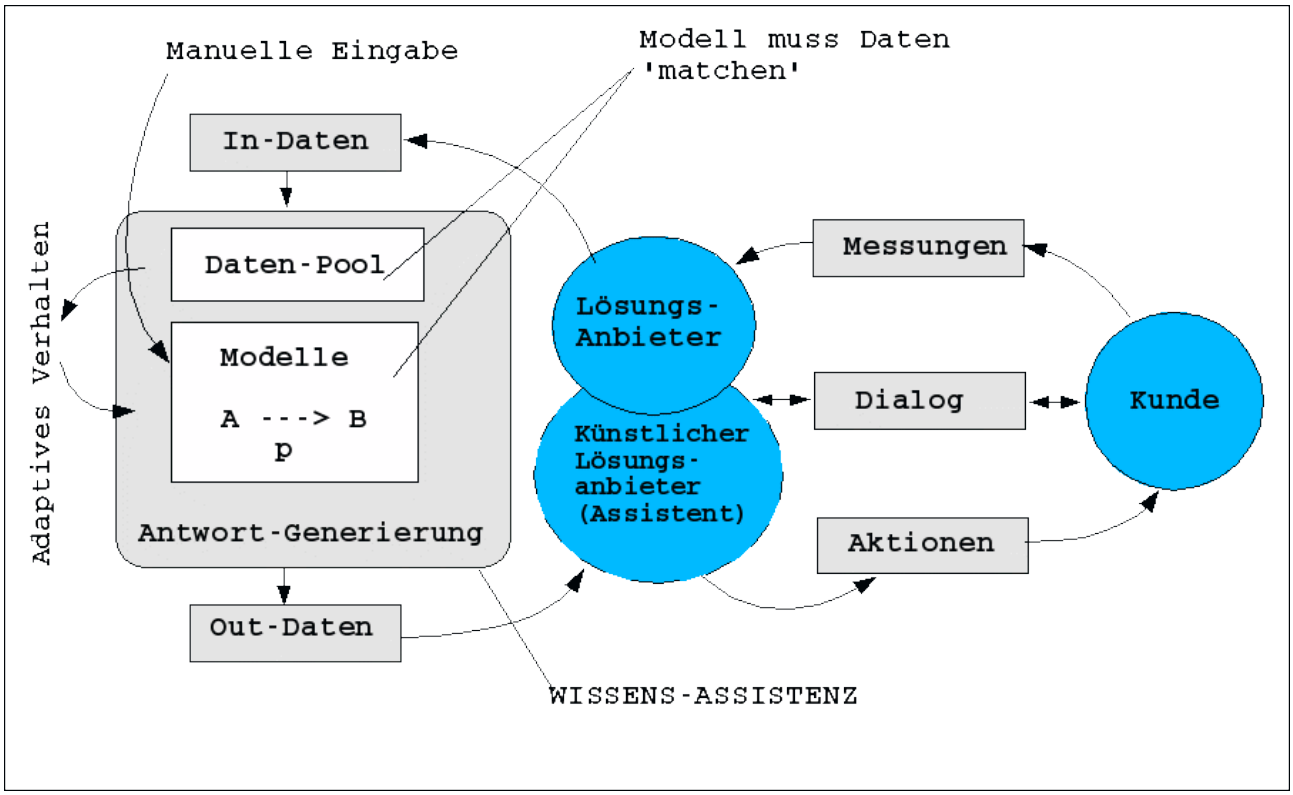
Simulation von Wissen - Ein Modellierungsbeispiel

1. Einführung

In der letzten Sitzung hatten wir eine erste Skizze von einem möglichen Szenario mit einem recht allgemein einsetzbarem *Wissens-Assistenten* entwickelt (siehe Schaubild).



In der heutigen Sitzung soll damit begonnen werden, zu untersuchen, inwieweit sich diese Vision eines Wissens-Assistenten mit den Mitteln der Informatik umsetzen laesst. Wir beginnen mit dem Beispiel einer klassischen Expertensystem-Shell, genannt *CLIPS* (siehe: http://www.ghg.net/clips/CLIPS.html) (Anleitung zur Installation von CLIPS unter Linux im Anhang dieses Textes).

2. Expertensysteme

Das Motiv, die Überlegungen mit Expertensystemen zu beginnen, liegt einmal in der Sache begründet --erklärende *Modelle* lassen sich auch als *Regelmengen* auffassen-- zum anderen gibt es einen historischen Grund, da mit den Expertensystemen die computerbasierte Wissenstechnologie zum ersten Mal eine grössere Ressonanz auch in kommerziellen Anwendungen gefunden hatte (vgl. die kurze historische Übersicht in [D.KARAGIANNIS/ R.TELESKO 2001:25ff].

Bisher hatten wir gesagt, dass *Fakten* alleine kein *Wissen* bilden. Einzelne Worte wie '60318', 'Nibelungenplatz', 'Frankfurt am Main', '1', 'Fachhochschule' für sich alleine genommen sagen nichts oder 'alles'. Wenn man aber weiss, dass eine Postanschrift gewöhnlich aus dem *Muster* besteht

Name des Empfängers	
Strasse	Hausnummer
Postleitzahl	Ort

und dass sich eine Zuordnung herstellen lässt der Art

Fachhochschule	
Nibelungenplatz	1
60318	Frankfurt am Main

Dann *bekommen* die einzelnen Fakten plötzlich eine *Bedeutung*. Sie erscheinen als *Elemente einer Beziehung*, mathematisch gesprochen als Elemente einer *Relation*. Man könnte auch schreiben

$$\text{ADRESSE} \subseteq \text{Name} \times \text{Strasse} \times \text{Hausnummer} \times \text{Postleitzahl} \times \text{Ort}$$

(Für Grundbegriffe der Mengenlehre siehe z.B. Basisbegriffe der Mengenlehre)

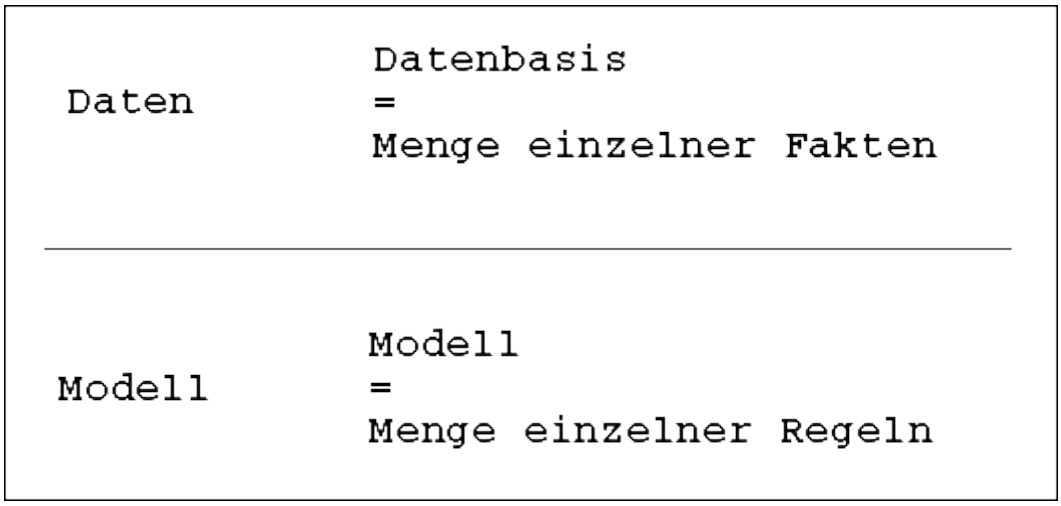
Für Expertensysteme benutzt man spezielle Fälle von Regeln, nämlich sogenannte *Relationen.* dies sind *interpretierte Regeln* der Art

'Wenn BEDINGUNG erfüllt ist, dann soll FOLGE gelten'

BEDINGUNG ==> FOLGE

Unter BEDINGUNG versteht man dann allgemein eine Kombination von *Aussagen*, die *wahr* sein können, und als FOLGE eine andere Kombination von *Aussagen*, die dann *wahr* sein sollen. Die Wahrheit der Aussagen im Bedingungsteil einer Regel wird durch Bezugnahme auf eine vereinbarte *Datenbasis DATA* geklärt. Eine solche Datenbasis enthält eine Menge von einzelnen Aussagen, die *Fakten* beschreiben. Jede Aussage in der Datenbasis wird als *wahr (TRUE)* angenommen. In der Regel vereinbart man zusätzlich, dass ein einzelnes abgrenzbares *Faktum,* das *wahr* oder *falsch* sein kann, als eine *atomare Aussage* gelten soll und *logische Kombinationen* von solchen atomaren Aussagen komplexe Aussagen sind, deren *Wahrheitswert* sich aus den einzelnen Bestandteilen und der *Art der Verknüpfung* errechnen lässt. Kommen nun alle Aussagen einer Regelbedingung in der Faktenbasis vor, dann gilt die Bedingung der Regel als *erfüllt*. Ist dies der Fall, dann werden alle Aussagen im Folgeteil der Regel *wahr gemacht*, d.h. entweder, dass diese Aussagen zu Fakten gemacht werden oder dass diese Aussagen einfach nur *ausgeführt* werden. Bei den Aussagen im Folgeteil einer Regel spricht man daher auch meist von *Aktionen*.

Das grundsätzliche Schema für Expertensysteme ergibt sich also dann wie folgt:



Dabei ist zu ergänzen, dass die Regeln in dem Sinne auf die Fakten angewendet werden, dass in

Abhängigkeit von vorhandenen Fakten bestimmte Regel in ihrem Bedingungsteil zutreffen/ erfüllt sind und deswegen ihr Folgeteil/ ihre Aktionen zur Ausführung kommen.

Die spannende Frage hier ist nun, ob Expertensysteme, die nach solch einem Muster gestrickt sind, in der Lage sind, einen interessanten Beitrag für die Modellierung von Wissens-Assistenzen leisten können. Insbesondere interessiert auch die Frage, ob und inwieweit sich diese Modelle eignen, um *adaptive* Strukturen zu modellieren. Minimale Voraussetzungen für adaptive --sprich: lernende-- Strukturen ist, dass diese sich unter dem Eindruck bestimmter Ereignisse im *notwendigen Umfang ändern* können.

Mit Blick auf die Grundstruktur von Expertensystemen kann man sagen, es gibt grundsätzlich zwei Möglichkeiten, Änderungen zu realisieren. (i) man ändert die Faktenbasis und/ oder (ii) man ändert die Regelmenge. Im folgenden soll an dem konkreten Beispiel der CLIPS-Expertensystem Entwicklungsumgebung geprüft werden, wie diese Fragen beantwortet werden können.

3. Expertensystem-Entwicklungsumgebung CLIPS

Betrachten wir ein einfaches Beispiel mit CLIPS (siehe dazu die Literatur und die Software auf der Eingangsseite des Kurses). Wir benutzen CLIPS mit dem XWindows-Interface unter Linux. Wenn man den Pfad in der .bashrc-Datei richtig gesetzt hat, dann kann man CLIPS starten mit

>xclips

Es wird ein Fenster aufgebaut, in dem man CLIPS-Programme editieren, ablaufen und debuggen kann.

Ds Programm *is-sw-unit3-ex1.clp* enthält zwei Fakten, die mit dem Schlüsselwort (deffacts base1 ...) in einem' Datepaket' mit Namen ' base1' verpackt wurden:

```
;;;======================================================
;;; is-sw-unit3-ex1.clp
;;;
;;;   Einfache Fakten
;;;
;;;======================================================


;;;***************************
;;;* FACTS                   *
;;;***************************

(deffacts base1
  (start a)
  (fortsetzung b))
```

Fakten werden in runden Klammern '*(...)*' geschrieben. Dazwischen können beliebig viele Worte oder Zahlen auftreten bzw. --wie wir später sehen werden-- auch komplexere Ausdrücke ähnlich den 'struct' Elementen aus der Programmiersprache C. Liest man diese Datei mit dem Befehl *(load ...)* ein, dann werden diese Fakten nach einem *(reset)* der aktuellen Datenbasis hinzugefügt. Dabei sieht man, dass automatisch noch ein Fakt f-0 hinzugefügt wird; dies ist der sogenannte ' initial fact' :

```
CLIPS> (clear)
CLIPS> (load "/home/gerd/public_html/fh/IS/SIMULATION-
WISSEN/IS-SW-UNIT3/is-sw-unit3-ex1.clp")
Defining deffacts: base1
TRUE
CLIPS> (reset)
<== Focus MAIN
==> Focus MAIN
==> f-0     (initial-fact)
==> f-1     (start a)
==> f-2     (fortsetzung b)
CLIPS>
```

Im nächsten Beispiel soll eine einfache Regel hinzukommen. Regeln werden in CLIPS wie folgt definiert:

(defrule <rule-name> [<comment>]

[<declaration>] ; Rule Properties

<conditional-element> ; Left-Hand Side (LHS)*

=>

<action>) ; Right-Hand Side (RHS)*

D.h. eine Regel wird durch das Schlüsselwort *(defrule ...)* festgelegt. Es können optional Kommentare und Erklärungen folgen. Am wichtigsten sind die BEDINGUNGS-Elemente, die die *Left-Hand-Side (LHS)* bilden. Diese legen fest, welche Fakten gegeben sein müssen, damit die FOLGE-Elemente, die die *Right-Hand-Side (RHS)* bilden, aktiviert werden. Diese Elemente werden auch Aktionen genannt.

betrachten wir wieder ein Beispiel:

```
;;;======================================================
=======
;;; is-sw-unit3-ex2.clp
;;;
;;;   Einfache Fakten und Regeln
;;;======================================================
=======
;;;***************************
;;;* FACTS                   *
;;;***************************
(deffacts base1
  (start a)
  (fortsetzung b))
;;;*********************************
;;; RULES
;;;*********************************
(defrule aaa
   (start ?x)
=>
   (assert (start b))
   (printout t "rule aaa " ?x crlf)
   )
```

In diesem Beispiel gibt es zusätzlich zu den Fakten aus Bsp.1 eine Regel mit Namen *aaa*. Der BEDINGUNGSTEIL der Regel besteht nur aus einem Element, nämlich dem Fakt *(start ?x)*, der die Variable *?x* enthält. Dies bedeutet, wenn es einen Fakt der Art *(start ---),* dann würde das Element, das an der Stelle ' ---steht als Wert der Variablen *?x* gespeichert werden und damit eine erfüllte Bedingung erzeugen. Im vorliegenden Fall gibt es nur den Fakt *(start a)* in der Datenbasis, also würde die Variable *?x* den Wert ' a' annehmen.

Bei Erfüllung des BEDINGUNGSTEILS würden dann die Elemente des FOLGETEILS aktiviert werden. Der Ausdruck *(assert (start b))* beinhaltet die Funktion *assert*, die besagt, dass das nachfolgende Element *(start b)* der aktuellen Faktenbasis hinzugefügt werden soll. Das Element *(printout t "rule aaa " ?x crlf)* besagt, dass die Funktion *printout* ausgeführt werden soll; diese

erzeugt eine Ausgabe auf das device 't', das für 'terminal', also Bildschirm, steht. Der Inhalt dr Augabe ist ein String *"rule aaa "* sowie der Inhalt der Variablen *?x*. Die Ausgabe wird abgeschlossen durech einen Zeilenvorschub *crlf.*

Im xclips-Fenster sieht dies so aus:

```
CLIPS> (clear)
CLIPS> (load "/home/gerd/public_html/fh/IS/SIMULATION-WISSEN/IS-SW-UNIT3/is-sw-unit3-ex2.clp")
$*
TRUE
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (start a)
f-2     (fortsetzung b)
For a total of 3 facts.
CLIPS> (run)
rule aaa a
rule aaa b
CLIPS> facts)
facts
CLIPS> (facts)
f-0     (initial-fact)
f-1     (start a)
f-2     (fortsetzung b)
f-3     (start b)
For a total of 4 facts.
CLIPS> (run)
CLIPS>
```

Nachdem die beiden Fakten *(start a), (fortsetzung b)* der aktuellen Faktenbasis hinzugefügt worden sind, konnte die Regel 'aaa' mittels *(run)* einmal angewendet werden. Aufgrund der

Anwendung wurde die Faktenbasis um das Faktum *(start b)* erweitert. Dies führte dazu, dass die Regel ' aaahochmals angewendet werden konnte. Eine weitere eingabe des Kommandos *(run)* führte zu keiner Anwendung der Regel ' aaamehr; alle möglichen Interpretationen waren erschöpft. Erst nach einem *(reset)* würde die regel ' aa' wieder zur Anwendung kommen.

An diesem kleinen Beispiel kann man schon sehen, dass CLIPS die Möglichkeit bietet, die Datenbasis während des Ablaufs des Programms zu ändern, indem Fakten hinzugefügt werden.

```
;;;******************************
;;;
;;; is-sw-unit3.ex3.clp
;;;
;;; Änderung der Datenbasis durch
;;; Hinzufügen und Entfernen von Fakten
;;;*****************************************
;;;
;;;***************************
;;;* FACTS                   *
;;;***************************
;;;
(deffacts base1
  (start a)
  (fortsetzung b))
;;;*********************************
;;;
;;; RULES
;;;*********************************
;;;
(defrule aaa
   ?z <- (start ?x)
=>
   (assert (start b))
   (retract ?z)
   (printout t "rule aaa has deleted " ?x crlf)
   )
```

In diesem Programm treten zwei neue Konstrukte auf: das Element *?z <- (start ?x)* besagt, dass, wenn es ein Faktum *(start ?x)* gibt, die Adresse von diesem Faktum in der Variablen *?z* gespeichert werden soll. Damit ist dann der BEDINGUNGSTEIL erfüllt und die Aktionen des FOLGETEILS sollen ausgeführt werden. Neben den bekannten Elementen findet sich als neues Element *(retract ?z)*. Dies besagt, dass die Funktion *retract* das Faktum, das durch die Adresse in *?z* markiert wird, aus der Faktenbasis löschen soll. In CLIPSE sieht dies dann so aus:

```
CLIPS> (load "/home/gerd/public_html/fh/IS/SIMULATION-WISSEN/IS-SW-UNIT3/is-sw-unit3-ex3.clp")
$*
TRUE
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (start a)
f-2     (fortsetzung b)
For a total of 3 facts.
CLIPS> (run)
rule aaa has deleted a
rule aaa has deleted b
CLIPS> (facts)
f-0     (initial-fact)
f-2     (fortsetzung b)
For a total of 2 facts.
CLIPS> (run)
CLIPS>
```

Man kann sehen, wie die Regel nacheinander alle Fakten der Art (*start ...)* aus der Faktenbasis entfernt; indirekt neutraliiert sich die Regel damit.

Die oben gestellte Frage bzgl. der möglichen *Lernfähigkeit* von Expertensystemen kann man also sagen, dass CLIPSE mindestens bzgl. der Faktenbasis veränderungsfähig und damit prinzipiell lernfähig ist.

Bzgl. des Aspektes der Veränderung der Regelbasis ergibt sich allerdings, dass CLIPSE zwar mittels des Befehls (defrule aaa ... *(undefrule bbb ...) ...)* zuvor definierte Regeln aus der aktiven Regelbasis entfernen kann, es kann aber nicht im Rahmen einer Regeldefinition *(defrule aaa... (defrule bbb...) ...)* eine neue Regel definieren. Hier liegt eine gewisse Assymmetrie vor.

Es stellt sich damit die grundsätzliche Frage, was dies für die prinzipielle Modellierbarkeit einer Adaptivität mittels Expertensystemen bedeutet. Diese Frage soll in der nächsten Sitzung weiter geklärt werden.

4. Übungsaufgabe

Als Übungsaufgabe kann versucht werden, ausgehend von dem heutigen Studenten-Vortrag zum Problem Nr. 3 (Arzt und Patient) zu überlegen, wie man die Aufgabenstellung (i) Erstellung eines Diagnosemodells und (ii) Erstellung eines Therapiemodells sowie (iii) Erstellung eines Evaluationsmodells mittels CLIPS angehen könnte.

Anhang: Installation von CLIPS unter Linux

Für die Installation von CLIPS unter Linux (getestet wurde SuSe 8.2 oder 9.0) gibt es neben der Möglichkeit, die CLIPS-Quellen direkt von der CLIPS-Homepage zu laden, auch die einfache Möglichkeit, die Datei

clips_transfer.tar.gz

aus dem Verzeichnis

http://www.fbmnd.fh-frankfurt.de/~doeben/IS/SIMULATION-WISSEN/IS-SW-UNIT3/

auf den eigenen Rechner zu laden. Dann entpackt man die Datei mit dem Befehl:

tar zxvf clips_transfer.tar.gz

Dadurch wird ein neues Verzeichnis mit Namen

clips_transfer

angelegt. Dann wechselt man mit dem Befehl

cd clips_transfer

in dieses Verzeichnis. Hier gibt es das Makefile

makefile.x

In diesem Makefile werden die Pfadangaben für die X-Windows Bibliotheken und Include-Dateien

gesetzt (siehe Kasten). Falls diese Pfade auf dem eigenen Rechner anders lauten, muss man die Pfadangaben in der Datei makefile.x entsprechend mit einem Editor ändern. Ist dies erledigt, dann kann man mit dem Befehl

make -f makefile.x

den Quellcode kompilieren. Wenn keine Fehler auftreten, dann wurde die Objektdatei

xclips

```
#----------------------------------------------------
# The paths to the X libs and include files should
# be changed to whatever appropriate for the
system
# that xclips color are made.
# ----------------------------------------------------


WHERE_XLIBS_ARE =       /usr/X11R6/lib
WHERE_INCL_FILES_ARE =  /usr/X11R6/include
....
```

erzeugt. Das ist die ausführbare Zieldatei. Man kann dieses Programm dann entweder direkt aus diesem Verzeichnis starten mit

./xclips

oder man macht sich die Mühe, und man erstellt mit dem Editor in der .bashrc-Datei des Heimatverzeichnisses eine Pfadangabe der Art:

PATH=$PATH:~/Pfad zum Programm/

export PATH

Ab der nächsten Anmeldung beim System (Sie loggen sich ein...) kennt dann das System diesen Pfad und Sie können das Programm xclips von jeder beliebigen Stelle aus starten durch Angabe von

xclips

Am einfachsten ist die Arbeit, wenn man xclips in dem Verzeichnis startet, in dem man die Dateien hat, die man bearbeiten möchte; dann entfällt das lange Herumsuchen in Verzeichnisbäumen.