení a vybavování si faktů
- jaké problémy jsou řešitelné pomocí neuronových sítí
- jak tyto problémy pomocí neuronových sítí řešit

Čas pro studium

Pro prostudování této kapitoly budete potřebovat minimálně 3 hodiny.

4.1 Biologické základy neuronových sítí

V umělých neuronových sítích se snažíme napodobit způsob fungování mozků vyšších živočichů. Podívejme se tedy jakým způsobem tyto mozky fungují.

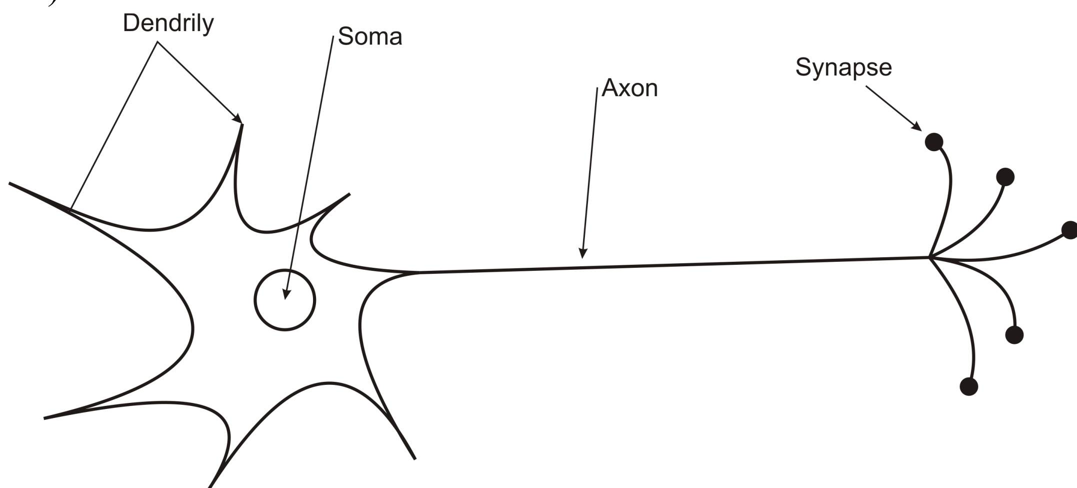
Neuron

Neuron

Základním stavebním kamenem mozku je neuron [16]. Velikost neuronu se v lidském mozku pohybuje od 4 – 100 micronů (0,004 – 0,1 mm), podle druhu neuronu. U různých živočichů je přitom velikost neuronů různá. Obecně platí, že čím je živočich výše v evolučním stromu tím má větší množství menších neuronů.

Lidský mozek přitom obsahuje okolo 10^{11}neuronů. Tento počet se během života mění v závislosti na věku, ale také požadované mozkové aktivitě, fyzickém stavu apod.

Podívejme se jakým způsobem neuron fyzicky vypadá (viz. obr. 30).



Obr. 30: Vzhled neuronu

Neuron tvoří jádro (soma), tělo, kterého vyčnívají výběžky tzv. dendrily. Neuron obsahuje také jeden obzvláště dlouhý výběžek, který nazýváme axon. Axon slouží k přemostění vzdálenosti mezi jednotlivými neurony. Axon je zakončen synapsemi, které se připojují na dendrily dalších neuronu.

Synapse obsahuje mechanismus prahu, který uvolňuje chemické látky v závislosti na tom, jestli elektrický impuls kterým se šíří informace má být posílen nebo naopak potlačen.

V případě posílení hovoříme o excitátorech a v případě oslabení hovoříme o inhibitorech. O tom, zda signál má být utlumen (inhibován) nebo posílen (excitován) rozhoduje síla signálu. Pokud síla signálu překročí určitou prahovou hodnotu pak je posílen (excitován) a maximální míru a je umožněno jeho další šíření přes synapsi do připojeného neuronu a jeho prostřednictvím dále, kde je signál ovšem podrobován stejnému testu.

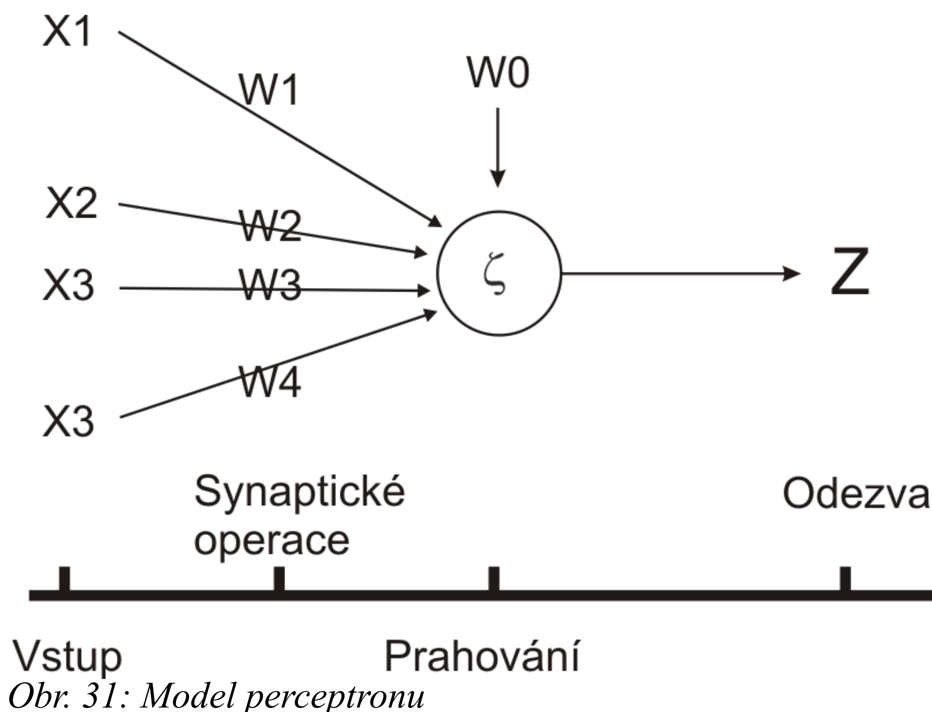
Proces učení spočívá právě ve správném nastavení těchto prahů, aby v mozku byly excitovány ve správný čas správné vzorce neuronů, což způsobí že si nějakou informaci vybavíme. Práh tedy je různý pro různé neurony a v souvislosti s učením (nebo naopak zapomínáním) se mění.

4.2 Perceptron

Perceptron

Jedním z prvních modelů umělých neuronů, který se velmi podrobně snaží napodobit funkčnost přirozeného neuronu byl perceptron. Tento model vznikl v 60. letech minulého století a z počátku vypadal, že se bude jednat o univerzální model schopný řešit prakticky všechny úlohy.

Později se ukázalo, že model Perceptronu, zdaleka není vhodný pro řešení všech typů úloh. Podívejme se však jak tento model vypadá (viz. obr. 31).



Obr. 31: Model perceptronu

Vstupy x se nám sbíhají do neuronu. U reálného neuronu je toto napojení realizováno prostřednictvím synapsí, které úroveň signálu posílí nebo utlumí. V umělém neuronu stejnou úlohu plní váhy označované w. Váhy umožňují ukládat do neuronové sítě znalosti. Procesem učení modifikujeme právě tyto váhy (proces učení samotný si necháme na později).

Vážené vstupy jsou agregovány a podrobeny procesu prahování – tedy rozhodnutí, zda neuron bude excitován nebo ne.

Podívejme se na matematické vyjádření celé operace. Nejprve synaptické operace (1).

$$y_i = x_i * w_i \tag{1}$$

Celkový potenciál neuronu

Na takto upravené vstupy použijeme práh neuronu w_0 a agregujeme, čímž získáme celkový potenciál neuronu (2).

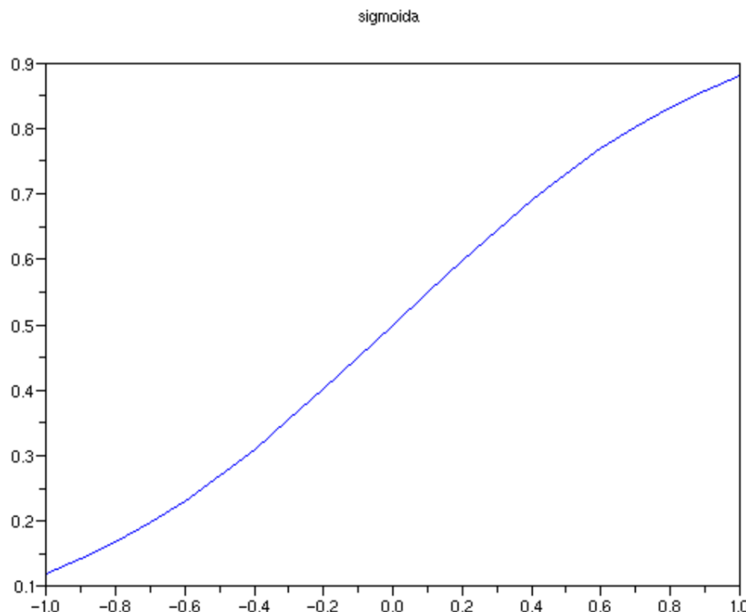
$$\zeta = \sum_{i=1}^{n} y_i - w_0 = \sum_{i=1}^{n} x_i * w_i - w_0 \tag{2}$$

Přenosová funkce

Výsledná odezva neuronu bude záviset na prahování. Tedy celkový potenciál bude dosazen do předepsané přenosové funkce, která rozhodne o výši odezvy z. Nejčastěji se pro tento účel používá přenosová funkce nazývaná sigmoida.

$$z = \frac{1}{1 + e^{-\lambda\zeta}} \tag{3}$$

Koeficient lambda ve vzorci je koeficientem strnulosti nebo strmosti, chcete-li, a rychlost s jakou přechází křivka mezi mezními hodnotami 0 – 1. Podívejme se na sigmoidu nasimulovanou pro lambda = 2 na obr. 32.

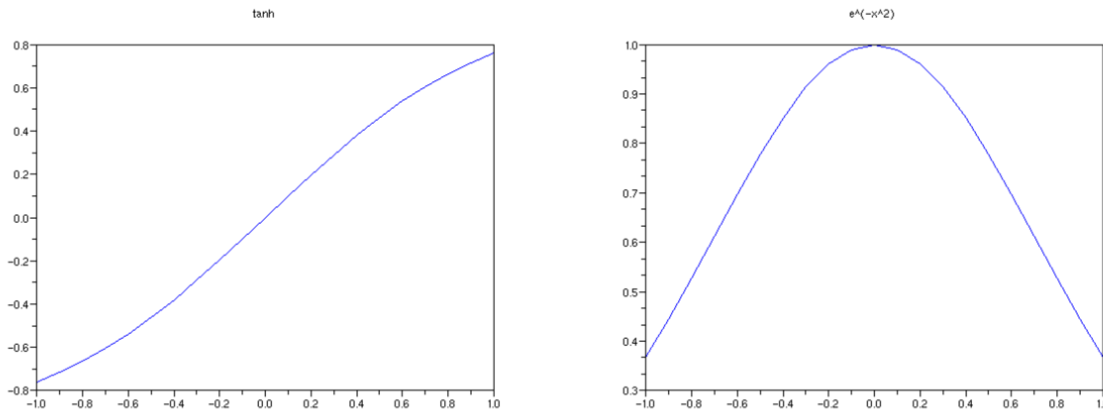


Obr. 32: Přenosová funkce - sigmoida

Kromě této přenosové funkce lze využít také jiné funkce, třeba ty ve vzorcích (4) a (5). Grafy těchto funkcí si můžete prohlédnout na obr. 33.

$$z = \tanh(\zeta) \tag{4}$$

$$z = e^{-\zeta^2} \tag{5}$$



Obr. 33: Další přechodové funkce podle vzorců (4) a (5)

Nyní již víme jakým způsobem funguje neuron samostatně, bohužel jeden neuron je málo. Abychom byli schopni z neuronové sítě získat nějaké údaje, je nutné, aby tato síť byla složena z většího množství neuronů.

Vrstvy neuronové sítě

Navíc je potřeba tyto neurony vrstvit. V rámci neuronových sítí rozlišujeme tři druhy vrstev:

1) vstupní
2) výstupní
3) pracovní (skryté)

Vstupní vrstva slouží pro zachycení vstupních údajů nutných pro zjištění výsledku. Výstupní vrstva slouží pro zachycení výsledků práce neuronové sítě (výsledek transformace vstupů na výstupy).

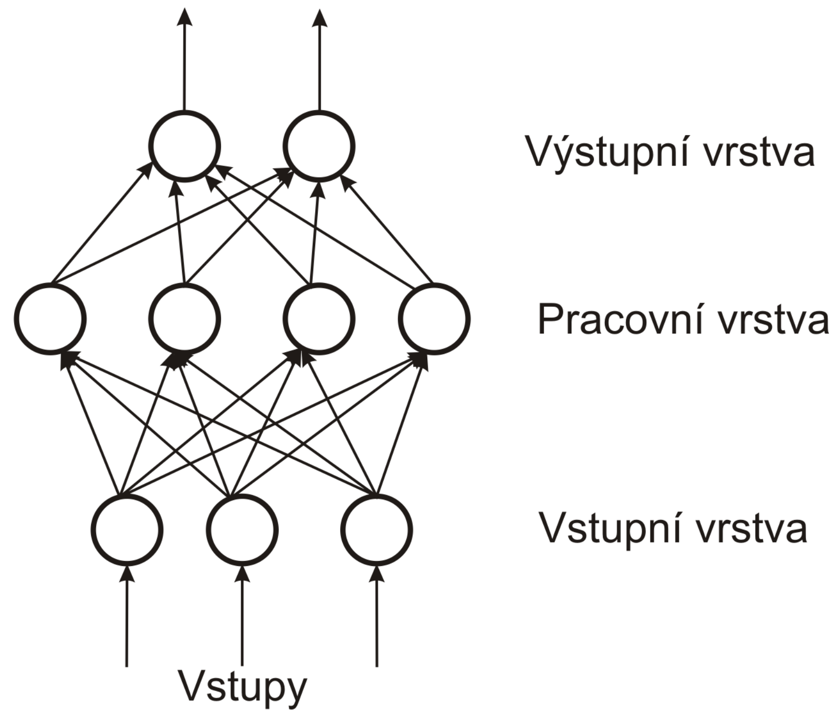
V síti může být (nemusí být) obsažena jedna nebo více tzv. pracovních vrstev. Tyto vrstvy pracují skrytě. Skrytostí rozumějte to, že uživatel neuronové sítě s nimi na rozdíl od vstupní a výstupní vrstvy nepřijde do přímého styku, pouze těží nepřímo z jejich přítomnosti tak že má k dispozici kvalitní výsledky práce této vrstvy (tedy za předpokladu úspěšné adaptace sítě).

Zvolenému systému vrstev a jejich propojení souhrnně říkáme topologie neuronové sítě.

Schématicky si takovou síť můžeme představit následovně (obr. 34).

Všimněte si, že každý neuron nižší vrstvy je přímo spojen s každým neuronem vyšší vrstvy. O tom, které vazby mezi neurony budou ve finále aktivní rozhodne hodnota vah, která podléhá procesu učení (adaptace neuronové sítě).

Nyní se podívejme na proces adaptace takové sítě. Sítě perceptronů



Obr. 34: Schéma vrstvené neuronové sítě

jsou obvykle adaptovány tzv. učením s učitelem. Tomuto procesu rozumějte tak, že neuronové síti jsou předkládány vstupy a správné výstupy těmto vstupům odpovídající. Neuronová síť tak získává zpětnou vazbu o tom, jak je „dobrá“.

Backpropagation

Jednou z nejpoužívanějších metod adaptace neuronové sítě pro takové případy je metoda backpropagation (zpětné šíření chyby). Podívejme se na mechanismus, jakým probíhá v takových případech učení.

Trénovací množina

Proces učení probíhá s učitelem, tedy specifikujeme tzv. trénovací množinu obsahující vstupní hodnoty x a očekávané výstupní d. Matematicky můžeme zapsat následovně:

$$T\{x[k],d[k]\}_{k=1}^{N} \tag{6}$$

Vstupu x, který zadáváme bude odpovídat skutečný výstup neuronové sítě z, který se nepochybně bude lišit od očekávaného výstupu sítě. Rozdílem těchto dvou hodnot můžeme vyčíslit chybu. Chyba může být obojího druhu – tedy očekávaný výstup může být větší nebo menší než výstup skutečný, z tohoto důvodu obvykle počítáme se čtverci chyb. Podle těchto pravidel si můžeme matematicky naspecifikovat chybovou funkce E(k).

$$E(k)=\frac{\sum_{j=1}^{m}[y_j(k)-d_j(k)]^2}{2} \tag{7}$$

Na počátku adaptačního procesu většinou stanovujeme požadovanou cílovou chybu v procentech. Adaptační proces je ukončen, pokud chyba pro všechny vzory poklesne pod tuto požadovanou úroveň.

Váhy aktualizujeme následujícím způsobem:

Váhy

$$w_{ij}(t+1)=w_{ij}(t)+\Delta w_{ij}(t) \quad (8)$$

Aktualizace vah probíhá iteračním způsobem. Tedy hodnota váhy pro další krok je vypočtena na základě hodnoty váhy v kroku předchozím a změně váhy Δw. Tuto změnu vah, která bude z hlediska adaptace sítě kritická můžeme vypočítat následovně:

$$\Delta w_{ij}^{0}(t)=\eta\,\delta_{i}^{0}(t)\,z_{j}^{1}(t)+\mu\,\Delta w_{ij}^{0}(t-1) \quad (9)$$

Ve vzorci (9) jsou obsaženy pro nás dosud neznámé symboly. Podívejme se na to, co znamenají.

η parametr učení (konstanta)

μ momentum (konstanta)

δ chyba neuronu v dané vrstvě

t časový parametr udávající epochu učení

ostatní symboly odpovídají symbolům vysvětlených v předchozím textu.

Vzorec (9) definuje samotnou podstatu metody backpropagation. Nejprve „dopředně" vypočteme výstup neuronové sítě z odpovídající předloženým vstupům x.

Pokud dosažené výsledky z neuronu neodpovídají očekávaným výstupům d, pak je to způsobeno tím, že výstupy neuronů z předchozí vrstvy jsou zpracovávány chybně (pomocí vah) a je nutno je změnit.

Horní index 0 a 1 ve vzorci (9) určuje vrstvu, která se zpracovává. Protože nasazujeme zpětný chod 0 odpovídá vyšší vrstvě než 1. Pokud bychom použili běžnou tří-vrstevnou síť, pak by 0 mohlo odpovídat výstupní vrstvě a 1 vrstvě pracovní.

4.3 Pracovní prostředí SciLab

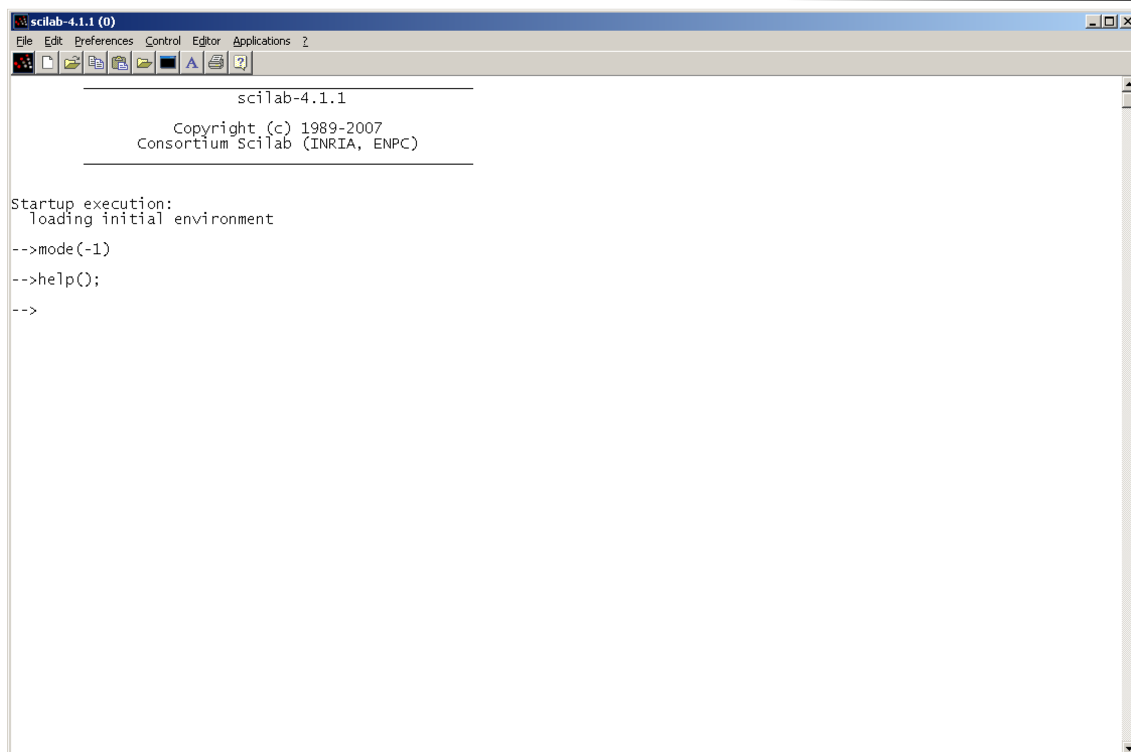
Pro simulace neuronových sítí budeme potřebovat nějaký nástroj prostředí. Pro svou univerzálnost jsem zvolil prostředí open-source nástroje SciLab [19] s rozšíření ANN Toolbox 0.4.2.

Na CD k tomuto předmětu máte k dispozici kromě obou těchto programů také upravenou verzi ANN Toolboxu, tak aby fungovala ve SciLab 4.1.1 (poslední stabilní verzi prostředí).

Po spuštění vypadá prostředí následovně:

V prostředí SciLab podobně jako v prostředí MatLab nebo Mathematica se pracuje z příkazové řádky. Práce probíhá buď v interaktivním režimu, kdy uživatel zadává příkazy a sleduje výstupy, které mu SciLab vrací nebo v dávkovém režimu, kdy si uživatel připraví dávku příkazů, které se mají vykonat a uživatel pak dostane k dispozici výsledek.

Podívejme se na některé příklady, abychom se seznámili se způsobem práce s tímto prostředí. Zkusme si vytvořit jednoduché grafy přechodových funkcí z předchozí kapitoly.



Obr. 35: Prostředí SciLab

Pro vykreslení sigmoidy použijeme následující skript:

```
clear;
x=(-1:0.1:1);
zeta = 1 ./ (1 + %e ^ (-2 * x));
plot(x, zeta);
title(„sigmoida");
```

Příkaz clear; provede výmaz veškerých předchozích výpočtů, proměnných, matic apod. Tímto příkazem začneme a zajistíme tak, že náš výpočet nebude zatížen z jakýmikoliv výpočty, které jsme mohli provádět předtím.

Řádek x = (-1:0.1:1) znamená, že potřebujeme získat vektor který dosadíme do rovnice sigmoidy. Samozřejmě bychom jej mohli vytvořit ručně (takovým způsobem postupujeme třeba u MS Excel), SciLab nám však poskytuje nástroje, které nám umožní nechat si tento vektor vygenerovat přímo v tomto prostředí.

Pro sigmoidu má smysl generovat hodnoty (pro účel vykreslení) v intervalu <-1; 1>, což jsou, jak si zajisté všimnete, obě krajní hodnoty výrazu. Hodnota 0.1 určuje po jakých krocích se má přechod udát. V našem případě po 0.1, což znamená, že získáme vektor s celkem 20-ti složkami {-1, -0.9, -0.8 …, 0.8, 0.9, 1}. Tento vektor se pro pozdější užití uloží do proměnné x.

Další řádek obsahuje vzorec samotné sigmoidy. Zajímavé na něm je pouze použití operátoru ./ místo běžného lomítka. Použití ./ je nutné,

protože za x budeme dosazovat vektor. Po použití běžného lomítka proběhne výpočet také, ale výsledek nebude korektní. Výraz typu 6/3, ale bude vypočítán správně, takže pozor!

%e říká SciLab, že se má použít zabudovaná konstanta eulerova čísla (přibližně 2.718).

Příkaz plot už pouze vykreslí výsledek naší práce a title umožní zapsat nadpis grafu.

Práce se SciLab je relativně intuitivní, na Internetu navíc (i v češtině) existuje celá řada materiálů použitelných pro podrobnější seznámení se s tímto zajímavým nástrojem.

Zkusme využít ANN Toolbox. Zkusíme nejprve nasimulovat jednoduchou funkci y = x. Použijeme tří vrstevnou neuronovou síť s topologií [1 5 1], tedy vstupní/výstupní 1 neuron, pracovní 5 neuronů.

Skript pro SciLab následuje:

```
rand('seed', 0);    //iniciace generátoru náhodných čísel
N = [1,5,1];   //topologie sítě
x = (0:0.1:1); //vstupní vektor
x2 = (0.05:0.1:1);//verifikační vektor
d = x;         //aplikace f-ce
d2 = x2;
lp = [2.5,0];  //parametry učení
W = an_FF_init(N);  //počáteční iniciace vah
T = 400;       //cílový počet iterací
W = ann_FF_Std_online(x,d,N,W,lp,T);    //adaptace sítě
vystup = ann_FF_run(x2,N,W);
plot(x2, vystup);   //vynes do grafu vypočtené hodnoty
plot(x2, x2);  //vynes do grafu předpokládané hodnoty
title(„předpověď vs. realita");
```

Interpretace tohoto skriptu není nijak obtížná. Připravili jsme si vstupní vektor x, kterému odpovídá výstupní vektor d = x. Dvojice x, d nám tvoří trénovací množinu. Pro účely přímého použití funkcí pro ovládání neuronových sítí je potřeba zajistit aby hodnoty vstupů i výstupů byly vždy v intervalu <0; 1>. Tedy vstupní vektor i výstupní vektor je potřeba normalizovat.

Výsledky neuronové sítě, protože jsou také normalizované je potřeba transformovat do přijatelné podoby. Jedná se o triviální úpravu, kterou tady nebudu rozebírat, měli byste být schopni tento problém vyřešit sami.

Kromě vstupu x si připravíme ještě verifikační množinu x2. Podobně jako u statistiky je výhodné dostupnou množinu vstupů rozdělit na několik částí (pokud je to počtem dostupných měření možné). Podle jedné části se tvoří model a pomocí druhé se tento model verifikuje.

Lp je vektor obsahující parametry učení. První z těchto parametrů je parametr učení dle vzorce (9). Druhým je nastavení přijatelné chyby. Nastavení úrovně chyby na 0, není úplně obvyklé, ale v našem pokusném případě si to můžeme dovolit.

Pomocí funkce ann_FF_init(N) provede iniciaci všech vah v síti náhodnými malými hodnotami. Iniciace je povinným krokem adaptace neuronové sítě. Této funkci předkládáme vektor topologie sítě.

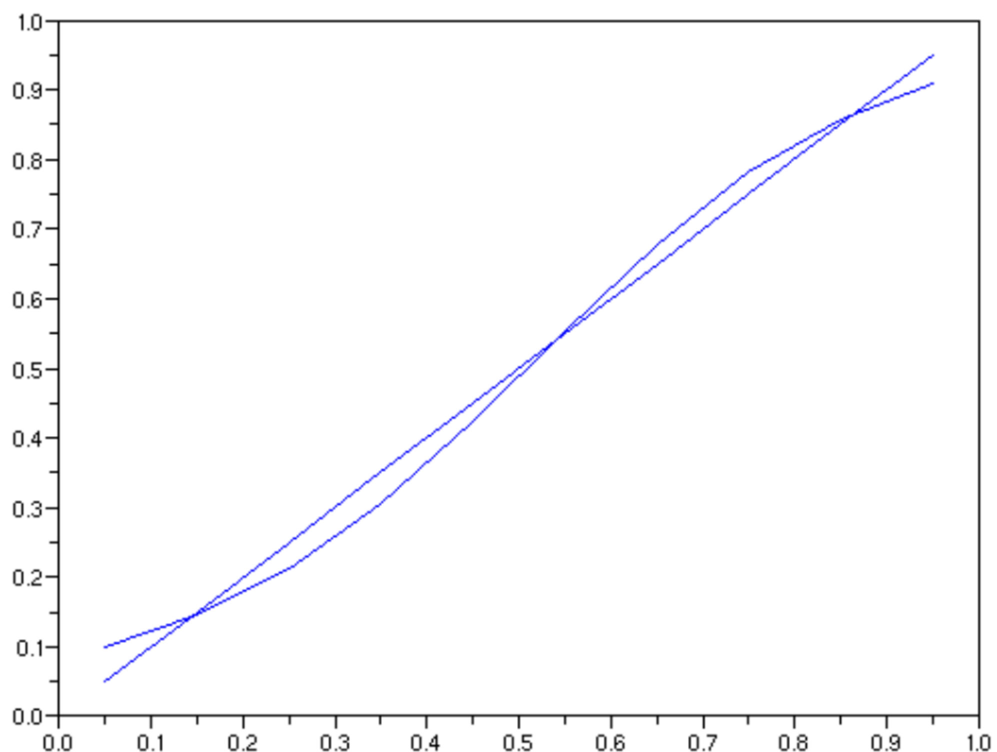
Parametr T představuje maximální počet iterací. Nastavením tohoto čísla (což je také povinné) ovlivňujeme maximální délku procesu adaptace. Záleží přitom jak přesných výsledků potřebujeme dosáhnout – čím vyšší počet iterací, tím vyšší naděje, že výsledná chyba se bude blížit požadované úrovni.

Tohoto počtu nemusí být dosaženo, pokud v rámci procesu učení poklesne chyba pod úroveň nastavenou parametrem lp. 400 iterací je mimochodem relativně málo. Složitější sítě mohou vyžadovat klidně desetitisíce nebo miliony iterací.

Před experimentováním s procesem adaptace není možné určit jaký počet iterací nebo nastavení parametrů učení lp bude optimální.

Síť samotnou adaptujeme pomocí funkce ann_FF_Std_online s parametry trénovací množiny x a d, určením topologie sítě N, váhami W, parametry učení lp a maximálním počtem iterací T.

Funkce ann_FF_run provede skórování množiny, v našem příkladě verifikační množiny x2. Výsledek práce neuronové sítě spolu s vynesením referenční funkce je zobrazeno na obr. 36.



Obr. 36: Neuronová síť vs. referenční funkce y = x

4.4 Poznámky k modelování pomocí neuronových sítí

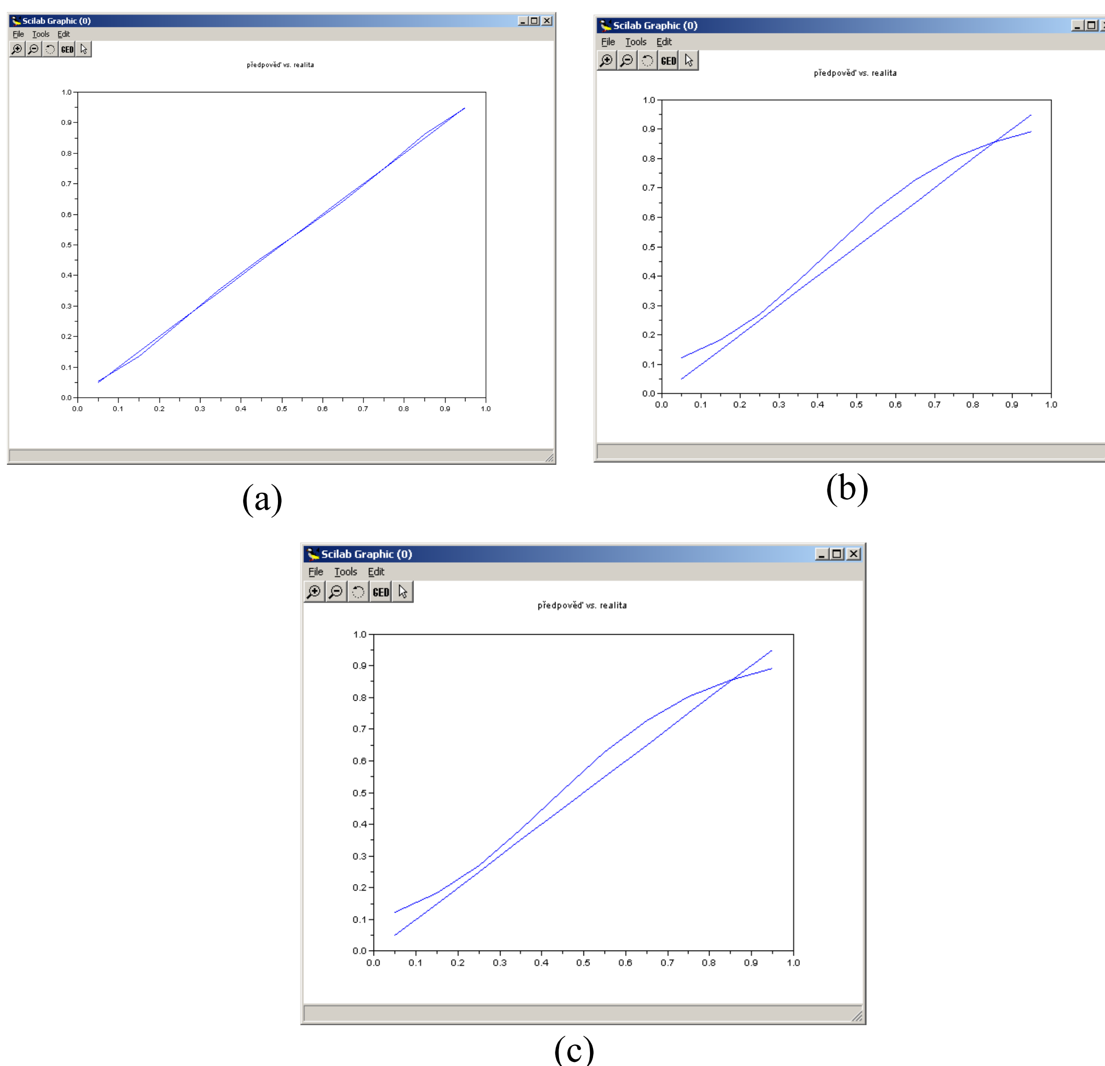
Jak vidíte, hodnoty předpovězené neuronovou sítí oscilují okolo správných hodnot. O tom, jak moc se budu blížit rozhodne nastavení parametrů modelu sítě. Zkusme s modelem ještě chvíli experimentovat.

Iterace vs. chyba

První věc, kterou můžeme jednoduše vyzkoušet je navýšení počtu iterací. Abychom viděli rozdíl zkusme navýšit tento počet na 10 000 (obr. 37a). Kvalita predikce se výrazně zlepšila, ovšem na úkor času nutného pro adaptaci sítě (na mém domácím počítači z 20s na 7min).

Zkusme zvýšit maximálně přípustnou chybu na 10% při 400 iteracích (výsledek na obr. 37b). A konečně zkusme použít 10 000 iterací pro chybu 10% (obr. 37c).

Jak je vidno, zvýšení počtu iterací s nulovou cílovou chybou vede



Obr. 37: Vliv změny parametrů na výsledek neuronové sítě

ke zlepšení prediktivních schopností sítě. Při chybě 10%, se však ani s navýšením počtu iterací výsledek nezlepšil. Je tedy vždy nutné najít určitý kompromis mezi požadovanou přesností a dobou, kterou jsme ochotni věnovat adaptaci sítě.

Prediktivní schopnosti neuronových sítí také nejsou všemocné,

podléhají podobným omezením jako statistické modely (např. regresní modely). Tyto modely jsou obvykle schopny věrohodně zachycovat trendy v oblasti zachycené daty použitými pro tvorbu modelu a ztrácejí věrohodnost, pokud model má predikovat hodnoty mimo interval těmito trénovacími daty pokrytý.

Podobně je tomu tak i u neuronových sítí. Z tohoto důvodu je nezbytně nutné volit trénovací množinu tak, aby co nejlépe odrážela vlastnosti hledané závislosti pokud možno v celé její šíři.

Přeučenost sítě

Dalším problémem s modelováním neuronových sítí je tzv. přeučenost sítě. Jedná se o chvíli, kdy je pro danou situaci zvolena příliš vysoká přesnost. Výsledkem může být, že neuronová sít se skutečně s požadovanou přesností adaptuje a bude přesně zobrazovat hodnoty obsažené v trénovací množině, avšak při verifikaci modelu na další skupině dat selže.

Problém přeučenosti je způsoben určitou chybou, kterou vnášíme do našeho vidění modelované situace již v okamžiku samotného měření. Přístroje měří pouze s určitou přesností – výsledek měření je tak okamžitě „na vstupu“ zatížen náhodnou chybou. Do měření mohou vstupovat i jiné systémové vlivy, které zanášejí do datového souboru další chyby.

V takové chvíli je kontraproduktivní se snažit o dosažení přehnané přesnosti, proto obvykle nastavujeme nějako malou nenulovou požadovanou maximální chybu (třeba 3%). Optimální nastavení maxima požadované chyby se liší případ od případu a určujeme jej spíše intuitivně a v rámci experimentování s modelem upravujeme na základě výsledků, kterých model dosahuje.

V našem vzorovém příkladě k problému s přeučením nedošlo, protože jsme modelovali obecnou matematickou rovnici, která chybou zatížena nebyla, proto volba nulové maximální chyby byla volbou správnou, navyšování počtu iterací může v takovém případě přinést pouze vetší přesnost adaptovaného modelu.

V praxi se optimální cílová chyba liší případ od případu. Vždy bychom se měli zamyslet nad některými faktory:

- přesnost měření (cílová chyba by měla být >= něž je chyba měření)
- kvalita dostupné trénovací množiny (čím větší, tím lepší, tím menší může být cílová chyba)
- doba nutná pro adaptaci sítě (čím menší chyba, tím delší adptace sítě)
- jaké máme přímé požadavky na přesnost.

4.6 Problémy řešitelné neuronovými sítěmi

Neuronové sítě jako metodu modelování volíme tam, kde máme k dispozici množinu vstupních hodnot a jim odpovídajících hodnot výstupních a máme podezření, že mezi těmito hodnotami existuje jakási dosud nevyčíslená závislost. Při řešení takových problémů se využívá toho, že neuronová síť funguje jako černá skříňka, která přizpůsobí své

„neviditelné“ vnitřní chování předkládaným vstupům a výstupům.

Neuronové sítě nemusí působit pouze jako náhrada běžně používaných statistických modelů, ale jako jejich doplněk, kontrolní výpočet, chcete-li. rozšířené dataminingové nástroje jako je SPSS nebo SAS Dataminer, jsou k souběžnému použití statistických modelů i modelů na bázi umělé inteligence dobře přizpůsobeny a ve velkých podnicích jsou rutinně nasazovány.

Zamysleme se nad způsobem modelování různých typů problémů.

V případě modelování známých matematických závislostí je situace jednoduchá. všechny proměnné z pravé části rovnice použijeme jako vstupní údaje neuronové sítě. Počet neuronů bude totožný s počtem těchto proměnných. Ve výstupní vrstvě bude neuron reprezentující závislou veličinu. Známé matematické závislosti je obvykle výhodnější vypočítávat než je řešit prostřednictvím neuronových sítí (je to rychlejší a přesnější).

Zajímavější je použití neuronových sítí pro modelování časových řad. Kromě přímé závislosti na datu (třeba) je možné vnášet do modelu parametry související se sezónností jako je třeba roční období, měsíc a podobně. Neuronová síť sama v rámci procesu adaptace použije nebo naopak nepoužije neurony zabezpečující tuto funkčnost (nastavením vah).

Neuronové sítě je možno s úspěchem použít pro rozlišování obrazců (jako jsou třeba písmena). Posuzovaný obrazec je nutno rozdělit do menších oblastí čtvercového tvaru. Jinými slovy pokryjeme obrazec sítí, kde každé políčko sítě bude reprezentováno jedním neuronem vstupní vrstvy. Podle toho, zda se v políčku objeví barva nebo ne nabývá vstupní signál hodnoty 1 (barva přítomna) a 0 (barva nepřítomna). Takové sítě logicky musí mít stejný počet neuronů vstupní i výstupní vrstvy.

Kontrolní otázky

1. Jaké vrstvy se používají v neuronových sítích?
2. Jaký parametr neuronových sítí a jakým způsobem se mění během adaptace neuronové sítě?
3. Vysvětlete proč se pro ověření činnosti neuronové sítě používá jiná sada dat než pro adaptaci?
4. Co je to přechodová funkce?
5. Vysvětlete (alespoň přibližně) fungování biologického neuronu.

Úkoly k zamyšlení

1. Zkuste zrealizovat jednoduché OCR pomocí neuronových sítí pro rozlišování čísel.
2. Zkuste vytvořit neuronovou síť zachycující vztah: $y = x + 2x^2 + 15$.

Pro realizaci obou úkolů použijte prostředí SciLab.

5 Buněčné automaty

Průvodce studiem

V této kapitole se podíváme na možnosti použití umělého života pro modelování některých dějů. V rámci společného studia budeme experimentovat se jednoduchým programovacím jazykem Logo, pomocí kterého, lze jednoduchým způsobem tyto automaty ovládat.

Po prostudování této kapitoly budete vědět

- co jsou to buněčné automaty
- k modelování jakých problémů je možné je využít
- jakým způsobem probíhá programování automatu

Čas pro studium

Na prostudování této kapitoly budete potřebovat dvě hodiny.

5.1 Počátky buněčných automatů – Conwayova hra života

Buněčné automaty

Buněčné automaty (celular automata) jsou jednou z aplikací umělého života. Jedná se o dynamické modely, které jsou diskrétní v čase, tedy průběh simulace probíhá po samostatných, v čase oddělených iteracích – neprobíhá v plynule.

Samotné univerzum modelu je rozděleno na samostatné prostory – buňky, které si jednak žijí vlastním životem v rámci nastavených pravidel, jednak působí na buňky ve svém okolí (opět podle určitých pravidel). Iterace probíhá na základní lokální přechodové funkce.

Lokální přechodovou funkcí rozumíme pravidlo/předpis, který je aplikuje zároveň na všechny buňky modelu. Výsledek funkce (nový stav buňky) se vypočte na základě stavu předchozí iterace buňky a stavu buněk v jejím okolí.

Neumanův buněčný automat

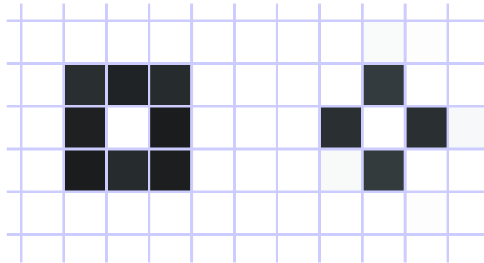
Základy této oblasti umělé inteligence položil John von Neuman v 50. letech minulého století. Když vytvořil s použitím tužky a papíru první buněčný automat, který byl schopen samoreplikace. Tento automat byl sestaven z přibližně 200 000 buněk, přitom každá z těchto buněk mohla mít 29 stavů. Ve své době se jednalo o unikátní teoretické dílo, které bylo vzhledem k neexistenci počítačů, tak jak je známe dnes, neověřitelné. Práci na tomto unikátním díle přerušila v roce 1957 von Neumanova smrt.

Conwayova hra života

Složitost celé práce zapříčinila, že od prvních teoretických pokusů v letech 60. k praktičtějším pokusům uběhla poměrně dlouhá doba. Další podstatný průlom nastal až v roce 1970, kdy matematik John Horton Conway zveřejnil koncepci buněčného automatu, který dnes nazýváme Conwayova hra života.

Tento buněčný automat je mnohem jednodušší než původní von Neumanův. Počet stavů je omezen pouze na dva. Takovou konfiguraci můžeme interpretovat biologicky prostě tak že buňka na daném umístnění

žije a nebo nežije. Univerzum, ve kterém probíhá simulace také nemusí obsahovat sta tisíce buněk, stačí desítky až stovky buněk. Tento druh automatu je složitější pouze v jediném parametru a to uvažováním tzv. úplného okolí buněk (tvoří 8 buněk) oproti von Noumanovu okolí (tvoří 4 buňky). Rozdíl mezi oběma okolími je jasně patrný z obr. 38.



Obr. 38: Úplné okolí (vlevo) vs. von Neumanovo okolí (vpravo)

V Conwayově hře života existují tři možné výsledky iterace:
1) narození buňky – buňka se zrodí pokud v jejím okolí jsou právě tři žijící buňky,
2) přežití buňky – buňka přežije pokud v jejím okolí jsou 2 – 3 žijící buňky,
3) v ostatních případech buňka zahyne nebo se nezrodí.

Při experimentování s modelem Conway zjistil, že výsledky simulací je možné zařadit do jedné ze čtyř skupin:
1) totální zánik – po uplynutí několika generací se celá plocha úplně vyčistí,
2) vytvoření stabilního, v dalších generacích neměnného obrazce,
3) cyklicky se opakující obrazce,
4) cyklicky se opakující obrazce s posunem (stejný tvar ale posunutí oproti původnímu umístnění).

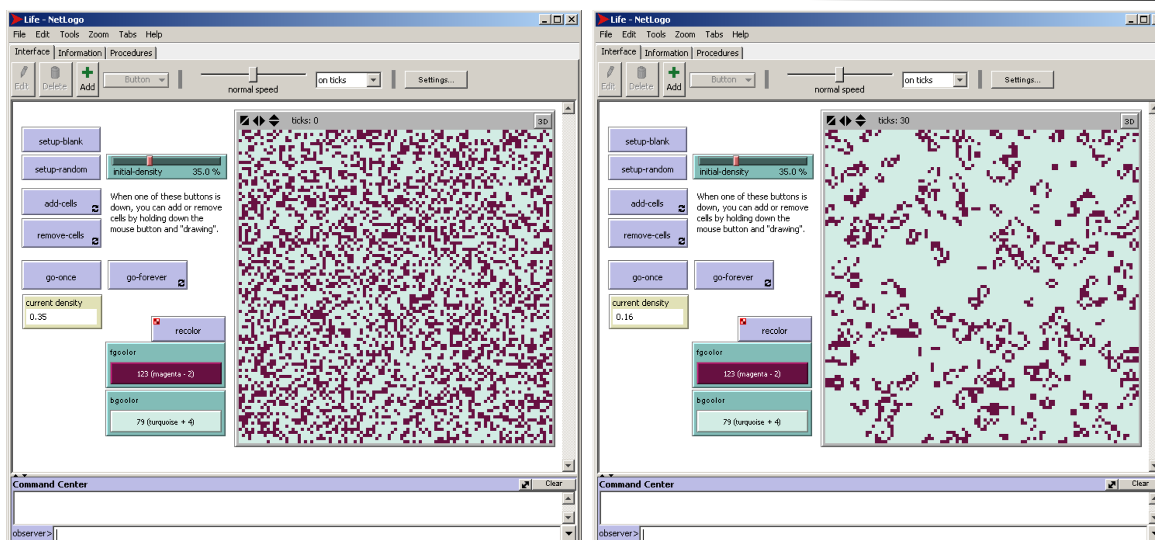
Dost bylo teorie – podívejme se na vzhled Conwayovy hry života. Pro tuto simulaci jsem zvolil prostředí NetLogo [20], které je volně dostupné ke stažení i s řadou různých modelů včetně Conwayovy hry života.

Tento model v BetLogu naleznete ve File -> Models Library -> Computer Science -> Cellular Automata -> Life.

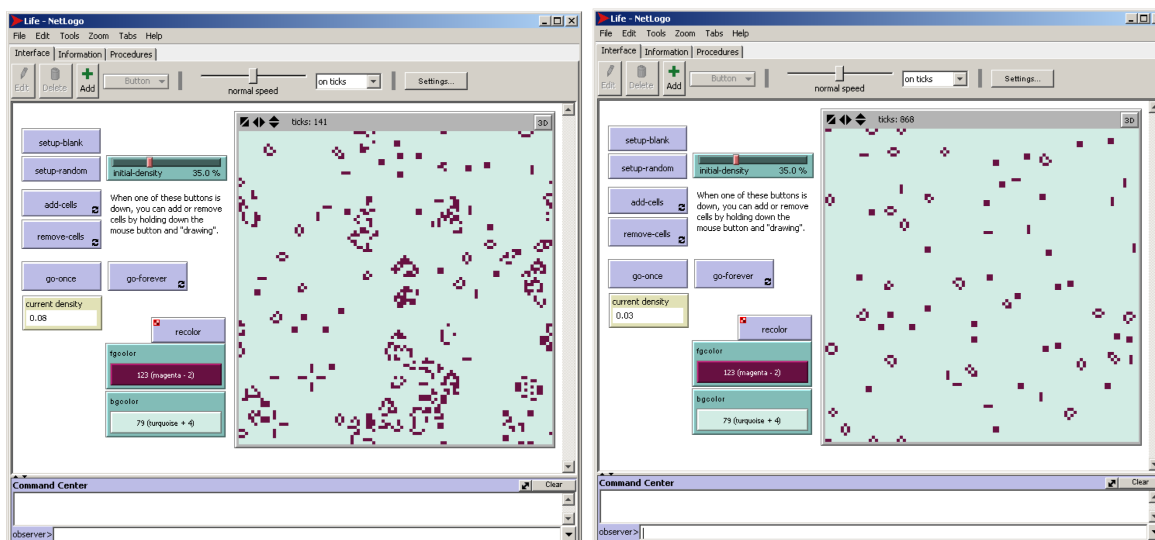
Pro experimentování s modelem máme k dispozici jednoduché GUI. V prvním kroku nastavíme hustotu modelu (initial density). Hustotou rozumíme, kolik % univerza modelu zaberou živé buňky. Implicitní nastavení hustoty je na 35%, které postačují pro demonstraci (ale klidně experimentujte s jiným nastavením).

Kliknutím na tlačítko setup-random si vygenerujeme počáteční stav našeho univerza. Simulaci spustíme kliknutím na go-once (každé kliknutí 1 iterace) nebo go-forever (automatické provádění iterací podle nastavené rychlosti).

Modely založené na podobných principech jsou schopny zachytit



(počátečnístav) (30-tá iterace)



(141 iterace) (868 iterace - stabilní obrazec)

Obr. 39: Conwayova hra života

dobře některé dynamické děje jako může být třeba vztah predátor – kořist nebo simulace šíření lesního požáru apod. Podívejte se na některé další modely z knihovny modelů NetLogo.

Objevily se některé další aplikace buněčných automatů jako je třeba Coddův automat (1968) nebo Langtonovy Q-smyčky (1984). Novým impulzem pro výzkum v této oblasti však přinesl až Wolframův 1D buněčný automat.

5.2 Wolframův 1D buněčný automat

Americký matematik Stephen Wolfram začal s buněčnými automaty experimentovat na počátku 80. let. Na rozdíl od předešlých typů automatů však pracuje pouze s jednoduššími jednorozměrnými automaty.

Graficky takový automat lze zobrazit prostřednictvím jediné řádky. Výhodou také je, že průběh generací můžeme vizualizovat najednou na další řádky, aniž by došlo k přepsání výsledku minulých iterací.

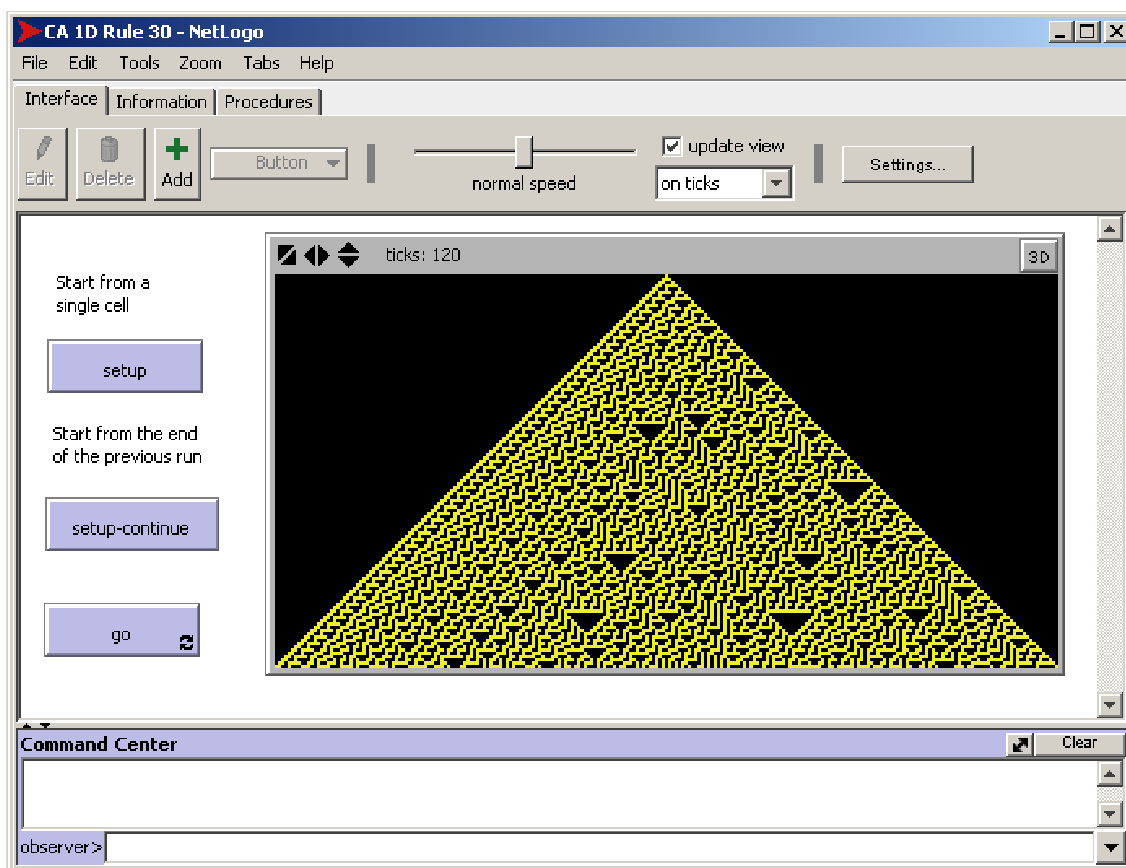
Pokud budeme zkoumat nejjednodušší případ automatů tohoto typ, pak okolí posuzované buňky budou tvořit právě dvě buňky, se dvěma možnými stavy každé z nich tato trojice může nabýt celkem 23 = 8 podob. Na tuto osmici může být aplikováno celkem 28 = 256 pravidel.

Tato pravidla označuje Wolfram podle dekadického zápisu pravidla. Například pravidlo 30 (označení dekadické) znamená v binárním zápisu 00011110.

Můžeme zapsat ve formě transformačního pravidla

```
okolí           111 110 101 100 011 010 001 000
výsledek         0   0   0   1   1   1   1   0
```

Podívejme se jakým způsobem toto pravidlo vypadá vizuálně (viz. obr. 40).



Obr. 40: Vizualizace pravidla 30

K obr. 40 připomínám, že každý řádek odpovídá jedné iteraci. NetLogo poskytuje několik dalších předpřipravených pravidel. Jinak můžete použít můj program Buněčné automaty 1.1 [22], který obsahuje větší množství zajímavějších pravidel „předpřipravených“.

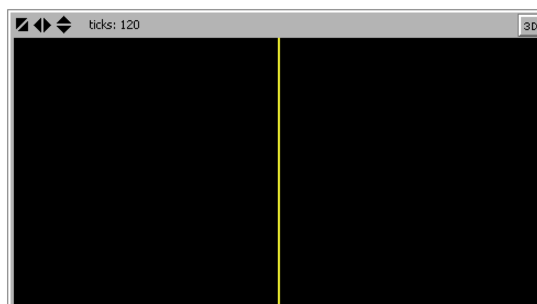
Tento program máte taktéž na doprovodném CD k těmto skriptům.

Tím, že možných pravidel pro tento typ buněčných automatů je pouze 256, bylo možné provést analýzu výsledků všech těchto pravidel. Wolfram zjistil, že všechny výsledky lze zařadit do jedné ze čtyř kategorií (všimněte si souvislosti s pozorovaným chováním u Conwayovy hry

života).
CA1 - všechny buňky nabudou stejné hodnoty (0 nebo 1) – např. pravidlo 40
CA2 – postupný přechod na opakující se tvary – např. pravidlo 228
CA3 – chaotický vývin (náhodný šum) – např. pravidlo 22
CA4 – složitá pravidelnost – např. pravidlo 110

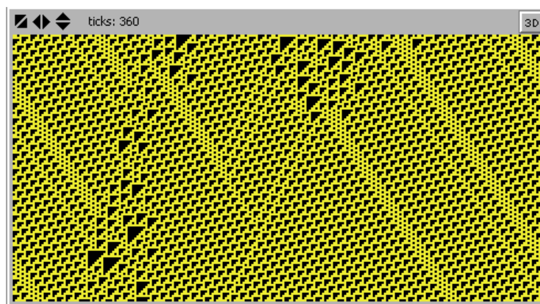
Výsledky těchto čtyř příkladů máte možnost vidět na obr 41.

CA1 - pravidlo 40



CA2 - pravidlo 228

CA3 - pravidlo 22



CA4 - pravidlo 110

Obr. 41: Wolframovy 1D automaty CA1 - 4

Bylo vysledováno, že některými z těchto pravidel se patrně řídí i příroda, např. při tvorbě některých vzorů na ulitách mlžů apod. (viz. obr. 42).

Obr. 42: Ulita mlže Conus textile (zdroj Wikipedie [23])

Do doby studia buněčných automatů se zdálo, že pro pigmentaci ulit těchto mlžů neexistuje žádné pravidlo (pigmentace je náhodná) a nebo je natolik složitá, že ji nejsme a pravděpodobně nikdy nebudeme schopni poznat. Ukázalo se že ani jedna z těchto úvah nebyla správná – ve skutečnosti pravidlo existovalo a bylo nesmírně jednoduché (viz. obr. 40).

Je možné, že podobným způsobem lze popsat i některé mechanismy, které v dnešní době považujeme za chaotické, což by výrazně zvýšilo naši šanci je podchytit modely.

5.3 Boidi – Raynoldsův model shlukování ptáků

Raynolds se při vytváření svého modelu nechal inspirovat hejny tažných ptáků. Tažní práci se pohybují v hejnech, která jako celek vykazují poměrně složité chování. Přesto se ukazuje, že toto chování je výsledkem aplikace pouze velmi jednoduchých pravidel, a že s použitím těchto pravidel je možné vytvářet modely, které věrně simulují chování jedinců v těchto hejnech.

Tato pravidla jsou:

1) shromažďování – jedinec se snaží přesunout k těžišti svých sousedů
2) sladění rychlosti – jedinec se pohybuje stejnou rychlostí jako jeho sousedé
3) vyhnutí se kolizi – jedinec opouští své místo pokud se neúměrně přiblíží jinému jedinci nebo nějaké překážce.

Podobná pravidla lze aplikovat i na jiné druhy zvířat žijících v hejnech jako jsou třeba některé druhy ryb.

Na obr. 43 je výsledek jednoduchého modelu boidů. Počáteční rozložení a směrování boidů v modelu je čistě náhodné, ale postupem času aplikací jednoduchých pravidel se boidi začínají shlukovat do malých hejn a taktéž se upravuje směr pohybu.

Tento model si můžete vyzkoušet sami v NetLogu z knihovny modelů -> Perspective demos -> Flocking (Perspective Demo).

5.4 Lindenmayerovy systémy

Lindenmayerovy systémy, někdy také nazývané zkráceně L-systémy, jsou pojmenovány po Arisitdu Lindenmayerovi, který modifikoval formální logiku pro účely modelování vývoje jednoduchých organismů.

L-systém je potřeba chápat jako řetězec:

$$G = \{V, S, \omega, P\} \quad (10)$$

kde V … proměnné (proměnné elementy)
S … konstanty (neměnné elementy)
ω … počáteční stav
P … pravidla

Obr. 43: Chování hejna

Můžeme demonstrovat na jednoduchém případě Sierpińského trojúhelníku.

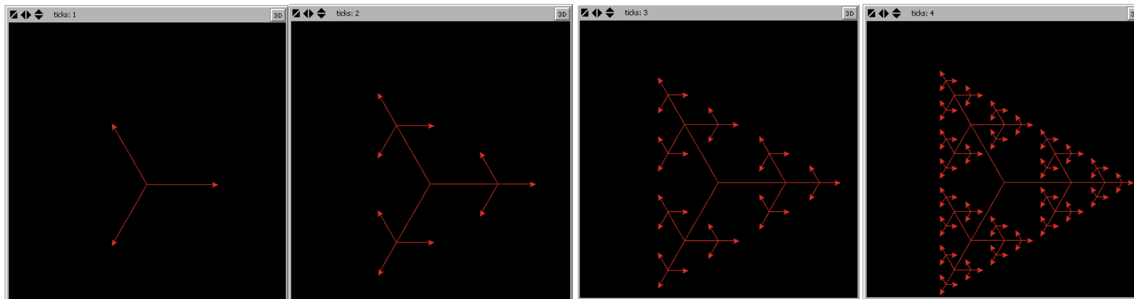
proměnné: A B
konstanty: + -
počáteční stav: A
pravidla: (A -> B – A – B), (B -> A + B + A)
úhel: 60˚

Pravidlo tvoří předpis provádějící transformaci proměnné. Proměnná A se tedy transformuje podle předpisu B – A – B. Tento předpis je možné interpretovat pomocí pravidel programovacího jazyka Logo, kde operátor + znamená otočení orientace v našem případě o 60˚ doleva, a operátor – otočení o 60˚ doprava.

Tedy B – A – B provede vykreslení B, otočí pero o 60˚ doprava,

vykreslí A, otočí o 60° doprava a vykreslí B.

L-systémy podobně jako běžně buněčné automaty pracuje v iteracích. Podívejme se na obrázek 44, který lépe demonstruje způsob konstrukce tohoto fraktálního obrazce.



Obr. 44: Sierpinského trojúhelník

Na obr. 44 jsou znázorněny 4 iterace L-systému podle výše uvedených pravidel. Simulace však může probíhat neomezeně dlouho. V každé další iteraci se trojúhelník zahustí a výpočet bude trvat podstatně déle. Ostatně si to můžete vyzkoušet sami v NetLogu – knihovna modelů -> Mathematics -> Fractals -> Sierpinski Simple.

Zajímavá je i konstrukce vločky Kochové, kterou tady ovšem uvádět nebudu, máte možnost si ji spustit z téhož umístnění (pod názvem Koch Curve).

Svůj věhlas získaly však L-systémy především svou schopností modelovat růst rostlin nebo jejich částí (např. listů). Výsledkem takového modelu může být třeba následující obrázek

Výhody využití tohoto postupu jsou jasné. Tvary popsatelné pomocí L-systémů není potřeba uchovávat ve formě 3D modelu, stačí pouze

Obr. 45: Modelování travin pomocí fraktálů (zdroj Wikipedie [24])

specifikovat předpis, podle kterého se vytvoří, jejich vytvoření může proběhnout za běhu daného programu. K těmto činnostem jsou dnešní výkonné grafické karty jako dělané.

5.5 Programovací jazyk Logo

Zatím jsme pracovali v prostředí NetLogo uživatelsky, tedy používali jsme rozhraní, které pro nás vytvořil někdo jiný. Na tomto přístupu není samo o sobě nic špatného, pouze jsme odkázáni na někoho jiného, který s největší pravděpodobností nevytvoří takový model jaký bychom potřebovali.

Takže několik málo slov o programovacím jazyku Logo. Jedná se o programovací jazyk (počátek 1967) určený pro výuku programátorských konceptů pro malé děti. Funguje to tak, že děti dostanou k dispozici kurzor (ve tvaru želvy), kterou může dítě ovládat a pomocí ní kreslit.

V ČR se používal jiný koncept pro výuku programování a to programovací jazyk Karel [32], kde uživatelé ovládali robotka Karla, který mohl různě přenášet kostičky apod.

Podívejme se na program pravidla 22.

```
globals [row]
patchess-own [left-pcolor center-pcolor right-pcolor]

to setup
     ca
     set row max-pycor
     ask patch 0 max-pycor [ set pcolor yellow ]
end

to go
     if (row = min-pycor)
          [ stop ]
     ask patches with [pycor = row]
          [ do-rule ]
     set row (row – 1)
     tick
end

to do-rule
     set left-pcolor [pcolor] of patch-at -1 0
     set center-pcolor pcolor
     set right-pcolor [pcolor] of patch-ar 1 0
     if((left-pcolor = yellow and center-pcolor =  black
and right-color = black) or
     (left-pcolor = black and center-pcolor = yellow and
```

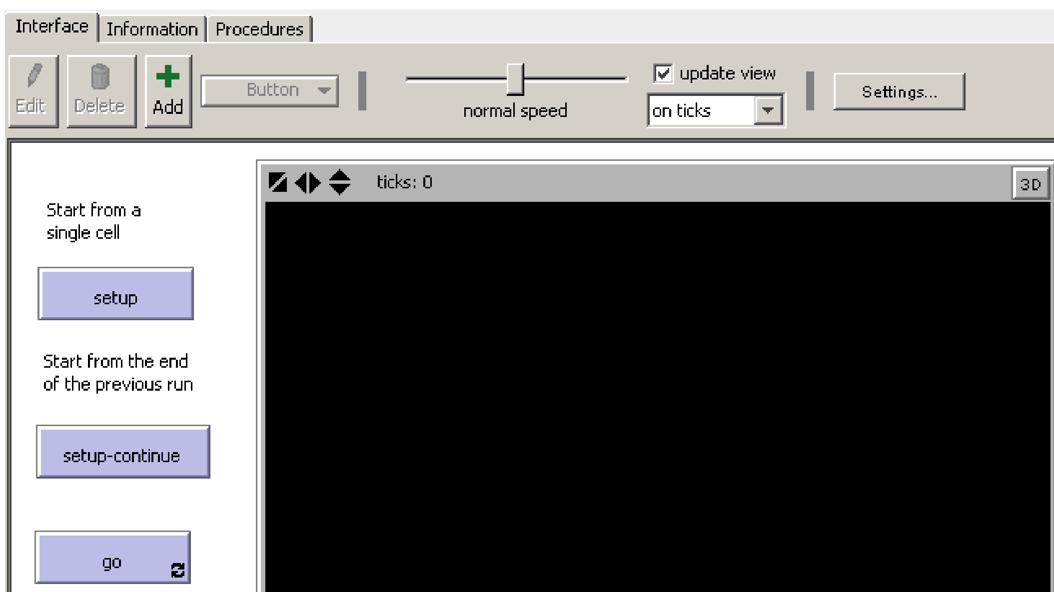
```
right-pcolor = black) or
     (left-pcolor = black and center-pcolor = black and
right-pcolor = yellow) or
     left-pcolor = black and center-pcolor = yellow and
right-pcolor = yellow))
          [ set [pcolor] of patch-at 0 -1 yellow ]
          [ set [pcolor] of patch-at 0 -1 black ]
end

to setup-continue
     ask patches with [pycor = max-pycor]
          [ set pcolor ([pcolor] of patch pxcor min-
pycor) ]
     ask patches with [pycor != max-pycor]
          [ set pcolor black ]
     set row max-pycor
end
```

Jak vidíte samotný program není dlouhý (pouze jedna stránka) a vyskytuje se v něm pouze minimum příkazů. První dva řádky mají deklarační charakter. První z nich definuje proměnnou row jako globální proměnnou dostupnou ve všech procedurách programu.

Proměnné left-pcolor, center-pcolor a right-pcolor jsou dostupné pouze buňkám (patches-own). Při zkoumání okolí buňky v 1D hodnotíme buňku samotnou a její okolí, tedy buňky vpravo a vlevo. Stav buňky můžeme vyjádřit graficky změnou barvy. V tomto programu budeme využívat barvy 2 a to černou pro „mrtvou“ buňku a žlutou pro „živou“ buňku.

Předtím něž se podíváme na zbytek programu, podívejme se na uživatelské rozhraní.



Obr. 46: GUI NetLogo

Jednotlivá tlačítka, nebo i jiné prvky GUI se přidávají velmi jednoduše kliknutím na tlačítko velkého plus s popiskou add (tedy přidat) a výběrem typu prvku GUI, který je potřeba přidat. Na pracovní ploše potom jednodiše tento prvek vytvoříme kliknutím a tažením.

Po vytvoření prvku je pak nutno nastavit základní vlastnost, především jméno tohoto prvku, ale i některé další věci.

V našem případě jsou již prvky GUI předpřipraveny. Všimněte si nápisu na tlačítcích setup, setup-continue a go. Pokud se podíváte do programového kódu tak zjístíme, že jsou v něm obsaženy procedury to setup, to setup-continue a to go. Logo totiž obsahuje velmi jednoduchý model událostí, zejména pro tlačítka. Po kliknutí na tlačítko se provede procedura nazvaná stejně.

Kromě těchto tří procedur napojených na tlačítka je v programu ještě procedura do-rule, která se stará o aplikaci pravidla. Podívejme se na jednotlivé procedury.

Procedura setup se stará o úvodní nastavení modelu. Provede se resetování univerza modelů do prázdné (černé) podoby a do prvního řádku se přidá jediná aktivní buňka. Proberme proceduru příkaz za příkazem

ca je synonymum pro příkaz clear-all, který provede smazání obsahu všech.

Příkaz set nastaví proměnnou row na hodnotu max-pycor, tedy maximální hodnotu Y-nové koordináty buňky dostupné v našem univerzu.

Konečně poslední řádek procedury setup můžeme do běžné řeči přeložit, že požadujeme (ask) po buňce (patch) uprostřed (0) v prvním řádku (max-pycor), aby nastavil (set) svou barvu (pcolor) na žlutou (yellow). Protože bychom po buňkách mohli chtít více operací oddělujeme to co mají buňky (nebo cokoliv jiného) udělat do hranatých závorek.

Jako alternativou k proceduře setup je procedura setup-continue. Tato procedura opět provede vyresetování univerza, jako počáteční stav se ale použije poslední iterace předchozího běhu modelu. Nezačínáme tak s jednou buňkou ale větším množství buněk.

Procedura go provádí procházení univerza v rámci jednotlivých iterací modelu. Činnost probíhá tak, že se zkontroluje zda jsme náhodou nedošli na poslední řádek univerza (if (row = min-pycor) a v takovém případě se přeruší (stop) běh programu, v opačném případě se ale požádají buňky na řádku (ask patches with [pycor = row]) aby na nich bylo aplikování pravidlo obsažené v proceduře do-rule. Nakonec dojde k přesunu na další řádek (set row (row – 1)) a provedení další iterace (tick).

Pravidlo samotné je aplikováno procedurou do-rule. Tato procedura nejprve načte do promenných stav posuzované buňky a jejích okolí a potom rozhodne, zda výsledkem bude „živá“ nebo „mrtvá“ buňka.

Pravidlo 22 vypadá následovně:

```
111 110 101 100 011 010 001 000
0   0   0   1   0   1   1   1
```

Pro aplikaci pravidla musíme otestovat zda konfigurace buněk odpovídá některé ze čtyř konfigurací vedoucích k výsledku 1 (život). všechny ostatní logicky musí vést k výsledku 0. Tuto funkčnost zajistíme použitím podmínky, kde jako predikát použijeme deklarativní vyjádření výše uvedeného pravidla, např. 100 zapíšeme left-pcolor = yellow and center-pcolor = black and right-pcolor = black.

Kontrolní otázky

1) Jaké je použití L-systémů?
2) Řadíme buněčné automaty mezi statické nebo dynamické modely?
3) Vysvětlete způsob chování ryb v hejnu.
4) Jaký je rozdíl mezi von Neumanovým a tzv. úplným okolím?

Úkoly k zamyšlení

1) Prozkoumejte další předpřipravené modely v NetLogu.
2) Zkuste modifikovat existující pravidlo 22 1D Wolframova buněčného automatu na pravidlo 222.

6 Multiagentní systémy

Průvodce studiem

V této kapitole se seznámíme se možnostmi využití jednodušších samostatně fungujících komponent na bázi umělé inteligence (agentů), pro získání komplexního – inteligentního chování.

Po prostudování této kapitoly budete vědět

- co je to agent
- jak takový agent funguje
- jakým způsobem dochází k výměně informací mezi agenty

Čas pro studium

Na prostudování této kapitoly budete potřebovat minimálně dvě hodiny.

6.1 Inteligentní agenti

Multiagentní systémy jsou oblastí umělé inteligence, která zejména v poslední době nabývá na významu. Multiagentní systémy se zásadně liší od aplikací umělé inteligence, které jsme probírali dosud. Všechny tyto druhy se totiž snaží napodobit jak nejlépe je to možné systém uvažování myslící bytosti. Expertní systémy svou pokročilou funkčností v oblasti odvozování poznatků, neuronové sítě se snaží podchytit samotnou fyzickou strukturu mozku. Agentní systémy na takové komplexní chování rezignují.

Úkolem agenta není přiblížit se co možná nejvíce myslící bytosti, ale vykonávat co nejlépe úkol mu svěřený. Takových agentů přitom může v modelovaném univerzu být celá řada s podstatně odlišnými úkoly i prostředky, které jednotliví agenti používají pro jejich splnění. Agenti dokonce mohou používat expertní systémy nebo neuronové sítě, pokud se při jejich návrhu přijde na to, že by to mohlo být efektivní.

Komplexnosti chování se dosáhne pomocí výměny informací mezi agenty. Cílem takové informační výměny je, aby se vhodně zkombinovala funkčnost všech agentů.

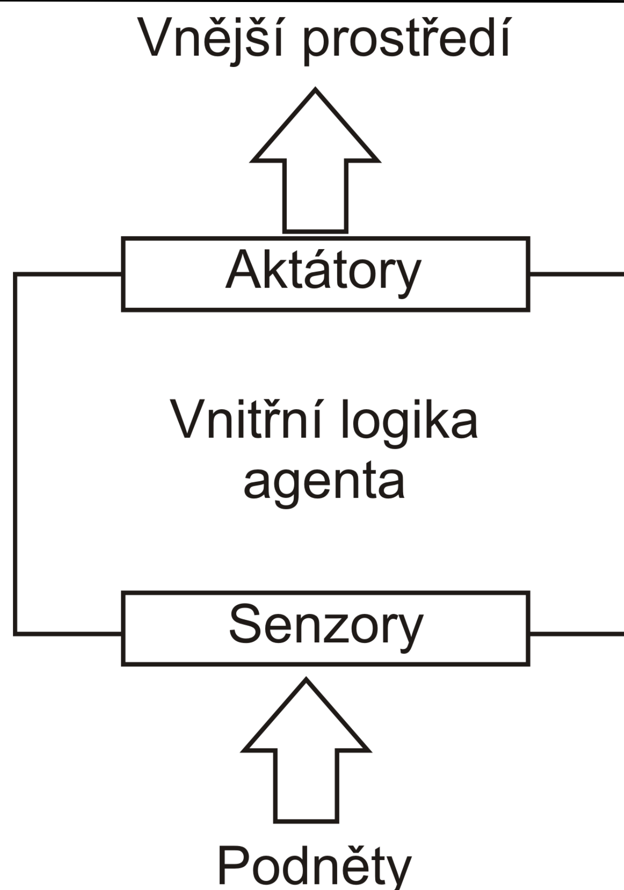
Agent

Jakým způsobem můžeme agenta definovat, Kubík [25] definuje inteligentního agenta následovně: Agent je entita zkonstruovaná za účelem kontinuálně a do jisté míry autonomně plnit své cíle v adekvátním prostředí na základě vnímání prostřednictvím senzorů a prováděním akcí prostřednictvím aktuátorů. Agent přímo ovlivňuje podmínky prostředí tak, aby se přibližoval k plnění cílů.

Podívejme se na vnitřní konstrukci agenta (viz. obr. 47).

Senzor

Agent získává podněty z okolí prostřednictvím senzorů, kterými je vybaven. Za senzory přitom považujeme vše, co je schopno přijímat informace z vnějšku, v případě softwarových agentů se tak může jednat o



Obr. 47: Konstrukce agenta

nějakou službu nebo interface, který zachytává určité údaje.

Aktátor

Údaje zachycené senzory jsou zpracovávány vnitřní logikou agenta, tato vnitřní logika přitom může být rozdílná, proto se k ní ještě vrátíme podrobněji později. Každopádně na základě těchto podnětů začne agent konat prostřednictvím mechanismů k tomu určeným tzv. aktátorů.

Aktátory rozumíme všechny mechanismy, kterými agent působí na své okolí. Může se jednat o kola pro přesun agenta, může se jednat o manipulační „ruku“, nebo o nějaké jiné nástroje.

Z hlediska vnitřní logiky agenta je možné rozdělit agenty do dvou skupin:

- reaktivní agenty
- deliberativní agenty

Reaktivní agenti

Reaktivní agenti svou činností pouze reagují na změny v okolním prostředí. V této koncepci se vychází z toho, že živé organismy vykonávají řadu funkcí automaticky, aniž by musely nad těmito úkoly přemýšlet, prostě jako reakce na různé události, se kterými se setká. Reaktivní agenti jsou jsou založení právě na tomto principu.

Deliberativní agenti

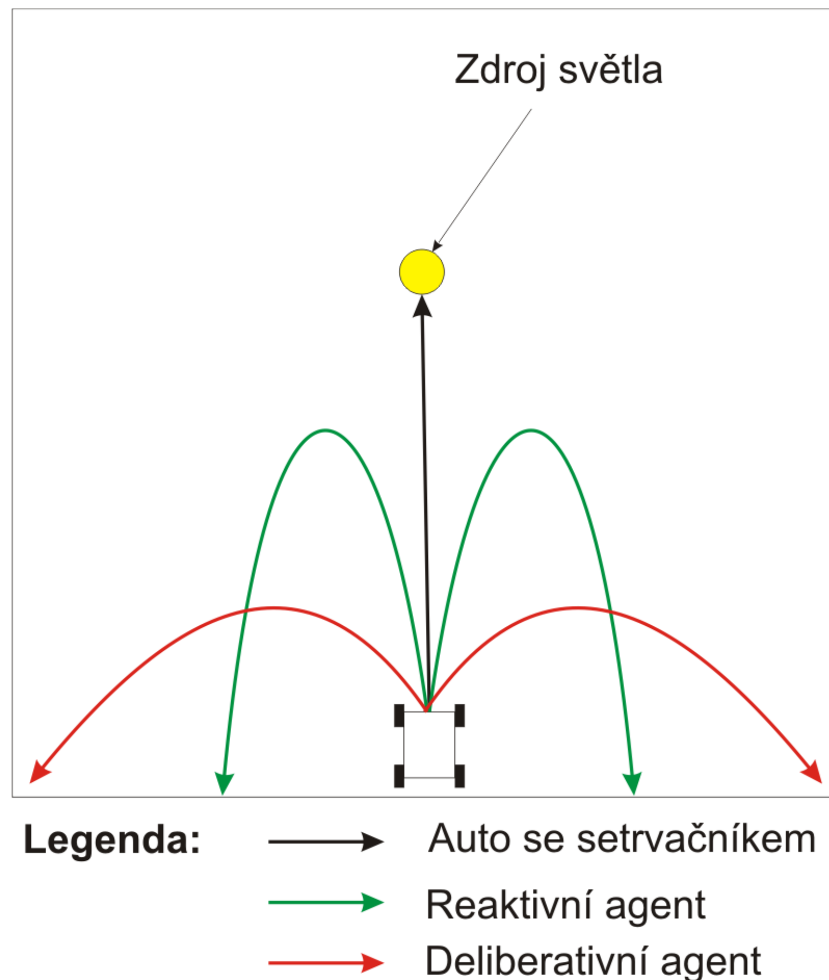
Deliberativní agenti oproti agentům reaktivním vytvářejí vnitřní reprezentaci (model) problémové situace, mají určité cíle a těchto cílů se snaží optimálně dosáhnout. Agent v tomto případě má k dispozici konečnou sadu akcí, které může použít pro řešení situace. Agent nejprve

zkusí jaký následek bude mít použití těchto akcí ve vnitřním modelu a určí optimální akci nebo posloupnost akcí které povedou k jeho cíli.

Teprve v tomto okamžiku se akce považované za optimální začnou provádět, agent zároveň kontroluje prostřednictvím senzorů, zda prováděné akce skutečně vedou k očekávaným výsledkům a pokud ne provádí opětovné přehodnocení.

Demonstrujme si jednoduchý příklad reakce agenta. Náš agent bude mít podobu autíčka, jehož úkolem bude dostat se do místa co nejdále od zdroje světla.

Reakce různých typů agentů jsou zobrazeny na obr. 48.



Obr. 48: Reakce různých typů agentů

Auto se setrvačníkem samozřejmě není agentem a do obrázku je přidáno jednom z důvodu demonstrace směru, kterým se agent v počátku simulace ubírá. Autíčko se setrvačníkem prostě pokračuje kupředu bez ohledu na své okolí, protože prostě není ani trošku inteligentní.

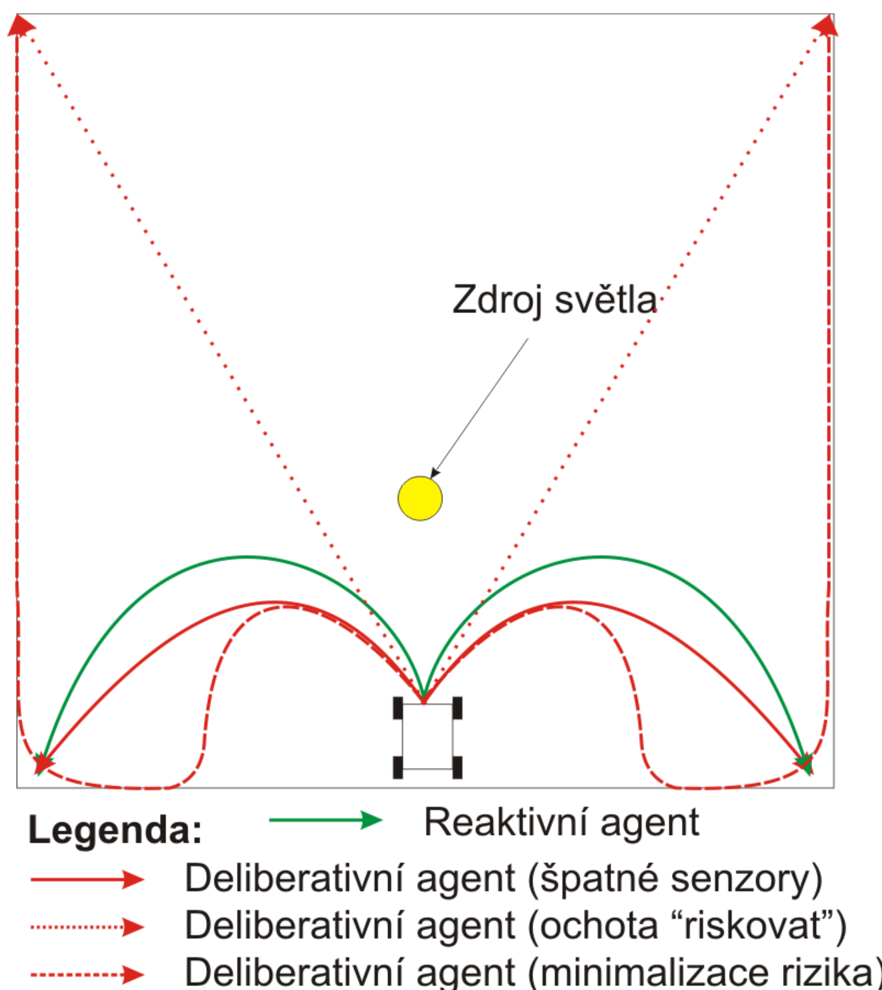
Reaktivní agent již vykazuje inteligentnější chování. Jeho pouť začíná stejným způsobem jako pouť autíčka setrvačníkem, tedy kupředu. Reaktivní agent ale záhy zaregistruje prostřednictvím svých senzorů, že intenzita světla, kterou má minimalizovat roste a je nucen na to reagovat. Reakce je v tom, že se začne otáčet doleva nebo doprava (rozhodnutí může být náhodné) tak dlouho, než nabere směr od zdroje světla.

Zakřivení křivky pohybu (prudkost reakce) agenta je dána nastavením parametrů agenta. Reaktivní agent tak úplně klidně může skončit ve stejném umístnění jako deliberativní agent, dá se ale očekávat, že mu to bude trvat déle, protože jeho křivka pohybu nebude optimální.

Konečně deliberativní agent pomocí senzorů „pozná“ prostředí, ve kterém se pohybuje, na jeho základě si vytvoří model (vnitřní reprezentaci) prostředí a naplánuje optimální trasu pohybu aby se dostal do nejvzdálenějších rohů, kde je intenzita světla nejmenší.

Zakřivení křivky pohybu bude jen tak velké, aby se agent otočil do požadovaného směru.

Rozdíl mezi oběma druhy agentů bude ještě markantnější pokud předchozí pokud modifikujeme tak, aby zdroj světla byl na stejné půlce jako hrací plochy (univerza agentů), viz obr. 49.



Obr. 49: Rozdíl v pohybu agentů

Reaktivní agent se bude chovat přibližně stejně. Zásadní rozdíl v chování ale může nastat u deliberativního agenta, protože v závislosti na kvalitě senzorů a vnitřního modelu může odhalit, že ve vzdálenější části univerza se nacházejí oblasti s nižší intenzitou světla než se nachází v jeho bezprostředním okolí.

Pokud agent disponuje dostatečnými senzory a vnitřním modelem o cestě do vytouženého místa rozhodne jeho tolerance k dočasnému

„nepohodlí“, tedy dočasně se zvyšující intenzitě světla. Agent si tak může optimalizovat své nepohodlí plánováním trasy na základě pravidel daných tvůrcem agenta.

6.2 Multiagentní systémy

Multiagentní systémy

Multiagentní systémy jsou systémy složené z inteligentních agentů, kteří vzájemně spolupracují nebo na sebe jinak působí.

Z tohoto pohledu za nejjednodušší multiagentní systém může být považován jakýkoliv buněčný automat. Jednotlivé buňky by pak byly agenty a působily by na sebe svým stavem. Takovýto multiagentní systém by nám však příliš užitku (ve srovnání s teorií buněčných automatů) nepřinesl.

Podívejme se na složitější příklad. Pro demonstraci možností multiagentního simulačního prostředí se obvykle používá simulace heatbugs, která zachycuje život brouků, kteří žijí ve světě s místy o různé teplotě. Brouk, pokud má správnou teplotu, je šťastný a nepohybuje se, pokud je mu ale zima nebo horko, snaží se dostat na teplejší nebo chladnější místo.

Tuto simulaci si můžete prohlédnout v NetLogu, ale také v jiných simulačních balících jako je Repast [26] nebo Swarm [27]. Takové simulace jsou však obvykle založeny na zkoumání emergentního chování agentů. Tedy jako takové spadají někde mezi umělý život a agenty. Tyto nástroje je možné použít i pro simulace skutečných agentů, je to pouze o hodně složitější.

Heatbugs na sebe opět působili svou přítomností, co když ale potřebujeme zajistit, aby se agenti přímo domluvili. V takovém případě je nutné implementovat společný komunikační protokol, který všichni agenti dodrží. Přitom je potřeba si uvědomit, že vnitřní logika agentů může být velmi odlišná od operačního systému, ve kterém jsou implementováni až po různé programovací jazyky.

Samozřejmě je možné vyvinout komunikační protokol přímo pro naši aplikaci multiagentní technologie, nicméně by se jednalo o proces svým významem odpovídajícímu opětovnému vynálezu kola. Existuje řada protokolů, které pro tento účel lze použít. Z této řady jsou nejrozšířenější dva a to konkrétně KQML [28] a FIPA-ACL [29].

Komunikační akty

Význam těchto protokolů spočívá především v tom, že standardizují tzv. komunikační akty mezi agenty. Při návrhu komunikačního protokolu se vycházelo z lingvistické analýzy běžného jazyka. Teorie komunikačního aktu vychází z hypotézy, že řečový akt (to co říkáme) souvisí s tím co je nebo bude provedeno.

Příklad vypíši zkušební termín na středu druhý týden v únoru. Tento akt deklaruje co bude provedeno v budoucnu.

Komunikační akty podle obsahu lze rozdělit [25] do několika kategorií:

oznamovací – tento akt sděluje dosažení nebo nedosažení nějakého cíle nebo třeba zjištění důležité okolnosti. Př.: Dorazil jsem na místo. Baterie je plně nabitá. Diagnostika systému proběhla v pořádku.
zavazující akt – zavazuje účastníka komunikace provést nějakou činnost. Př.: za 5 minut provedu odečet hodnoty.
přikazující akt přikazuje příjemci vykonat nějakou činnost. Př. Proveď odečet hodnoty. Zavři, ventil.

Komunikace mezi agenty má formu obvykle odpovídající výše uvedeným kategoriím.

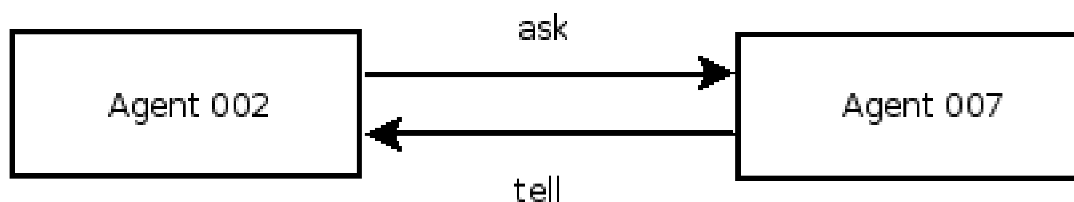
V KQML by samotný komunikační akt mohl vypadat následovně:

```
(ask-one
     :sender        Agent002
     :reciever      Agent007
     :language      prolog
     :ontology      NL
     :content       „identifyNL(0055)‟
)
```

Tento komunikační akt vyžaduje odpověď agenta 007 agentovi 002 na identifikaci nebezpečné látky z ontologie NL (nebezpečné látky).
Performativ

Každý protokol obsahuje odlišnou sadou dostupných komunikačních aktů. Tyto akty, jelikož vyžadují na druhé straně nějakou činnost nazýváme performativy.

Agenti mohou komunikovat přímo (viz. obr. 50). Například na výše uvedenou otázku by mohl agent 007 přímo odpovědět jménem identifikované nebezpečné látky (otázka - odpověď).



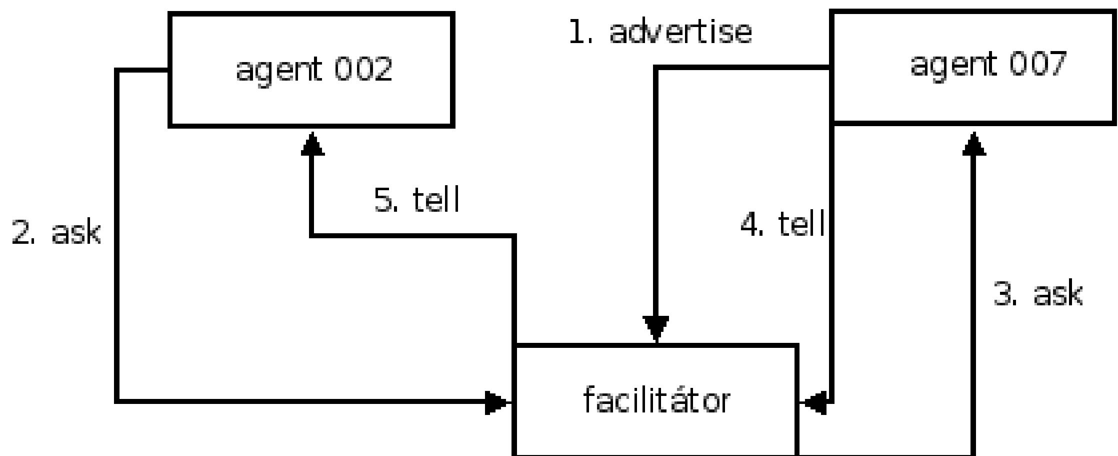
Obr. 50: Přímá komunikace mezi agenty

Facilitátor

Taková komunikace však vyžaduje splnění několika podmínek. Agenti o sobě musí vědět nejenom jací agenti jsou v prostoru přítomni, ale také jaké služby jsou schopni poskytovat. V případě, že toto nejsme schopni nebo ochotni zajistit, je potřeba použít jiný typ scénáře a to s použitím prostředníka – facilitátora.

Úkolem facilitátora je „seznámit“ agenty v případě potřeby a nebo zprostředkovat poskytnutí služby (viz. obr. 51).

V prvním kroku zaregistruje své schopnosti agent 007 u facilitátora.



Obr. 51: Komunikace agentů prostřednictvím facilitátora

Podobným způsobem by měly postupovat všichni agenti, aby pak v okamžiku, kdy vznikne nějaký požadavek (2) byl facilitátor schopen najít a zkontaktovat (3) vhodného agenta, získat od něj odpověď (4) a tuto odpověď sdělit původnímu agentovi.

V takovém komunikačním scénáři agenti nemusí mít povědomí o počtu a schopnostech dalších agentů. Stačí jim znát facilitátora, který jim zprostředkuje veškeré odpovědi.

Facilitátora je možné použít i distribuovanější formě. V takovém případě facilitátor přímo nezprostředkovává poskytování služeb, ale pouze poskytuje kontaktní informace na vhodného agenta a očekává, že tito agenti se domluví sami na přímo.

Kontrolní otázky

1) Co je to komunikační akt?
2) Jaký je rozdíl mezí inteligentním agentem a multiagentním systémem.
3) Co je to performativ?
4) Jakou úlohu plní facilitátor?
5) Vzpomněli byste si na nějaké kategorie komunikačních aktů (vyjmenujte alespoň dva)?

Otázky k zamyšlení

1. Prozkoumejte model heatbugs v NetLogu nebo jiném simulátoru multiagentních systémů.

7 Genetické algoritmy

Průvodce studiem

V této kapitole se podíváme na použití evolučních strategií pro řešení některých zejména optimalizačních problémů.

Po prostudování této kapitoly budete vědět

- jak funguje genetický algoritmus
- jakým způsobem se používají mutace, křížení a přirozený výběr
- jaká jsou úskalí použití genetických algoritmů

Čas pro studium

Pro prostudování tohoto tématu budete potřebovat minimálně dvě hodiny.

Evoluce

Genetické algoritmy vycházejí z pojetí evoluce tak jak jej popsal Charles Darwin v roce 1858 ve své knize On the Origin of Species. Evoluce má dva hybné mechanismy – geny, jako základní prostředek kterým se předávají informace z předchozích generací a potomky, a přirozený výběr- který preferuje jedince, jejichž genetická výbava je činí maximálně přizpůsobenými prostředí, ve kterém žijí.

Genetické algoritmy vycházejí přesně z tohoto konceptu a používají jej pro řešení především optimalizačních problémů. Tyto algoritmy přitom nejsou schopny hledat optimální řešení problémů ale pouze sub-optimální řešení. V tomto smyslu tedy fungují podobně jako třeba neuronové sítě.

Jelikož u genetických algoritmů vycházíme z biologie používáme i podobnou terminologii. Kupříkladu pracujeme s populací – tedy souhrnem jedinců různých vlastností (majících jinou „genetickou" výbavu). Základním mechanismem optimalizace je použití křížení – tedy kombinace dvou (nebo více) jedinců za účelem vzniku nového jedince – potomka.

Fitness funkce

Aby celý algoritmus směřoval k nějakému řešení je nutné použít nějaké optimalizační pravidlo. V přírodě tuto funkci plní schopnost reprodukce. Schopní jedinci mají větší šanci na reprodukci než ti méně úspěšní. V počítači musíme tuto funkci nahradit funkcí umělou poměřující úspěšnost jedinců v řešení našeho problému. Tuto funkci můžeme nazývat fitness funkcí. Bez přítomnosti fitness funkce je sice možné genetický algoritmus spustit, ale není možné dosáhnout optimálního řešení, protože by v populaci nedocházelo k optimalizaci, resp. i kdyby docházelo, nejsme schopni to zhodnotit.

Křížení

Křížení probíhá v iteracích (tedy diskrétních časových krocích). V přírodě tento přístup není obvyklý, ale z hlediska jednoduchosti simulace a úspory výpočetního výkonu se jedná o krok logický a nutný. Tyto iterace obvykle nazýváme generace.

Nová generace obvykle do určité míry nahradí generaci původní. Toto nahrazení může být úplně – tedy rodiče po splození potomka okamžitě uhynou, v takovém případě hovoříme o generační evoluci. Pokud rodiče a potomci koexistují nějakou dobu společně, pak hovoříme o tzv. stále evoluci.

Jelikož genetické algoritmy jsou umělé můžeme si typ evoluce volit sami, obě dvě přitom mají určité výhody i nevýhody. Generační evoluce je mnohem dynamičtější, změny ve vlastnostech populace jsou velmi rychlé. Nevýhodou tohoto přístupu je to, že v rámci selekce mohu vyřadit jedince, kteří mají z hlediska řešení problému zajímavé vlastnosti.

Stálá evoluce takové nevýhody nemá, nicméně jelikož vývoj v ní je mnohem plynulejší, je tento vývoj také mnohem pomalejší (potřebuje mnohem více generací).

Mutace

Křížení jako takové umožňuje, aby jedinci nových generací dědily pouze vlastnosti svých předků. Problémem tohoto přístupu je, že různorodost těchto vlastností je konečná a daná iniciací počáteční generace. Nikde přitom není zajištěno, že požadované vlastnosti (které by se měly objevit ve výsledku evoluce) jsou v populaci vůbec obsaženy nebo že je v rámci selekce nevyřadím. Z těchto důvodů se používá také operátor náhodné změny jedince – mutace.

BSF

Mutace zajistí, že vývoj nebude stagnovat, není schopna sama o sobě ale zajistit, aby v populaci byly zachovány žádoucí znaky. Z tohoto důvodu se někdy používá BSF strategie. BSF je zkratka pro best so far. Strategie spočívá v tom, že jedinec s nejlepšími vlastnostmi automaticky zůstává přítomen v populaci (není možné jej vyřadit) do doby než se objeví nový jedinec s ještě lepšími vlastnostmi.

Selekční strategie

Předtím než se začneme zabývat selekčními strategiemi, podívejme se na matematické vyjádření toho, co jsme si v předchozích odstavcích pověděli.

Populace G v generaci t je tvořena jedinci xt,i.

$$G(t)=\{x_{t,1}, x_{t,2}, \ldots, x_{t,N}\} \quad (11)$$

Schopnosti jednotlivců v populaci hodnotíme pomocí fitness funkce. Pro fitness funkci f(x) by mělo platit, že je nezáporná. V takovém případě lze implementovat mechanismus ruletového kola. Pro vzorec (12) genetický algoritmus řeší maximalizaci fitness funkce.

$$P(i)=\frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)}, i=1, \ldots, N \quad (12)$$

Účelem selekce je výběr je preference jedinců, kteří mají z hlediska řešení problému dobré vlastnosti. Ve vzorci (12) bude pravděpodobnost výběru jedince i P(i) vyšší pro vyšší hodnoty fitness funkce. To je také důvod proč tento vzorec nelze použít pro záporné hodnoty fitness funkce

a řešení minimalizačních optimalizačních problémů. Oba tyto problémy lze však odstranit pomocí formálních úprav fitness funkce (alespoň na intervalu našeho zájmu).

Výsledkem selekce podle vzorce (12) je preference jedinců s lepšími (nadprůměrnými) vlastnostmi na úkor jedinců s vlastnostmi horšími (podprůměrnými). Počet průměrných jedinců zůstává přibližně stejný.

Nevýhodou tohoto přístupu je nutnost dostatečně velké populace, se kterou pracujeme.

Pro připomenutí statistiky – při dostatečném počtu nezávislých pokusů lze předpokládat, že relativní četnosti těchto pokusů se budou blížit teoretické hodnotě pravděpodobnosti. Matematicky, lze toto tvrzení vyjádřit následovně (13):

$$\lim_{n \to \infty} P(|\frac{1}{n}\sum_{i=1}^{n} X_i - \frac{1}{n}\sum_{1=1}^{n} E(X_i)| < c) = 1 \qquad (12)$$

Tuto náhodnou selekci je možné dále modifikovat, zejména s ohledem na to, že obvykle máme k dispozici pouze omezenou populaci, v takovém případě bohužel nemůžeme použít vzorec (12) determinističtější způsob, lze použít např. zbytkový stochastický výběr nebo řadu dalších strategií.

Podívejme se na praktickou ukázku použití genetických algoritmů na řešení problému obchodního cestujícího. Pro náš pokus použijeme FGA: Graphical TSP solver [30]. Jak víme, problém obchodního cestujícího je tzv. NP kompletní problém, který není možné pro velký počet měst řešit standardními cestami (numerickým výpočtem).

Řešení problému je vidět na obr. 52.

Obr. 52: Řešení problému obchodního cestujícího (zdroj [30])

Kontrolní otázky

1) Jaký je rozdíl mezi křížením a mutací?
2) Vysvětlete princip ruletového kola pro selekci jedinců?
3) V čem spočívá přínos BSF?

Literatura

[1] Page-Jones, M.: *Základy objektově orientovaného návrhu v UML*. Grada: Praha 2001, 367 str., ISBN 80-247-0210-X
[2] *ISO/IEC 19501:2005 Information technology -- Open Distributed Processing -- Unified Modeling Language (UML) Version 1.4.2*
[3] *Dia a drawing program* [on-line]. Dostupné z WWW <URL: http://www.gnome.org/projects/dia/ > [cit. 2007-07-07]
[4] *Domácí stránky StarUML*. Dostupné z WWW <URL: http://staruml.sourceforge.net/en/ > [cit. 2007-07-07]
[5] *Domácí stránky ArgoUML*. Dostupné z WWW <URL: http://argouml.tigris.org/ > [cit. 2007-07-07]
[6] *Domácí stránky Visual Paradigm*. Dostupné z WWW <URL: http://www.visual-paradigm.com/product/vpuml/ > [cit. 2007-07-07]
[7] Schmuller, J.: *Myslíme v jazyku UML*. Grada: Praha 2001, 359 str., ISBN: 80-247-0029-8
[8] Klement, J., Kubík, A., Lanhsrčík, I., Mikulecký, P.: *Tvorba expertních systémů v prostředí CLIPS – podrobný průvodce*. Grada: Praha 1999, 252 str., ISBN: 80-7169-501-7
[9] *Artificial intelligence* [on-line]. Dostupné z WWW <URL: http://wn.wikipedia.org/wiki/Artificial_intelligence > [cit. 2007-07-16]
[10] *A. L. I. C. E. Artificial Intelligence Foundation* [on-line]. Dostupné z WWW <URL: http:www.alicebot.org > [cit. 2007-07-16]
[11] Mařík, V. a kol.: *Umělá inteligence 2*. Academia: Praha 1997, 373 str., ISBN: 80-200-0504-8
[12] *Protege2000* [on-line]. Dostupné z WWW <URL: http://protege.stanford.edu/ > [cit. 2007-07-17]
[13] *Domácí stránky CLIPS* [on-line]. Dostupné z WWW <URL: http://www.ghg.net/clips/CLIPS.html > [cit. 2007-07-17]
[14] *Domácí stránky CLIPSTab* [on-line]. Dostupné z WWW <URL: http://protege.stanford.edu/plugins/CLIPSTabPages/CLIPS_tab.html > [cit. 2007-07-17]
[15] Pavel Šenovský – *Domácí stránky* [on-line]. Dostupné z WWW <URL: http://homen.vsb.cz/~sen76/ > [cit. 2007-07-21]
[16] Novák, M. et. al.: *Umelé neuronové sítě teorie a aplikace*. C.H.Beck: Praha 1998, 382 str., ISBN: 80-7179-132-6
[17] Mařík, V. et. al.: *Umělá inteligence 1*. Academia: Praha 1993, 264 str., ISBN: 80-200-0496-3
[18] *Matematicky odvoďte a vysvětlete algoritmus backpropagation (zobecněné delta pravidlo) učení vícevrstevné sítě* [on-line]. Dostupné z WWW <URL: http://netreisen.cz/bery/stanice07/cabal/Denaci/INF_12.doc > [cit. 2007-07-23]
[19] *Domácí stránky SciLab* [on-line]. Dostupné z WWW <URL:

http://www.scilab.org > [cit. 2007-07-23]

[20] *Domácí stránky projektu NetLogo* [on-line]. Dostupné z WWW <URL: http://ccl.nortwestern.edu/netlogo/ > [cit. 2007-07-27]

[21] Mařík, V. et al.: *Umělá inteligence 3*. Academia: Praha 2001, 328 str., ISBN: 80-200-0472-6

[22] Šenovský, P.: *Buněčné automaty 1*. [on-line]. Dostupné z WWW <URL: http://homen.vsb.cz/~sen76/programy/vb/ca.cab > [cit. 2007-07-28]

[23] Wikipedie: *Cellular Automata* [on-line]. Dostupné z WWW <URL: http://en.wikipedia.org/wiki/Cellular_automata > [cit. 2007-07-28]

[24] Wikipedie: *L-system* [on-line]. Dostupné z WWW <URL: http://en.wikipedia.org/wiki/L-system > [cit. 2007-07-28]

[25] Kubík, A.: *Inteligentní agenty – tvorba aplikačního software na bázi multiagentních systémů*. Computer Press: Brno 2004, 280 str., ISBN: 80-251-0323-4

[26] *Domácí stránky Repast – Recursive Porus Agent Simulation Toolkit* [on-line]. Dostupné z WWW <URL: http://repast.sourceforge.net > [cit. 2007-08-02]

[27] *Domací stránky Swarm* [on-line]. Dostupné z WWW <URL: http://www.swarm.org > [cit. 2007-08-02]

[28] *UMBC AgentWeb* [on-line]. Dostupné z WWW <URL: http://www.cs.umbc.edu/kqml/ > [cit. 2007-08-02]

[29] *FIPA Agent Communication Language Specifications* [on-line]. Dostupné z WWW <URL: http://www.fipa.org/repository/aclspecs.html > [cit. 2007-08-02]

[30] *FGA: Graphical TSP solver* [on-line]. Dostupné z WWW <URL: http://fga.sourceforge.net/tsp_graphic.html > [cit. 2007-08-03]

[31] *FGA – Fast Genetic Algorithm* [on-line]. Dostupné z WWW <URL: http://fga.sourceforge.net > [cit. 2007-08-03]

[32] *Programovací jazyk Karel* [on-line]. Dostupné z WWW <URL: http://karel.webz.cz/ > [cit. 2007-08-03]