COORDINATORE DEL COMITATO
DI PROGRAMMA

Prof. ***Vito Leonardo Plantamura***
Istituto di Scienze dell'Informazione
Facoltà di Scienze
Università degli Studi di Bari
Campus - Via G. Amendola, 173 - 70126 BARI
Tel. (080) 243260

COORDINATORE DEL COMITATO
ORGANIZZATORE

Dott. ***Paolo Sigillo***
NETSIEL S.p.A.
Via S. Dioguardi, 1 - 70124 BARI
Tel. (080) 5762111

SEGRETERIE CONGRESSUALI

Segreteria Scientifica - Congresso A.I.C.A.
Istituto di Scienze dell'Informazione
Facoltà di Scienze
Università degli Studi di Bari
Campus - Via G. Amendola, 173 - 70126 BARI
Tel. (080) 243272 - Telefax (080) 243196

Segreteria Organizzativa - Congresso A.I.C.A.
Centro Internazionale Congressi
V.le Papa Pio XII, 18 - 70124 BARI
Tel. (080) 517299 - Telefax (080) 514533

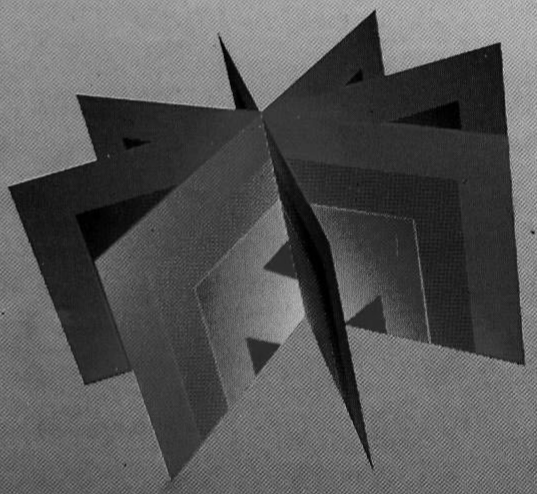
SEDE E SEGRETERIA A.I.C.A.

A.I.C.A.
Piazzale R. Morandi, 2 - 20121 MILANO
Tel. (02) 784970

Editrice Safra srl Bari

CONGRESSO ANNUALE
ANNUAL CONFERENCE

Programma
Program

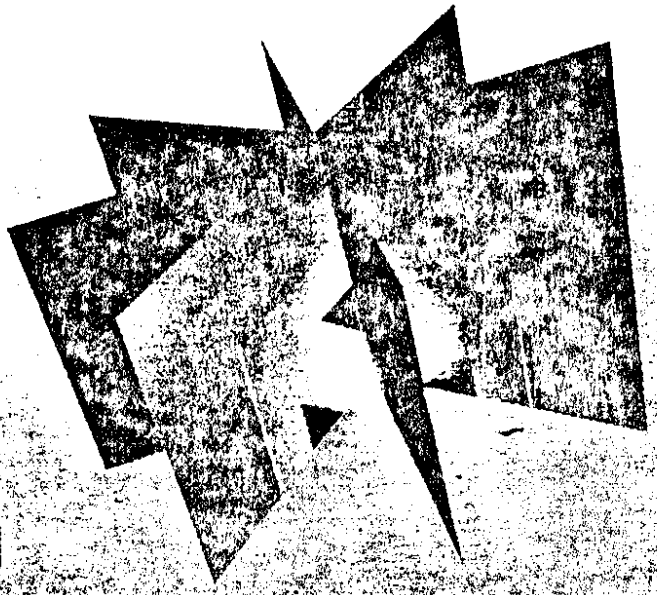


Bari
19-21 settembre 1990

Associazione Italiana
per l'Informatica
ed il Calcolo Automatico
A.I.C.A.

CONGRESSO ANNUALE
ANNUAL CONFERENCE

ATTI
PROCEEDINGS



Volume 1

Associazione Italiana per l'Informatica ed il Calcolo Automatico
AICA

ARCHITETTURA DI UN SISTEMA ESPERTO IN PROLOG

A.Gisolfi, W.Balzano, M.Palladino
Dipartimento di Informatica
Università di Salerno (Italia)

SOMMARIO

In questo lavoro è presentato uno Shell di Sistema Esperto capace di gestire conoscenza incompleta e di operare sia con un ragionamento backward che forward. Questo sistema produce non risposte ma soluzioni dettagliate. L' interfaccia, semplice e lineare, permette una facile interazione con il Sistema.

Le componenti del Sistema Esperto che tratteremo in questo articolo sono il modulo generatore delle inferenze (modulo "Forward"), un modulo preposto al rilevamento delle informazioni mancanti, un modulo per l'acquisizione della conoscenza ed un modulo per l'elaborazione della conoscenza. Per l'ambiente di sviluppo modulare è stato scelto il Prolog; in particolare ci siamo serviti della versione ARITY PROLOG 5.1 (1989).

INTRODUZIONE

In un mondo in cui è sempre più sentita l' esigenza dello scambio di informazioni attraverso reti sempre più complesse diviene necessario fissare e standardizzare protocolli di comunicazione, strutture, sistemi operativi, ecc.. Anche se finora è stato fatto molto per lo sviluppo modulare del software, poco e molto poco è stato fatto, lungo tali direttive, per ciò che concerne alcuni settori applicativi dell' Intelligenza Artificiale in campo didattico.

Oggetto della nostra attenzione è stata la progettazione di una complessa ed articolata struttura, quale quella di uno Shell di Expert System. Abbiamo pertanto ritenuto basilari tutte quelle componenti di progetto che potessero contribuire alla caratterizzazione di modularità e modificabilità dell' intera struttura. La strategia di progetto da noi seguita è di tipo misto, ovvero top-down/bottom-up il cui preciso intento è stato quello di massimizzare la generalizzazione per ogni risultato conseguito, cercando così di rendere la struttura "aperta" e predisposta ad acquisire via via tutte quelle componenti o moduli ed ogni altro tipo di contributi che saranno ritenuti necessari.

La tecnica di programmazione da noi adottata è quella oramai classica della *metainterpretazione*, ovvero la prerogativa e facilità con cui il linguaggio Prolog permette e garantisce la sua autointerpretazione.

STRUTTURA DEL GUSCIO E ALCUNI REQUISITI

Dei prototipi realizzati, pochissimi sono stati seriamente sperimentati e talune esperienze hanno prodotto risultati sorprendenti, per non dire deludenti. Si pensi ad esempio alla serie di prove comparative condotte da Derek Sleeman sul sistema PIXIE, capace di diagnosticare errori commessi sul calcolo di semplici operazioni algebriche sulla base di un ricchissimo catalogo di "misconceptions". I risultati hanno portato alla conclusione che PIXIE non sembrava contribuire al recupero di tali abilità più di quanto lo possa una semplice spiegazione del "come si può fare" del tutto indipendente dal tipo di errore commesso. Tale indipendenza dal dominio oggetto è, pertanto, la prerogativa basilare su cui fondare la progettazione di un guscio.

Prima di descrivere i moduli che andranno inseriti nella suddetta struttura, riteniamo opportuno spendere giusto qualche parola circa la struttura che dovrà supportarli. La letteratura è oramai ricca di proposte di architetture di Expert System, talvolta contrastanti, ma il modello più accreditato è certamente quello in cui ritroviamo il Modulo Esperto costituito dalla Base di Conoscenza e dallo stesso meccanismo di inferenza.

Più che descrivere tali componenti, crediamo ora proficuo evidenziare le differenze di questo tipo di struttura con quella di un Sistema Tutoriale Intelligente. Una delle differenze di base consiste nel fatto che il "carico di lavoro" dei due sistemi è sostanzialmente diverso: mentre nel Sistema Esperto ritroviamo soltanto la conoscenza del dominio oggetto, nel Sistema Tutoriale Intelligente coesiste, in maniera dinamica con la suddetta conoscenza, la conoscenza di chi utilizza la "conoscenza oggetto". Ciò avvalora l' ipotesi di quanti vedono un Sistema Esperto più propriamente come un sottosistema di un Sistema Tutoriale Intelligente (fig.1).

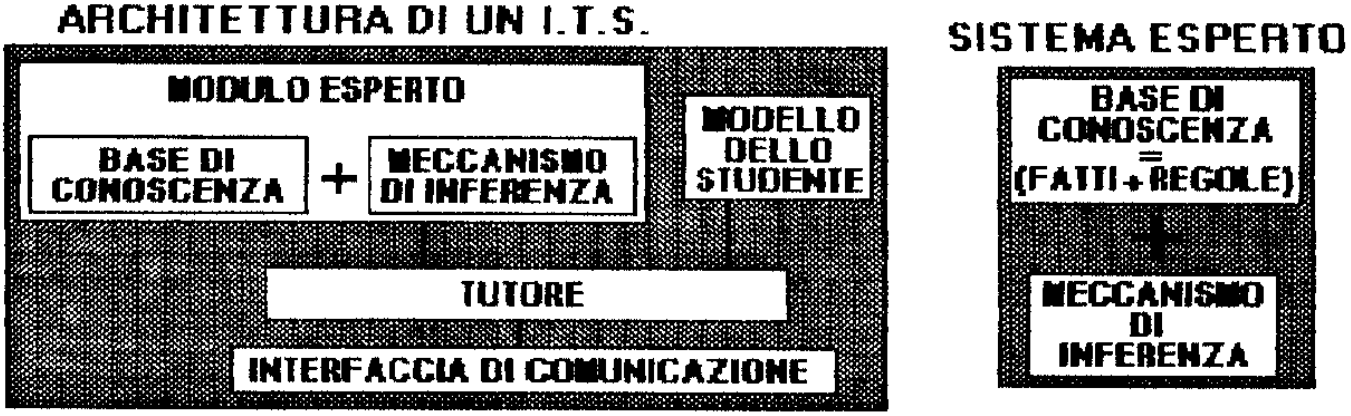


fig.1

Alcuni requisiti che dovranno essere considerati per la strutturazione del Modulo Esperto del guscio sono: **1)** giustificare il processo inferenziale **2)** gestire dati incerti **3)** manipolare dati incompleti. In un ambiente di lavoro quale quello offerto dal Prolog viene generalmente offerto solo un sottoinsieme limitato di strumenti e primitive che permettono però, mediante opportuni artifici (quale quello della metainterpretazione), di simulare le prerogative richieste a un dato sistema.

Analizziamo adesso più in dettaglio, relativamente all' ambiente di lavoro da noi prescelto, ovvero il Prolog, quali siano gli altri requisiti che il nostro Shell dovrà avere, giustificando così le scelte da noi adottate in seguito.

Abbiamo accennato che una delle capacità del Modulo Esperto del nostro guscio è quella di essere in grado di effettuare un ragionamento di tipo forward (dagli stati iniziali verso gli obiettivi), di tipo backward (dagli obiettivi agli stati iniziali), e di tipo misto. Risulta senz' altro più interessante analizzare le modalità con cui è possibile implementare i primi due tipi di ragionamento, in quanto il terzo tipo, quello misto, potrà essere ottenuto applicando opportune euristiche ai due primi tipi (che, in quanto tali, possono essere definiti "primitivi"). Ciò che dovrà esser esplicitamente tenuto in considerazione, prima di dover scegliere uno dei due tipi di ragionamento primitivi, dovrà esser il cosiddetto "fattore di ramificazione", cercando così, là dove possibile, di evitare la cosiddetta esplosione combinatoria. Il ragionamento di tipo misto sarà invece efficace soltanto se i due processi applicati parallelamente agli stati finali ed iniziali riusciranno ad accorciare progressivamente la distanza tra stati iniziali e conclusioni, convergendo in un punto intermedio. Molte di tali caratteristiche dovranno essere simulate, poiché lo "standard Prolog" prevede soltanto uno dei citati tipi di ragionamento: quello backward. Il nostro obiettivo è quindi quello di simulare uno Shell utilizzando uno Shell preesistente (l' ambiente di lavoro offerto dal Prolog). Focalizziamo pertanto la nostra attenzione su una semplice e potente struttura che permetta tale effettiva simulazione:

```
solve(GOAL)
solve(true).
solve((A,B)) :- solve(A),solve(B).
solve(A) :- clause(A,B),solve(B).
```

Questa struttura scritta in codice Prolog, rappresenta il **modello computazionale dei programmi logici** e, allo stesso tempo, costituisce il cardine per la stesura di un

metainterprete. Esso rende infatti trasparente il processo di unificazione, evidenziando la granularità d' esecuzione, permettendo così l' accesso ai singoli "pezzi" di computazione.

MODULO INSERIMENTO

Lo scopo del modulo Inserimento è di gestire basi di conoscenza, permettendo ad esperti non informatici di creare e modificare Basi di Conoscenza. Come il Sistema Autore "TRAS" (Tutoring Rule Authoring System) esso è formato da diversi sottomoduli:

a- sottomodulo Scelta Base
b- sottomodulo Pseudo Editor
c- sottomodulo Traduttore
d- sottomodulo Controllo

Il primo sottomodulo ha il compito di interfacciare l'utente nella scelta della base da editare. Esso rappresenta, quindi, un sottomodulo specificamente di interfaccia e per tale ragione sarà esaminato quando si parlerà di architettura esterna. In ogni caso si vuole osservare che basi di conoscenza create non utilizzando il modulo inserimento, non possono essere editate mancando per esse le strutture necessarie. Il secondo sottomodulo svolge il compito di Editor (-TRE- Tutoring Rule Editor, in TRAS), ovvero permette l'inserimento materiale delle informazioni costituenti la base di conoscenza. Il formalismo di rappresentazione della conoscenza è quello a regole di produzione anche se la forma è sensibilmente vicina al Prolog; ogni elemento di una regola, infatti, è del tipo:

funtore(arg1,arg2,...,argN)

il che consente una più semplice gestione della conoscenza nell'ambiente Prolog in cui ci troviamo ad operare.

Le strutture fondamentali sono tre e sono identificate dal nome della base di conoscenza opportunamente estesa.

Ad esempio, se il nome della base è PROVA, le tre strutture create sono:

PROVA.BAS
PROVA.LIS
PROVA.ARI

La terza struttura rappresenta il codice Prolog della base inserita; essa in effetti è l'output del sottomodulo traduttore. La seconda e la prima struttura sono sensibilmente più importanti per l'utilizzo completo del modulo inserimento. Esse sono aggiornate dinamicamemte all'introduzione di ogni singola regola o fatto. PROVA.BAS è una rappresentazione interna delle informazioni inserite dall'utente ad esempio se è inserita una regola del tipo :

```
regola i : SE cond1
           cond2
         ALLORA azione
```

e se è inserito un fatto del tipo:

```
fatto N : fatto
```

la corrispondente struttura PROVA.BAS risulta essere

```
regola([i,azione,cond1,cond2]).
fatto([N,fatto]).
```

L'importanza di tale struttura risulta palese se si pensa ad una semplice modifica di una regola; in tale situazione, indicando il numero della regola, si riesce a risalire alla stessa, a modificarla e ad aggiornare le strutture in funzione di tale modifica.

La struttura PROVA.LIS, come vedremo, è importante per il sottomodulo controllo. Essa è formata da due sottostrutture :

lista globale
lista globale1.

La prima contiene informazioni sulla base completa, la seconda informazioni relative alla sessione corrente. Il contenuto di tale struttura è, in ogni caso, un unico oggetto della forma :

lista_globale(Arg)

dove Arg è una lista che rappresenta secondo un formalismo ad albero, la conoscenza inserita fino a quel momento. Ad esempio, corrispondentemente ad una regola e ad un fatto del tipo seguente:

regola :SE cond1(X,Y,Z)
cond2(X,Y)
ALLORA azione(X,Y)
fatto: fatto1(X,Y).

si ottiene una PROVA.LIS del tipo :

lista_globale([fatto1,azione(cond1,cond2)])

in cui fatto1 è un nodo foglia mentre azione(cond1,cond2) rappresenta un sottoalbero del tipo :

Il terzo sottomodulo ha il compito di tradurre in un formalismo Prolog le informazioni inserite (-TPT- Text to Prolog Translator, in TRAS). Una euristica utilizzata in tale sottomodulo è quella di tradurre prima i fatti e poi le regole. Ciò consente in alcuni casi di evitare possibili cicli generati da regole ricorsive che non riescono ad attivare condizioni di stop. A livello implementativo, il problema di riconoscere atomi da variabili (entrambi scritti in lettera minuscola) è superato facendo precedere ogni atomo da un "-".

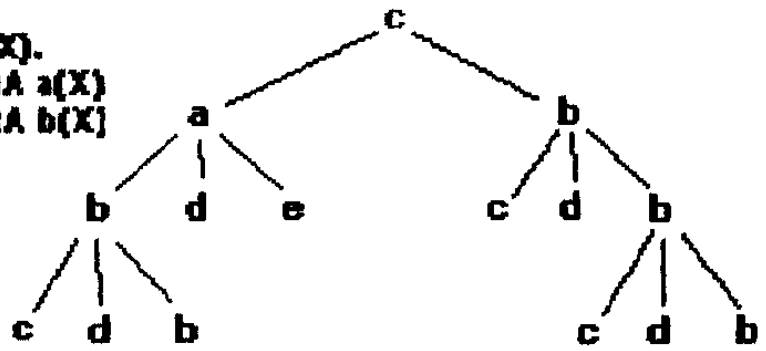
L'ultimo sottomodulo è attivabile direttamente da modo editor. Il suo compito è quello di riconoscere e visualizzare tutti i possibili cicli o le possibili ricorsioni esistenti tra le regole fino a quel momento inserite, lasciando comunque all'utente il compito di controllare l'esattezza delle regole segnalate.

L'algoritmo di fondo è un algoritmo di esaurimento in cui sono analizzati tutti i possibili percorsi.

Vediamo il seguente esempio.

Data una conoscenza del tipo :

```
regola1 : SE a(X),b(X)  ALLORA c(X).
regola2 : SE b(X),d(X),e(X) ALLORA a(X)
regola3 : SE c(X),d(X),b(X) ALLORA b(X)
regola4 : SE a(X) ALLORA d(X).
fatto1:a(X).
fatto2:b(X).
```



la corrispondente PROVA.LIS risulta essere :

lista_globale([a,b,c(a,b),a(b,d,e),b(c,d,b)]).

Il tutto si riduce alla ricerca in un albero rappresentato dalla struttura PROVA.LIS in cui ogni elemento è un nodo con i suoi discendenti. La ricerca effettuata è una ricerca Depth-first. Il cammino indicato individua i diversi passi che l'algoritmo compie. Uno svantaggio di tale sottomodulo è legato alla poca efficienza che caratterizza la ricerca in profondità. Tale svantaggio potrebbe essere superato introducendo particolari euristiche che permetterebbero di esaminare l'albero in modo più intelligente ma, di contro, si perderebbe la possibilità di diagnosticare tutti i possibili cicli. E' stato invece inserito un controllo di profondità in modo da lasciare all'utente la possibilità di continuare o meno la ricerca quando si è giunti oltre un certo limite.

IL MODULO DIMOSTRATORE

Il meccanismo di interpretazione del Prolog non è in grado di dimostrare un certo obiettivo data una base di conoscenza. La sua capacità è limitata alla verifica della veridicità di un certo obiettivo.

Per ovviare a tale problema è stato progettato ed implementato il modulo dimostratore che dà una spiegazione plausibile dei risultati ottenuti. Esso, come il motore interno del Prolog, si basa su un ragionamemto "backward" ma, a differenza di questo, simula una ricerca in profondità in grafi and/or che consente di ottenere esplicitamente un albero completo di soluzione.

Lo schema generale di ragionamento utilizzato dal modulo dimostratore è il seguente:

Se "O" rappresenta la domanda e "D" la sua risposta si ha:

1) SE O è un fatto nel D.B. ALLORA D è vera;

2) SE nella base di conoscenza esiste una regola del tipo "SE Cond allora O" ALLORA si esplora Cond per trovare D;

3) SE O è una congiunzione di obiettivi (O1 and O2) ALLORA si esplora O1; se è falso, D è falso, altrimenti si esplora O2 e si combinano le due risposte per ottenere D;

4) SE O è una disgiunzione di obiettivi (O1 or O2) ALLORA si esplora O1; se è vero, D è vero, altrimenti D assume il valore ottenuto esplorando O2.

Questo ragionamento, in conclusione, deve generare un albero di soluzione AND/OR composto dalle regole e dai fatti attivati e che viene utilizzato da particolari procedure di interfacciamento per consentire un'adeguata spiegazione del risultato ottenuto. Si vuole osservare, in ogni caso, che quanto detto è verificabile solo nel caso di una base di conoscenza completa e nel caso di obiettivi effettivamente deducibili. I casi che non rientrano in tale contesto sono gestiti dal modulo Interattiva discusso successivamente.

Il cuore del modulo è la procedura "solve(X,Y)" riportata di seguito:

```
1) solve(true,true).
2) solve((A,B),(DimA,DimB)):!,solve(A,DimA),solve(B,DimB).
3) solve(A,(A:-true)):-functor(A,S,Arita),
                        invisibile(S/Arita),!,A.
4) solve(A,(A:- true)) :- sistema(A),!,A.
5) solve(A,(A:- Dim)) :- clause(A,B),solve(B,Dim).
sistema(A/=B). ..... sistema(A=B). .....
```

Questa procedura implementa lo schema di ragionamento visto precedentememte ad esclusione del punto 4. Tale scelta è giustificata dal fatto che gli obiettivi delle disgiunzioni sono gestiti singolarmente ed in sequenza in modo relativamente semplice. Ad esempio, nel Data Base la disgiunzione "Obiettivo(i) **or** Obiettivo(j)" è equivalente alla doppia

asserzione "Obiettivo(i)." "Obiettivo(j).". A livello procedurale le clausule formanti il solve hanno il seguente significato:

1- indica che l'obiettivo vuoto, rappresentato in Prolog come l'atomo TRUE è vero con un albero di dimostrazione banale rappresentato da TRUE

2- indica che nel caso in cui l'obiettivo è una congiunzione di obiettivi, essi vengono risolti singolarmemte e l'albero di dimostrazione associato è formato come congiunzione degli alberi di dimostrazione dei singoli sottobiettivi.

3-4 saranno chiariti in seguito.

5- permette di scegliere dalla base di conoscenza una regola la cui testa unifica con l'obiettivo e ricorsivamente risolve il corpo della regola costruendo il corrispondente albero di dimostrazione. Questa rappresenta la clausola in cui si pone in rilievo il tipo di ragionamento backward che caratterizza il modulo. I cut (!) presenti nella procedura sono utilizzati per tagliare rami di eventuali percorsi da esaminare che non possono generare soluzioni desiderabili. Essi sono necessari al fine di migliorare l'efficienza della procedura. Le clausole 3 e 4 hanno uno scopo comune ovvero il trattamento dei predicati conosciuti dall'utente, la differenza sta nel tipo di predicati gestiti. Mentre la clausola 4 prende in esame dei predicati cosiddetti di "sistema", ovvero dei predicati Prolog che, se non esplicitamente definiti, non rientrano nella conoscenza del sistema e che potrebbero generare in fase di esecuzione problemi di interpretazione, la clausola 3 lavora con predicati presenti nella base di conoscenza, inoltre mentre i primi sono definiti in fase di progetto (qualche esempio è dato nello scorcio di programma precedente) i secondi, cosiddetti "invisibili", sono esplicitamente dichiarati dagli utenti nel corso di una sessione. Il risultato di entrambe le clausole è, in ogni caso, di evitare la dimostrazione di predicati conosciuti dall'utente; esse quindi rappresentano un primo passo verso la gestione del cosiddetto "modello studente" che consente, tra l'altro, di adeguare una certa sessione ad ogni singolo utente.

Tale prerogativa è essenziale nei sistemi che hanno come obiettivo l' insegnamento ed è raggiungibile in modi diversi. Il Sistema Esperto "Mathpert" per l' apprendimento della matematica, ad esempio, associa ad ogni operatore matematico un valore del tipo:

"Learning, Know, Well-know, Unknow"

che indica il livello di conoscenza del dato operatore. Da quanto detto risulta chiaro che il secondo argomento del predicato solve rappresenta l'albero di dimostrazione che deve poi essere interfacciato in modo opportuno per fornire all'utente una spiegazione comprensibile. Poco opportuno sarebbe visualizzare la dimostrazione nel modo in cui è fornita dal predicato solve; essa infatti è del tipo:

```
DIM=(Obiettivo :- ((sottobiettivo1 :- dim1),
                   (sottobiettivo2 :- dim2),
                   .....
                   (sottobiettivoN :- dimN)).
```

dove dim1,dim2,...,dimN sono rispettivamente le dimostrazioni dei sottobiettivi 1,2,...,N.

La procedura di interfacciamento più interessante è quella definita "dimostrazione per livelli" con la quale si riesce ad ottenere la dimostrazione di un certo obiettivo per livelli di profondità. Il predicato fondamentale è il seguente:

```
interpreta((Dim1,Dim2),Livello):-
        interpreta(Dim1,Livello), intepreta(Dim2,Livello).
interpreta(Dim,Livello):-fatto(Dim,Fatto),nl,
        write(Fatto), write($è un fatto nel d.b.$).
```

```
interpreta(Dim,Livello) :- regola(Dim,Testa,Corpo,Dim1),
        scrivi(Testa,Corpo),write(esci),
        tget(X,Y),tab(25),write('Livello='),
        write(Livello),
        dim_sottobiettivo(Dim1,X,Livello).
        .....
```

La prima clausola del predicato *interpreta* è applicabile quando la dimostrazione è una congiunzione di dimostrazioni. La seconda quando la dimostrazione è del tipo:

F:- true.

il che implica che F è un fatto nella base di conoscenza. La terza quando è una dimostrazione completa ed in tal caso viene divisa in TESTA, CORPO e DIM1 come risulta dagli argomenti del predicato "regola(X,Y,Z,W)". In base all' esempio generale di dimostrazione visto in precedenza, queste variabili sono instanziate, per effetto dell' applicazione del predicato "regola", in questo modo:

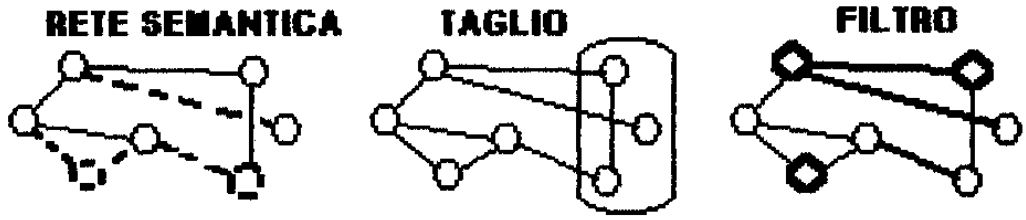
TESTA= obiettivo
CORPO=(sottobiettivo1,sottobiettivo2,...,sottobiettivoN)
DIM1=((sottobiettivo1:- dim1),...,(sottobiettivoN:- dimN)).

Con tale suddivisione si ha la possibilità di visualizzare la dimostrazione al livello desiderato, dando la facoltà all'utente o di scegliere, tra i sottobiettivi dati (corpo), uno da dimostrare e quindi scendere di un livello nell'albero di dimostrazione oppure ritornare con l'opzione "esci" al livello precedente. Nel caso in cui si opta per la prima scelta, occorre cercare in DIM1 l'elemento rappresentante la dimostrazione del sottobiettivo selezionato (il predicato che si fa carico di tale lavoro è "ricerca_dim_con_predicato") ed occorre interpretarlo evidenziando così la ricorsività della procedura. Nel caso in cui si opta per la scelta "esci", invece, si fa fallire il predicato attivato il che comporta il ritorno al precedente grado di ricorsione e quindi si risale l'albero.

IL MODULO DI RISOLUZIONE INTERATTIVA

La componente del Modulo Esperto che ci accingiamo a descrivere in questa sezione sarà utilizzata per monitorare possibili informazioni mancanti associate ad una Base di Conoscenza. Esso è quindi un modulo costruito col preciso intento di non restringere e "chiudere" il Sistema ad un tipo di lavoro che operi esclusivamente sulle informazioni ivi contenute. La necessità che generalmente induce alla realizzazione di un siffatto modulo è da ricercare principalmente nella circostanza che spesso un programma di I.A. non nasce già completo, e lo stesso avviene per la corrente Base di Conoscenza su cui il nostro modulo lavora; sarebbe infatti di scarso interesse pratico realizzare un sistema in cui tutto debba esser necessariamente ben predisposto e perfettamente funzionante. In altre parole, questo modulo servirà per interfacciare il rigore logico della programmazione con l' imprecisione e la negligenza tipiche degli ambienti di apprendimento. Premettiamo ancora che il sistema da noi realizzato è un sistema di produzione commutativo, ovvero un sistema di produzione che è allo stesso tempo monotonico e parzialmente commutativo. Forse la più immediata prerogativa di tale scelta di progetto risulta chiaramente identificabile dal fatto di non dover necessariamente ripristinare lo stato della Base Dati del sistema quando, essendo risultata fallimentare una scelta presa allo stato considerato, occorrerebbe invece tornarvi (ripristinando così le modifiche) per considerare le possibili alternative (backtracking). Non occorrerà pertanto tenere traccia ai cambiamenti che via via avvengono per dover poi ripristinare la Base Dati.

La Risoluzione Interattiva, partendo da queste premesse, vuole ricostruire il processo deduttivo lavorando su dominii in cui non tutta l' informazione sia stata inserita. La ricostruzione dei 'pezzi' mancanti sarà effettuata localizzando "strutture vacanti" ed interagendo con l' utente che potrà colmare o meno le possibili lacune di informazione segnalate dal sistema. Il processo in questione è un processo di tipo deduttivo costruttivo: le nuove informazioni non sono né già nella Base Dati (da esplicitare), né sono aleatoriamente congetturate dal sistema (inferenza induttiva). Analizziamo adesso il retroscena e la logica su cui è basato questo modulo. Anzitutto stabiliamo un isomorfismo tra una generica Base di Conoscenza espressa sotto forma di fatti e regole Prolog ed una rete semantica: in particolare, gli argomenti delle relazioni sono associati agli oggetti o nodi della rete semantica, mentre le relazioni medesime corrisponderanno ai links della rete. Ciò che questa trasformazione ideale evidenzia è la precisa differenziazione di due classi di elementi identificabili nella corrente Base di Conoscenza: Oggetti e Relazioni. Fondamentalmente, gli algoritmi di ricerca ipotizzabili su di una rete semantica possono essere opportunamente ricondotti ad algoritmi di taglio, del tipo "divide et impera", oppure ad algoritmi la cui prerogativa sia quella di operare sull' intera rete, ma utilizzando opportuni filtri (euristiche). La seconda di tali possibilità è quella da noi utilizzata nella procedura di ricerca; il dominio di ricerca dell' informazione mancante viene ad essere automaticamente descritto dalla definizione di un filtro che sia sensibile soltanto a 'quei' rilievi della rete.



La peculiarità di questo secondo approccio consiste nella capacità di monitorare le strutture delle informazioni mancanti direttamente nel loro ambiente di vita (la rete) senza dover necessariamente isolare questa o quella regione. Il processo di ricerca è guidato, poi, dallo stesso processo di soddisfacimento di un dato obiettivo. Sappiamo che il motore inferenziale incapsulato nell' interprete Prolog tenta di soddisfare un assegnato obiettivo espandendo (in modo backward a partire dal considerato nodo obiettivo) il nodo in albero le cui foglie coincidano con stati iniziali (fatti nel Data Base). Tale processo potrà bloccarsi se durante la visita dei nodi dell' albero sarà incontrato un nodo dal quale non possa esser generato l' insieme dei successori e a cui non corrisponde alcun fatto nel Data-Base: tale nodo è fatto oggetto della nostra attenzione perché ad esso potrebbe corrispondere un pezzo di informazione mancante. A questo punto, se è possibile per tale nodo una allocazione che risulti compatibile con i rimanenti sottobiettivi, verrà allora prodotta un' interazione con l' utente, permettendo a quest' ultimo di colmare la lacuna individuata. La compatibilità con i rimanenti sottoobiettivi garantisce la opportunità di una eventuale interruzione. Se infatti fosse originata una interazione avendo effettuato soltanto una istanziazione parziale dei sottoobiettivi, come nell' esempio prodotto da Sterling Shapiro, verrebbe a stabilirsi una eccessiva dipendenza delle interruzioni (del processo di ricerca) dal particolare ordine della sequenza dei sottoobiettivi, rendendo, allo stesso tempo, non strettamente necessarie le interazioni prodotte. In ogni caso il backtracking, durante l' intero processo di ricerca, non dovrà ripristinare gli stati della Base Dati, in quanto abbiamo supposto che il sistema in oggetto sia un sistema di

produzione commutativo. Il nucleo della procedura che controlla tale processo è il seguente:

```
solve(true,true).
solve((A,B),(DimA,DimB)) :- !, solve(A,DimA),
                              solve(B,DimB).
solve(A,(A:-true)) :- sistema(A),!,
                      (not insensata(A)),A.
solve(A,(A:-true)) :- test(A).
solve(A,(A:-Dim)) :- scegli_clausola(A,B),
                     solve(B,DimB).
```

Occorre osservare che la relazione *solve* questa volta è definita su due argomenti: il primo è stato utilizzato per ottemperare al processo di unificazione cui precedentemente accennato; il secondo argomento è necessario per "tenere traccia" dell' intero processo. L' operatore che individua la struttura a cui potrebbe essere associata l' informazione mancante è definito dal corpo della quarta regola, costituita dalla relazione «test(A)». Notiamo esplicitamente che in questa versione specializzata di metainterprete non compare il predicato predefinito clause(A,B): esso è stato sostituito con il nuovo predicato da noi definito «scegli_clausola(A,B)». La particolarità di questo nuovo predicato consiste nel fatto che esso unificherà autonomamente la testa A di una data regola (ovvero la parte azione) con quella regola il cui corpo B sia costituito dal minor numero di congiunzioni non note. Questo predicato, di fatto, permetterà la vera e propria realizzazione dell' euristica per la "scelta delle regole" nella loro concatenazione secondo il processo di backward chaining.

L' effetto di tale euristica conduce il processo a ricercare prima quelle dimostrazioni o ragionamenti a cui è associata una quantità minima di informazioni mancanti. Minimizzare la quantità di nuova informazione significa, in altre parole, voler scommettere sulla bontà della Base Dati: pochi (almeno si spera) possono essere stati gli elementi trascurati nella fase in cui è stata introdotta la conoscenza (fig.2).

ESEMPIO

R1: SE e(X),q(X),f(X) ALLORA a(X)
R2: SE d(X) ALLORA a(X)
R3: SE b(X),q(X),f(T),f(Z) ALLORA a(X)
R4: SE c(X),g(X),h(Z),e(Y) ALLORA a(X)

FATTI: g(5),h(8)

(una buona approssimazione per a(5) è ottenuta utilizzando R2)

fig.2

Un ulteriore effetto altrettanto importante è ottenuto da questa euristica: essa cioè funge da strategia tutoriale. Vengono infatti apprese (ricostruite) prima le informazioni che sono meno "distanti" dalla conoscenza attuale (Base Dati). Osserviamo che l' assunzione di lavorare su basi di conoscenza con informazioni mancanti fa "allargare" il dominio della ricerca di una soluzione. Ciò non sarebbe avvenuto se avessimo supposto di operare in un "mondo chiuso" che si rifà al principio secondo cui ciò che non si vede è falso.

La precedente strategia tutoriale ed una opportuna condizione per il controllo dei limiti della ricerca sono racchiusi nella seguente procedura:

```
scegli_clausola(A,B) :-
        findall(M,
                (clause(A,Bip),
                conta_congiunzioni_incognite(Bip,N),
                M = c(N,corp(Bip))),
            BB),
        sort(BB,Q),
        length(Q,N2),ifthen(N2 = 0,(!,fail)),
        !,
        controllo_stop,
        !,
        for(1,N2,X),
        posizione2(Q,X,Ber),
        Ber = c(_,corp(B)),clause(A,B).
```

In ogni sistema complesso in cui sia stata posta al centro dell' attenzione l' interazione uomo-macchina occorrerà, inoltre, prevedere un insieme minimale di primitive e direttive che possano esser facilmente utilizzate dall' utente al fine di rassicurarlo sulle scelte che il sistema opera durante l' arco della ricerca. Ciò perché crediamo che in un Sistema Tutoriale Intelligente l' obiettivo primario sia quello di trasmettere all' utente la capacità di gestire, manipolare e comprendere l' insieme di operatori che intervengono nella soluzione del problema, più che il problema in se stesso. A tal fine, il modulo di Risoluzione Interattiva è stato opportunamente corredato di alcune opzioni, il cui utilizzo fornirà un' accurata indagine sulle cause che hanno indotto il sistema allo stato corrente. Alcune delle opzioni a cui facciamo implicitamente riferimento sono quelle che permettono di ispezionare la traccia e l' albero completo di una determinata struttura logica.

IL MODULO FORWARD

In questo paragrafo analizziamo il modulo il cui compito è quello di gestire il processo di generazione delle inferenze definito, nel nostro sistema, "modulo Forward". Al fine di meglio comprendere in che modo e perché siano state operate talune scelte, occorre preliminarmente analizzare un modello di computazione parallela. Stabiliremo poi un preciso isomorfismo del suddetto mudulo con un Sistema Basato su Regole. Ciò permetterà di identificare ogni singola regola con un modulo Pattern-Directed. La ragione di questo isomorfismo è palesata dalla necessità di dover conciliare la filosofia intrinsecamente parallela di un sistema di produzione basato su regole con la filosofia intrinsecamente sequenziale della maggioranza delle macchine disponibili sul mercato.

Una struttura software può essere schematicamente vista come l' insieme di 'oggetti' che interagiscono tra loro in maniera più o meno articolata, prefissata e prevedibile. In un linguaggio di programmazione tradizionale, ciò viene a tradursi in un insieme di procedure che continuamente hanno il potere di attivare altre procedure a cui trasferiranno il controllo; in ogni istante, quindi, una sola procedura è attiva e l' attivazione di altre procedure dipende esclusivamente da quella che allo stato attuale ha il controllo.

Tutto ciò conferisce alla considerata struttura una caratterizzazione tipica dei processi (software e non) prettamente sequenziali. In una struttura software capace di supportare un tipo di computazione parallela, sono invece identificabili delle unità o moduli la cui attivazione non è strettamente dipendente dagli altri moduli. Essi sono innescati da un insieme ben preciso di condizioni che possono maturare nell' ambito del sistema. Tale status del sistema a cui facciamo riferimento è l' ambiente di vita dei suddetti moduli che, in un sistema di produzione Basato su Regole (RBS), è identificabile con il Data-Base. L' isomorfismo tra un R.B.S. ed un sistema Pattern-Directed resterà fissato quando associeremo le regole ai moduli e, più precisamente, la parte condizione della regola con il pattern di un modulo. Conseguentemente, verrà a determinarsi una corrispondenza tra le parti azioni della regola e del pattern.

Tale isomorfismo rende chiara l' idea che il processo di generazione delle inferenze non debba necessariamente essere un processo di tipo sequenziale. Ecco quindi che affiorano i cosiddetti "conflitti": se in un dato istante lo stato del sistema permette l' attivazione di due o più regole, quale sarà la strategia che limiterà la scelta ? Prima di dimostrare le euristiche di scelta da noi definite, schematizziamo le tre componenti dell' intero processo del modulo Forward: 1)componente di confronto 2)componente di selezione e 3)componente di esecuzione. Il compito della prima componente è quello di generare il conflict-set, costituito, per l' appunto, da un insieme di regole potenzialmente attivabili (fig.3).

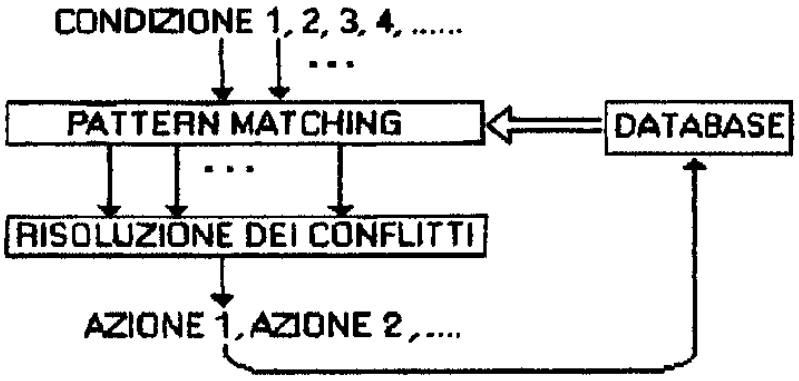


fig.3

```
findall(Regola,
seleziona_regola_da_applicare(Regole,Regola),
Regole_applicabili)

seleziona_regola_da_applicare(Regole,Regola) :-
        length(Regole,N2),
        for(1,N2,X),
        posizione(Regole,X,Elemento),
        Elemento = rule(A,B)
        clause(A,Corpo),
        test1(Corpo),not test1(A),
        Regola = rule(A,Corpo).

test1((A,B)) :- !,test1(A),test1(B).
test1(A) :- sistema(A),!,A.
test1(A) : - clause(A,true).
```

Poiché è altamente improbabile che per la risoluzione di un assegnato "caso" si debbano prendere in considerazione tutte le regole presenti nel sistema, abbiamo previsto la possibilità che la componente di confronto operi soltanto ed esclusivamente su specifiche partizioni della Base di Conoscenza. Ciò comporta una effettiva specializzazione della strategia che è stata realizzata ponendo un semplice controllo sulla variabile-vettore "Regole" (1° argomento della relazione precedente).

Ulteriore peculiarità del modulo Forward è l' agilità con cui permette all' utente di variare dinamicamente, passo dopo passo, la propria definizione operativa dando primaria importanza alla trasparenza, ovvero alla granularità d' esecuzione. L' output di tale componente di confronto viene trasferito all' input della componente di selezione. L' obiettivo di questa componente è quello di effettuare un vero e proprio sort sulle regole applicabili, inviando alla componente successiva (componente di esecuzione) solo la regola che sarà al top della lista ordinata dall' euristica.

```
findall(R,regola_prioritaria(R),R2),
ordina_regole(R2,R9),
  ifthenelse(seleziona_regola_da_applicare(R9,Regola),
        true,
       no_reg_applic2),
applica(Regola,Fatto),
incrementa_fatti(Fatto)
```

L' euristica che permette di estrarre una regola dal conflict-set è implementata dalla procedura "ordina_regole" che, in sintesi, opera secondo il seguente schema:

1) Se il conflict-set è vuoto Allora stop Altrimenti Scegliere nel conflict-set la regola che ha soddisfatte il maggior numero possibile di condizioni nella parte sinistra.

2) Se ci sono due o più regole che hanno ugual numero di condizioni soddisfatte nella parte sinistra, Allora preferire la regola che "scatena" più azioni.

3) Se ci sono due o più regole che hanno ugual numero di condizioni soddisfatte nella parte sinistra ed ugual numero di azioni nella parte destra, Allora segliere una regola a caso.

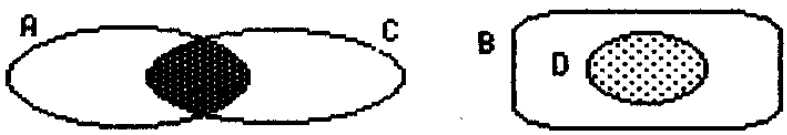
Un approccio classico di ordinamento è, ad esempio, quello descritto da Quigley per un tutore di equazioni lineari. Egli illustra la possibilità di associare staticamente all' insieme delle regole un parametro che denoti la relativa priorità d' applicazione; questa scelta comporterebbe, però, una conoscenza deterministica ed aprioristica delle regole dell' insieme di conflitto, per poterne calibrare ed adeguatamente pianificare una gerarchizzazione ottimale.

In tal senso, l' euristica da noi descritta crediamo meglio si adatti in situazioni particolarmente dinamiche (a cui è sottoposto un guscio) quale quelle della possibilità di operare su diverse basi di conoscenza.

Le motivazioni che ci hanno ispirato nella descrizione del precedente schema sono dovute al fatto che è preferibile privilegiare tutte quelle regole che utilizzano la massima quantità di informazione per produrre inferenza: ciò perché la maggiore informazione «restringe» la ricerca

---

```
Esempio   R1: SE  A    ALLORA  B  (=regola generale)
          R2: SE  A,C  ALLORA  D  (=regola specifica)
```



La procedura da noi implementata che espleta tale ordinamento è la seguente:

```
ordina_regole(ListaRegole,ListaRegOut) :-
length(ListaRegole,Lung),
findall(M,
        (for(1,Lung,X),
        posizione(ListaRegole,X,Elemento),
        Elemento = rule(Testa,Corpo),
        conta_congiunzioni(Corpo,C),
        Priorità1 is 20 - C,
        conta_congiunzioni(Testa,T),
        Priorità2 is 20 - T,
        M = rul(Priorità1,Priorità2,Testa,Corpo)),
              Out1),
sort(Out1,Out2),
findall(Z,
        (for(1,Lung,X1),
        posizione(Out2,X1,Elemento1),
        Elemento1 = rul(N1,N2,Testa1,Corpo1),
        Z = rule(Testa1,Corpo1)),
              ListaRegOut),
!.
```

Anche questo modulo, come il modulo precedente, ha la peculiarità di rendere ben visibile il processo Forward, rendendo trasparente il processo di esplicitazione di una Base di Conoscenza, mostrando, istante per istante, le regole applicate, le modalità con cui sono state raggiunte tutte le conclusioni fino a quel momento conseguite e, ancora, esponendo tutte le conclusioni raggiungibili al passo successivo.

IL MODULO MAIN

Le componenti di cui è costituito il nostro guscio non possono essere indipendenti tra di loro ma devono in qualche modo interagire. Il modulo Main è stato progettato proprio per ottemperare a tale esigenza ed inoltre per ottimizzare la gestione del Sistema in tempo e spazio. La strategia di base è strettamente legata alle caratteristiche del linguaggio di programmazione Prolog ed in particolare alla tecnica di gestione della memoria (L.R.U.) da esso utilizzata. La possibilità di mantenere in memoria virtuale tutti i moduli selezionati, ognuno in un proprio mondo, permette un più veloce accesso al Sistema ed una conseguente riduzione dei tempi medi di attesa.

OSSERVAZIONI CONCLUSIVE

Convinti che l' entusiasmo sia la componente regina durante la fase di apprendimento, abbiamo curato particolarmente l' interfaccia grafica dell' intero progetto, così da rendere chiara e piacevole l' interazione col sistema e ottenendo, allo stesso tempo, un desiderato effetto di trasparenza (fig.4). Abbiamo inoltre incluso nel progetto alcuni possibili modi di riempire lo Shell, tra i quali ci sembra di particolare rilevanza quello relativo ad una piccola Base di Conoscenza sul dominio dei circuiti logici digitali. Tali Basi di Conoscenza, caricate e scaricate dal sistema, offrono un valido modo per illustrare agli utenti che per la prima volta accedono al sistema la funzionalità del sistema stesso. Il modulo inserimento gestisce l' acquisizione della conoscenza, caricando una base già creata (fig.5) oppure creandone una nuova (fig.6); il modulo dimostratore

affronta le problematiche inferenziali relative al ragionamento del backward chaining, permettendo all' utente di scegliere il modo in cui visualizzare tali inferenze (fig.7); il modulo Risoluzione Interattiva mostra come sia possibile ricostruire collegamenti interrotti o inesistenti tra una porta logica ed un' altra (fig.8,9); il modulo Forward mostra quali sono i processi che governano l' elaborazione di un segnale binario attraverso una rete circuitale (fig.10,11). In conclusione vogliamo dire che il Sistema implementato può essere arricchito di altri moduli (modello studente, modello tutoriale....) in modo da riconfigurarlo come modulo esperto di un I.T.S. Shell.

FIGURE

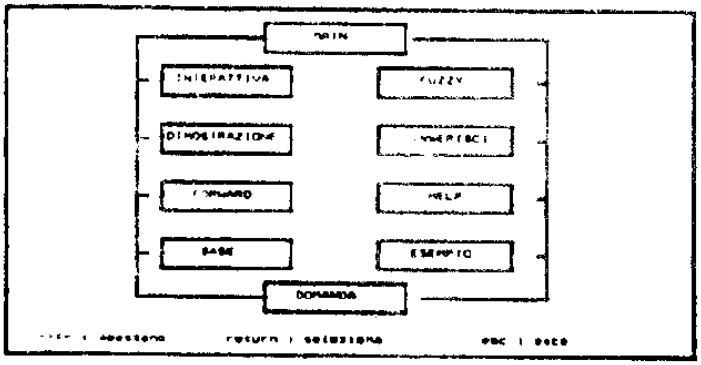


fig.4

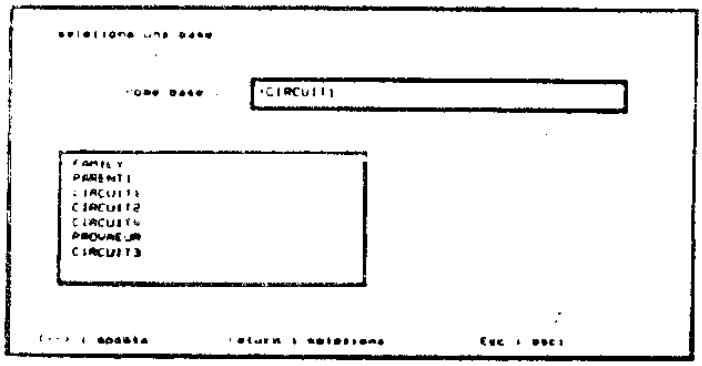


fig.5

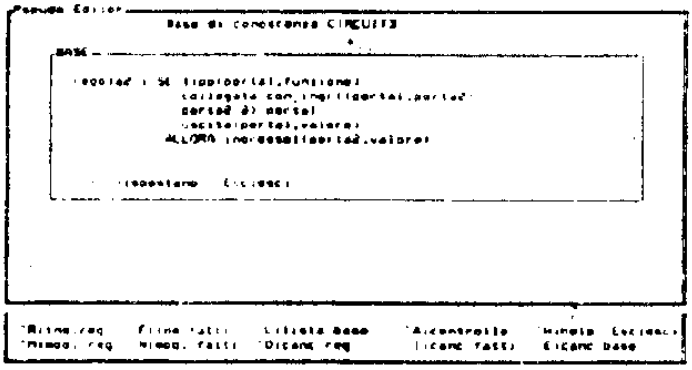


fig.6

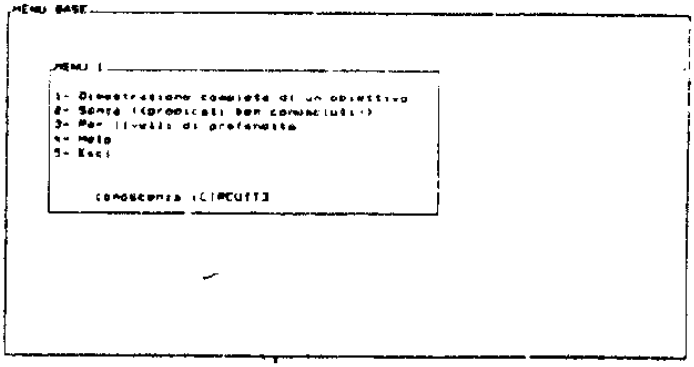


fig.7

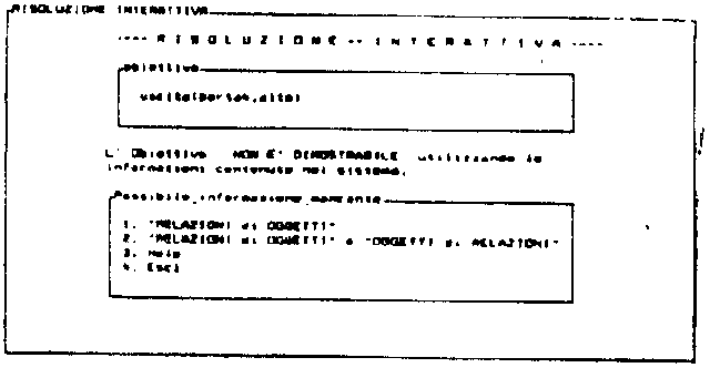


fig.8

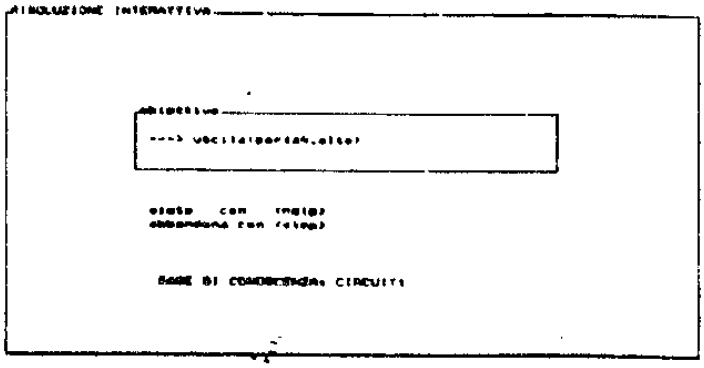


fig.9

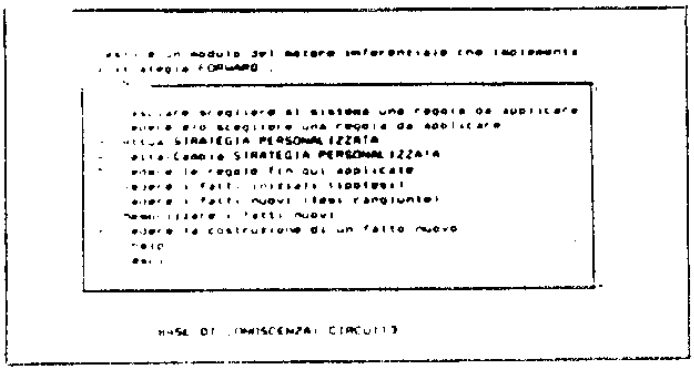


fig.10

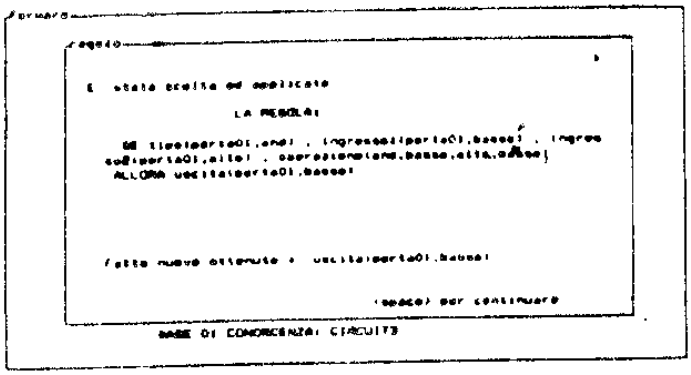


fig.11

BIBLIOGRAFIA

Wenger E. (1987): *Artificial Intelligence and Tutoring Systems*, Los Altos, California, Morgan Kaufmann Publ.

AA.VV. (1989): *Artificial Intelligence and Education*. Proceed. of the 4th Int. Conf. on A.I. and Educ. 24-26 May 1989 Amsterdam, IOS.

Clancey J. William (1987): *Knowledge-Based Tutoring: the GUIDON program*, Massachussetts & England, The MIT Press

Sleeman D. (1989): "PIXIE: a Shell for developping Intelligent Tutoring Sistems" in *Intelligent C.A.I*

Haque M., Rovick A., Michael A. e Evens M. (1989): *Tutoring Rule Authoring System (TRAS)*, Departement of Information Science Northeastern Illinois University.

Shapiro E. e Sterling L. (1988): *The Art of Prolog. Advanced Programming Techniques*, Massachusetts & England, The MIT Press.

Clocksin F. W. e Mellish C.S. (1988): *Programmare in Prolog*, Milano, Franco Angeli.

(TRAS), Departement of Information Science Northeastern Illinois University.

Shapiro E. e Sterling L. (1988): *The Art of Prolog. Advanced Programming Techniques*, Massachusetts & England, The MIT Press.

Clocksin F. W. e Mellish C.S. (1988): *Programmare in Prolog*, Milano, Franco Angeli.