UN SISTEMA BASATO SU REGOLE PER IL TRAINING PROFESSIONALE IN ELETTRONICA

GIOVANNA SISSA

tituto Professionale A. MEUCCI, VIA dei PLATANI 29, GENOVA

SSUNTO

ggetto della presente comunicazione è un' esperienza di lizzo del Prolog per la formazione professionale, che ha dotto alla realizzazione di un prototipo di sistema basato su ole per il training nel campo dell'elettronica integrata.
presente lavoro si inquadra nell'ambito di un progetto ennale CEE di insegnamento del linguagio Prolog e di suoi lizzi in campo didattico, "Experience micro-Prolog", svoltosi gli anni 1985 e 1986 e coordinato dal CREFI di Marsiglia.
esperienza è stata condotta preso un Istituto Professionale per ettronici con studenti di 16-17 anni che non avevano precedenti oscenze informatiche.
ettivi dell'esperienza sono stati l'insegnamento del guaggio micro-Prolog ed il suo utilizzo per applicazioni ative alla pratica professionale in un laboratorio ettronico. Sono stati sviluppati esempi di ragionamento uttivo su databases connessi con l'attività professionale di tecnico elettronico. Il progetto finale ha condotto alla lizzazione di sistema rule based, "EDY", di supporto al nico nella scelta dei Circuiti Integrati Digitali.

L' INSEGNAMENTO DEL LINGUAGGIO

o stati introdotti i meccanismi fondamentali del linguaggio denziandone dapprima la natura dichiarativa ed introducendo aspetti relativi al controllo solo in un secondo momento.
redicati predefiniti ed in particolare la nozione di grammazione "side-effect "sono stati introdotti gradualmente e risolvere situazioni specifiche.
esempi sviluppati riguardano:

- la classificazione di oggetti
- la descrizione di tali oggetti
- la scelta di tali oggetti sulla base di criteri dettati dalle esigenze particolari di una specifica applicazione.

L SISTEMA

2.1 LA CLASSIFICAZIONE

Per prima cosa si è proceduto alla classificazione degli oggetti di un laboratorio elettronico. Gli studenti hanno scritto, sulla base delle proprie competenze la classificazione di tali oggetti, esprimendola mediante un albero, di cui è mostrata una semplificazione nella Fig.1.

```
                        componente
                         attivo
          ----------------------------------
          !                                !
          !                                !
componente attivo                  Circuito Integrato
analogico                                Digitali
                        ------------------------------------
                        !                      !           !
                       TTL                    MOS         ECL
  ---------------------------------        ------------
  !   !    !    !    !     !      !        !    !     !
  !   !    !    !    !     !      !        !    !     !
TTLN TTLL TTLH TTLS TTLLS TTLAS TTLALS   CMOS HCMOS HCTMOS
```

Fig.1

In particolar modo si è concentrata l'attenzione sui Circuiti Integrati Digitali, componenti in cui è realizzato, su un singolo chip, un circuito che implementa una funzione logica complessa. Come mostrato in Fig.1 i circuiti integrati sono divisi, in base alla tecnologia con cui sono costruiti, in famiglie e sottofamiglie.
Ogni integrato è contrassegnato da una sigla che è possibile decodificare per dedurne alcune informazioni, tra cui la famiglia o sottofamiglia di appartenenza.
Ad esempio ogni integrato la cui sigla inizi per 74 e sia seguita dalle lettere (o gruppi di lettere) N, L, H, S, LS, AS, ALS è un componente della sottofamiglia TTL N, TTL L, TTL H, ecc.(*)

```
(74 _subfam _tipo) is-a (TTL _subfam) if
                    _subfam appartiene-a (N L H S LS AS ALS)
```

Un insieme di regole analoghe permette di attribuire a qualunque integrato la famiglia di appartenenza.

(*) si è fatto uso della sintassi SIMPLE del micro-Prolog

2.2 LA DESCRIZIONE DEI COMPONENTI

Nella fase successiva si sono descritti gli oggetti prima classificati.
Nella pratica le sorgenti da cui si desumono le informazioni sui componenti, manuali o datasheet, sono spesso ridondanti e confuse. Il primo problema è stabilire quali informazioni scegliere tra le molte a disposizione e come strutturarle.
Vengono considerate varie funzioni: porte, flip flop, contatori, ecc. Per ognuna si esse si è stabilito quali attributi vadano specificati. Ad esempio per i flip flop si riporterà il tipo (RS, D, ecc), il modo di funzionamento del clock e la presenza di ingresi asincroni. La descrizione dei componenti varierà dunque a seconda della funzione; ogni integrato è descritto nel database da un fatto, strutturato nel seguente modo:

(sigla) has ((funzione attributi) n circuiti n pins)

Abbiamo visto che dalla sigla è possibile dedurre a quale famiglia (o sottofamiglia) logica appartenga un circuito. Ogni famiglia logica è contraddistinta da alcune caratteristiche, che si possono raggruppare come:
- caratteristiche logiche, ossia valori di tensione(massimi e minimi) corrispondenti ai livelli logici 0 ed 1;
- caratteristiche elettriche, ossia le correnti di ingresso ed uscita a livello 0 e 1;
- prestazioni, quali la velocita di commutazione, la massima frequenza di clock, la potenza dissipata, ecc.

Le varie caratteristiche di ogni famiglia vengono scritte nel database come fatti; ad esempio si struttura l'informazione relativa alle caratteristiche logiche di una famiglia in un solo fatto mediante il predicato binario "ha-caratt-logiche" che ha come primo argomento il nome della famiglia e come secondo la lista dei valori (espressi in Volt) di tali caratteristiche,
Ogni circuito eredita le caratteristiche della famiglia logica a cui appartiene. Tale erediatarietà è esprimibile in Prolog, in modo molto immediato:

```
_CIRCUITO ha-caratt-logiche _X if _C is-a _FAMIGLIA and
                             _FAMIGLIA ha-caratt-logiche _X
```

Sono stati definiti alcuni predicati che consentono di effettuare un' interrogazione veloce della basedati, ponendo le domande più usuali nella reale attività di laboratorio.
Per esempio è stato definito un predicato a tre argomenti:

funzione-attributi(_I _F _A),

che consente di determinare quali sono gli integrati _I che

realizzano la funzione logica di tipo _F avente gli attributi _A.

2.3 I CRITERI DI SCELTA DEI COMPONENTI

La fase centrale del progetto consiste nella scrittura delle regole che consentono di scegliere i componenti più adatti per una situazione specifica. Nella pratica professionale si hanno a disposizione vari Circuiti Integrati Digitali, realizzati in tecnologie diverse che implementano tutti la medesima funzione logica e con gli stessi attributi. Se, quindi, dal punto di vista della "funzionalità" dovrebbero essere equivalenti, in realtà differiscono per alcune caratteristiche, ad esempio elettriche o di potenza o di velocità o di costo. Uno dei problemi principali che un tecnico elettronico si trova a dover risolvere è quindi: "Come scegliere il componente più adatto per una situazione specifica?".
In generale il tecnico opera tale scelta sulla base della propria esperienza professionale, che si esprime mediante un insieme di regole; scegliere un circuito significa verificare che sussistano determinate relazioni fra oggetti. Il ragionamento effettuato è di tipo backward.
L'idea fondamentale è riprodurre tale expertise in Prolog, con il duplice scopo di:

- abituare gli studenti ad esplicitare ed a formalizzare il ragionamento che si applica nella pratica professionale di laboratorio, esprimendolo in modo rigoroso;
- costruire uno strumento di ausilio nella reale attività professionale per supportare il tecnico nella scelta del componente, utilizzabile anche per il training.

Sono state definite le relazioni fra oggetti che consentono di stabilire quali componenti usare. Ad esempio è stata definita la "compatibilità" fra circuiti, ossia la possibilità di usare due circuiti in cascata. In generale si può dire che due Circuiti Integrati Digitali sono compatibili se sono compatibili dal punto di vista logico, ossia se i valori di tensione corrispondenti allo 0 ad all'1 logico sono riconosciuti correttamente, ed elettrico, ossia se un circuito non sovraccarica l'altro. In Prolog ciò si esprime come:

```
_C1 compatibile _C2 if _C1 logic-compatibile _C2 and
                       _C1 elettric-compatibile _C2
```

dove la compatibilità logica è espressa come:

```
_C1 logic-compatibile _C2 if
_C1 ha-caratt-logiche(_VOHmin1 _VIHmin2 _VOLMAX1 _VILMAX1) and
_C2 ha-caratt-logiche(_VOHmin2 _VIHmin2 _VOLMAX2 _VILMAX2) and
_VOHmin1 GE _VIHmin2 and _VILMAX2 GE _VOLMAX1
```

ed analogamente la compatibilità elettrica.
E' stata poi definita la "sostituibilità" fra circuiti. Due circuti per essere sostituibili devono: essere sostituibili dal punto di vista delle funzionalilità, ossia realizzare la medesima funzione e con i medesimi attributi ed essere pin-to-pin compatibili, ossia fisicamente sostituibili su una scheda.
La condizione di sostituibilità è così espressa come:

_C1 sostituibile _C2 if _C1 funz-sostituibile _C2 and
_C1 pin-to-pin-compatibile _C2

Poichè l'essere pin-to-pin compatibili non è determinabile con regole di tipo generale, ma si può stabilire solo con un confronto diretto fra i componenti in questione, il sistema chiederà di volta in volta all'utente se i due circuiti sono pin-to-pin-compatibili. Sulla base di tale interazione con l'utente la basedati può venir aggiornata, memorizzando, se necessario, fatti come:

(74 H 74) pin-to-pin-compatibile (74 S 74)

Il sistema calcola i valori di fan-out e di fan-in che si ottengono effettuando una certa sostituzione e li comunica all'utente, per consentirgli di verificare se l'integrato "sostituto" in una determinata applicazione è in grado di pilotare i circuiti a valle e di essere pilotato, a sua volta, da quelli a monte.
Perchè la sostituibilità sia completamente garantita devono essere verificate altre condizioni: tutte le prestazioni del circuito che sostituisce devono essere migliori (o eguali) a quelle del circuito sostituito.
Viene così definita una relazione di sostituibilità perfetta:

_C1 perfett-sostituibile _C2 if _C1 sostituibile _C2 and
_C2 ha-prestazioni-migliori _C1

A volte sostituendo un componente di una famiglia con uno di un'altra alcune prestazioni possono modificarsi, peggiorando. Ad esempio sostituendo un componente TTLL con un componente TTLH la potenza dissipata aumenta. In alcuni casi tali variazioni possono non avere importanza, ma in altri casi possono essere fondamentali.
Per trattare i casi in cui interessano mantenere solo alcune delle prestazioni del circuito da sostituire il sistema pone all'utente alcune domande sulle particolarità dell' applicazione e "consiglia" alcune scelte, segnalando quali caratteristiche possono essere soggette a peggioramento.

3. VALIDAZIONE

Il sistema è stato validato utilizzandolo in vari casi concreti sia come un ausilio per le scelte che come strumento per verificare la correttezza delle scelte fatte.

4. CONCLUSIONI

I problemi trattati non costituiscono che un esempio di una più vasta classe di problemi, relativi alla classificazione di oggetti, alla descrizione di tali oggetti ed ai criteri di scelta fra essi a fronte di situazioni specifiche.
La portata del discorso dal punto di vista della formazione è ben più generale. Basti pensare alle classificazioni in biologia o in scienze naturali. E basti bensare a quanti casi nella vita reale o nella pratica professionale in cui dover scegliere fra diversi oggetti qual'è il piu' adatto ad una specifica applicazione. Ad esempio i pneumatici di un'automobile sono scelti sulla base del tipo di strada che si presume di dover fare.
La scelta di utilizzo del Prolog si è dimostrata molto "naturale" ed adeguata al soggetto, data la sua duplice natura dichiarativa e "rule based". Inoltre in Prolog è agevole rappresentare i legami gerarchici e far ereditare ad ogni circuito integrato le caratteristiche della famiglia logica a cui appartiene. La scelta del Prolog inoltre ha consentito lo sviluppo del prototipo del sistema in tempi rapidi.

5. BIBLIOGRAFIA

1) M. Bergmann "Prolog: Programming in Logic ", First European Seminar on Informatic and Education, Marseille,1984.
2) A.Bundy,H.Pain "What stories should we tell novice Prolog programmers?",Dept on AI of Edimburgh Paper n.269,1985.
3) K.L.Clark, F.G.McCabe "Micro-Prolog:Programming in logic", Prentice Hall, 1984.
4) T.Conlon "Start Problem Solving with Prolog", Addison Wesley,1985.
5) H.O.de Saram "Programming in micro-Prolog", Ellis Horwood,1985.
6) J.R. Ennals "Beginning micro-Prolog" 2nd edition, Ellis Horwood,1984
7) F.Giannesini,H.Kanoui, R.Pasero,M.Van Caneghen "Prolog", InterEditions,1985.
8) G.Sissa "Using Proog in teaching of electronics", Annual Conference Prolog Education Group, Exeter, Luglio 1986.
9) The world TTL, IC DATA and cross reference guide, 1982.

VERSO UN SISTEMA AUTORE INTELLIGENTE.

Rappresentazione della conoscenza in un sistema didattico domanda-risposta.

Daniela Santangelo

DIDA*EL
Società per la Didattica con l'elaboratore

RIASSUNTO

In questo lavoro si descrive il nucleo centrale di un assistente alla formazione basato su conoscenza.
L'architettura del sistema si sviluppa intorno a quattro basi di conoscenza distinte, mantenenti la conoscenza corretta e quella degli errori, il modello studente (costruito durante l'interazione) e la strategia didattica, rispettivamente.
Nel seguito viene descritto l'attuale prototipo in cui sono state sviluppate le basi di conoscenza contenenti le descrizioni corrette ed errate del campo oggetto prescelto per la sperimentazione, e parte di quella descrivente la strategia didattica.
La versione corrente del sistema è in PROLOG-2 su Olivetti M-24; le possibili applicazioni coprono l'intero settore del dialogo cooperativo e sono quindi estendibili al di fuori dell'ambito didattico.

CONOSCENZA E SISTEMI ICAI

Si può ascrivere conoscenza ad un programma solo attraverso un'analisi comportamentale del programma stesso. Nell'ipotesi, formulata da Newell /NEWE 81/, dell'esistenza di un livello di conoscenza, un programma è un agente cui un osservatore ascrive delle finalità osservabili dal comportamento oppure note per comunicazione linguistica. La conoscenza che l'osservatore può attribuire all'agente è tutto ciò che consente ad esso di manifestare un comportamento che eventualmente conduce al raggiungimento delle finalità. Questo processo di attribuzione è però relativo all'osservatore che, a sua volta, è un agente con conoscenza e finalità proprie. Così nella prospettiva aperta dall'ipotesi di Newell non esiste sistema che non sia un agente con conoscenza o che non ascriva conoscenza ad altri agenti.
Questa descrizione sembra calzante per i sistemi ICAI di apprendimento per dialogo. Alcuni di tali sistemi hanno infatti una rappresentazione della conoscenza del partner che permette una individualizzazione del dialogo allo scopo di massimizzare l'apprendimento, risultato che corrisponde appunto alle finalità sia del docente che dell'utente.
Un'altra caratteristica fondamentale di questi sistemi è che

essi mantengono conoscenza che nei confronti del campo oggetto definiremmo "inesatta" ma che è necessaria per inferire le cause degli errori commessi dallo studente (esempi si possono trovare in: /SOLO 82/ e /CERR 84/).

Diversamente da quella corretta, la conoscenza incorretta non può essere acquisita solo interagendo con esperti, in quanto nessun esperto può presentare, neppure teoricamente, uno spettro degli errori che gli studenti possono commettere perchè questi sono originati da processi (nella mente dello studente) che neanche un esperto può controllare. La robustezza di un sistema ICAI dipende dunque da una fase di sperimentazione durante la quale i progettisti acquisiscono la conoscenza incorretta più valutando il comportamento degli studenti che interagendo con esperti.

STRUTTURA DEL SISTEMA DI RAPPRESENTAZIONE

In accordo con quanto detto, il prototipo mantiene due rappresentazioni separate della conoscenza relativa al campo oggetto. Il loro insieme descrive implicitamente tutti i possibili modelli utente definibili dal sistema; il lavoro attuale è volto a consentire al sistema di costruire un modello del particolare studente allo scopo di individualizzare l'interazione.

Il formalismo di rappresentazione astratta che abbiamo scelto per descrivere la conoscenza in gioco utilizza regole del tipo:

Nome_Regola : if Condizione then Conclusione.

e fatti espressi nella seguente forma:

Nome_Fatto : Fatto.

Le regole ed i fatti che costituiscono la conoscenza corretta e quella sbagliata possono essere distinti, oltre che per il luogo in cui sono fisicamente memorizzati, anche per il nome: nel primo caso (conoscenza corretta) viene usato un nome del tipo <regola-id>/<fatto-id>, dove id è un identificatore, mentre nel secondo vengono usati nomi del tipo <regola_errata-id>, <fatto_errato-id>.

Il sistema è in grado di risolvere gli stessi problemi che pone allo studente e ciò gli conferisce la capacità di fornire spiegazioni sul modo in cui determina le soluzioni. Osserviamo che nel contesto del dialogo tutoriale l'esigenza di spiegazioni convincenti è ancora più stringente che nel caso dei Sistemi Esperti in quanto il processo di soluzione e il suo livello di dettaglio devono "in qualche modo" riflettere il modo di pensare degli uomini (/CLAN 86/). Sulla base della propria soluzione il sistema è in grado di valutare la correttezza delle risposte dello studente. Se rileva un errore attiva la base di conoscenza "incorretta" e cerca di ricavare la stessa risposta fornita dallo studente ragionando sull'insieme di fatti e regole corretti

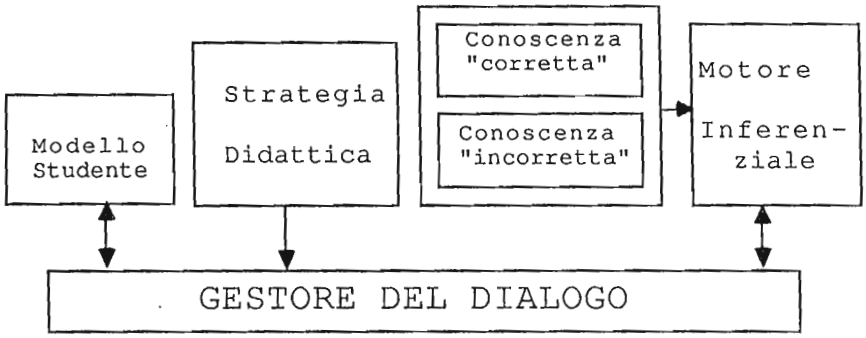


Fig. 1 Struttura attuale del sistema

sbagliati. Il risultato di questa analisi è, nel caso più generale, un insieme di catene inferenziali, cui corrispondono diverse possibili deduzioni dell risposta. Un problema che si presenta a questo punto è quindi quello di capire quale catena inferenziale, fra quelle individuate, corrisponde all'errore commesso dallo studente.

Questa componente di elaborazione si presta all'utilizzo di euristiche, e come tale è in larga misura dipendente dal campo oggetto. Per motivi di generalità le euristiche sono programmate dichiarativamente utilizzando un linguaggio definito ad hoc.

La filosofia generale del sistema non pone però particolare enfasi sull'uso di euristiche. Così come è impossibile che un esperto fornisca un elenco dei possibili errori dell'utente altrettanto è impossibile definire euristiche "forti" che permettano di individuare univocamente la sorgente di errore sotto forma di regole e/o assunzioni errate.

L'uso di euristiche deboli, che permettano una prima "scrematura" delle catene inferenziali prodotte in conseguenza di un errore, deve invece essere necessariamente seguito da una fase di interazione con l'utente volta ad individuare chiaramente la sorgente dell'errore.

Il linguaggio di implementazione deve pertanto permettere di programmare facilmente la capacità del sistema di risolvere i problemi che pone agli studenti, di rendere il processo di soluzione "trasparente" e cognitivamente plausibile e di effettuare la diagnosi sulla base di un confronto fra il modo con cui il sistema risolve i problemi e il modo in cui si pensa che lo studente abbia pensato di risolverli. Ciò può essere fatto analizzando i procedimenti deduttivi registrati sotto forma di catene inferenziali.

Il PROLOG soddisfa questi requisiti e inoltre garantisce, rispetto alle shell disponibili su Personal Computer, maggiore portabilità e più flessibilità nella programmazione di altre componenti del sistema per cui il paradigma delle regole non è adeguato.

L'architettura del sistema nello stato attuale è presentata in figura 1. Il modulo che gestisce il dialogo pone domande riguardanti il campo oggetto di cui è "esperto" e, nel caso di risposta sbagliata, dopo aver determinato le possibili cause di errore, seleziona un insieme di cause probabili (utilizzando un procedimento euristico), dal quale infine viene selezionata la causa di errore dopo una fase di interazione con l'utente.

ANALISI DELL'ERRORE E COSTRUZIONE DINAMICA DELLA STRATEGIA DI DIALOGO: UN ESEMPIO

Durante questa fase del lavoro abbiamo selezionato un campo oggetto che avesse la caratteristica di essere facilmente formalizzabile, e precisamente il compito costituito dalla formazione del participio passato dei verbi regolari e irregolari italiani.

Con l'aiuto di un esperto è stata fatta un'analisi approfondita delle analogie esistenti fra il participio passato di tutti i verbi irregolari, per cercare di trovare una classificazione da utilizzare poi in fase di diagnosi degli errori. I dati emersi da questa indagine sono stati inseriti nella base di conoscenza sotto forma di fatti e regole e adesso costituiscono la conoscenza corretta del sistema.

Particolarmente interessante è stata la costruzione della base di conoscenza che costituisce invece la conoscenza incorretta del sistema. Il metodo seguito è stato quello di esaminare in dettaglio ogni fatto ed ogni regola costituenti la conoscenza corretta e di inserire come fatto o regola sbagliati ogni alterazione di quel fatto o di quella regola che risultasse sensata agli occhi dell'esperto del campo oggetto.

Il rapporto fra la limitatezza degli obiettivi che ci siamo posti con l'attuale prototipo e la notevole quantità di regole che sono state individuate dall'esperto e "programmate" (circa 40; secondo /HAYE 85/ un sistema con 50 regole è tecnologicamente interessante e un sistema commerciale deve avere almeno 50 regole) da una idea della complessità di questo tipo di sistemi.

In figura 2a e 2b sono indicati rispettivamente alcuni dei fatti e delle regole "corretti" e "sbagliati" relativi ad una parte della conoscenza del sistema (quella sui verbi regolari). In figura 3 viene invece mostrato un breve frammento di interazione in cui le risposte dell'utente sono in grassetto precedute da un asterisco, mentre i commenti sono scritti in corsivo.

DISCUSSIONE

In questo lavoro abbiamo visto come il paradigma del dialogo fra agenti con obiettivi e conoscenza autonome sia adeguato per descrivere i requisiti di un programma didattico intelligente.

Il nostro vuole essere un contributo alla costruzione di un ambiente per la produzione di programmi didattici con queste

```
fatto_1a: des_part(prima,ato).
fatto_1b: des_part(seconda,uto).
fatto_1c: des_part(terza,ito).

regola_1:if
        infinito_regolare(Verbo)
        and
        participio_regolare(Verbo)
        and
        coniug(Verbo,Con)
        and
        des_part(Con,Des)
        and
        tema(Verbo,Tema)
        and
        appendi(Tema,Des,Part)
      then
        participio(Verbo,Part).

regola_2c:if
        des_infinito(Verbo,ire).
      then
        coniug(Verbo,terza).
```

a) rappresentazione di dati corretti

```
fatto_errato_1a1: des_part(prima,uto).
fatto_errato_1a2: des_part(prima,ito).
fatto_errato_1b1:des_part(seconda,ato).
fatto_errato_1b2:des_parti(seconda,ito).
fatto_errato_1c1: des_part(terza,ato).
fatto_errato_1c2: des_part(terza,uto).

regola_errata_2c2:
        if
              des_infinito(Verbo,ire)
        then
              coniug(Verbo,seconda).
```

b) rappresentazione di dati sbagliati

Fig. 2 Parte della conoscenza relativa alla formazione del participio passato di un verbo regolare.

Qual è il participio passato del verbo sopperire?

* **sopperuto** *A questo punto il sistema riconosce la presenza di un errore e parte con l'analisi attivando la base di conoscenza che contiene i dati sbagliati. Vengono generate dal motore inferenziale le catene inferenziali che possono portare alla deduzione della risposta sbagliata fornita dall'utente. Per póter selezionare una di esse il sistema deve appurare quale, fra gli errori che esse prevedono, può essere attribuito all'utente. Per far ciò il sistema deve porre una ulteriore domanda che permetta di discriminare. La domanda viene trovata nella base di conoscenza che abbiamo chiamato strategia didattica associata al nome dell'errore.*

Il participio passato del verbo sopperire è sopperito.
Di che coniugazione è il verbo sopperire ?

* **seconda** *Con questa risposta dell'utente il sistema è in grado di determinare la causa dell'errore dell'utente.*

Il verbo sopperire non è della seconda coniugazione ma della terza.
Per questo motivo avevi sbagliato anche la risposta alla domanda precedente.

Fig.3 Frammento di interazione

caratteristiche. La programmazione di un prototipo coincide con la definizione della conoscenza -corretta e incorretta- relativa al campo oggetto (mediante opportuno formalismo) e della strategia didattica che si intende adottare (mediante un linguaggio di specifica). A carico dell'ambiente rimangono invece tutta la gestione dell'interazione e la costruzione del modello dell'utente.

Attualmente stiamo mettendo a punto il linguaggio di specifica necessario per programmare la strategia didattica e definendo la parte del sistema che costruisce il modello utente.

Il prossimo obiettivo è quello di corredare il sistema con un insieme di moduli di interfaccia verso l'utente che permettano all'esperto di un particolare campo oggetto di programmare la propria applicazione senza ricorrere all'aiuto di un ingegnere della conoscenza.

Lo sviluppo di sistemi intelligenti con queste caratteristiche è tuttora argomento di ricerca fondamentale (vedi ad esempio /CERR 86/) e applicata (JENK 86) in AI.

RINGRAZIAMENTI

Questo lavoro avrebbe risentito della mia apprensività se non fosse stato per le frequenti discussioni avute con Stefano Cerri, Mauro Leoncini e Gianna Martinengo. Determinante è stato l'aiuto di Luciana Saetti per l'analisi del campo oggetto.

BIBLIOGRAFIA

/CERR 84/ Cerri S.A., Fabrizzi M., Marsili G.; The Rather Intelligent Little Lisper; in AISB Spring/Summer 1984 n. 50, pp. 21-24.

/CERR 86/ Cerri S.A.; The requirements of conceptual modelling systems; in INTELLIGENT COMPUTER AIDED INSTRUCTION, edito da John Self, pubblicato da Chapman and Hall (U.K.) and Methuen (U.S.A.) in stampa.

/CLAN 86/ Clancey William J.; Qualitative Student Models, "Annual Reviews Computer Science", 1986, pp 381-450.

/HAYE 85/ Hayes-Roth S.; Knowledge based Expert Systems - The state of the art in the U.S., Knowledge Engineering Review vol.1 n.2, Giugno '85.

/JENK 86/ Jenkins P.G.; TUTOR: a shell for intelligent omputer aided instruction; presentato a KBS '86: Online Publications, Pinner, UK, 1986.

/NEWE 81/ Newell A.; The Knowledge Level; AI Magazine, Summer 1981, pp. 1-22.

/SOLO 82/ Soloway E., Ehrlich K., Bonar J., Greenspan J.; What Do Novices Know About Programming?; in "HUMAN/COMPUTER INTERACTION", Badre A. e Shneidermann B. eds., Ablex Publishing Corporation, pp. 27-54.

UN AMBIENTE LOGICO PER IL GIOCO DEGLI SCACCHI

Paolo Ciancarini e Mauro Gaspari
Dipartimento di Informatica - Pisa

Riassunto
La ricerca, e la competizione, nel campo delle macchine capaci di giocare a Scacchi è correntemente dominata dal paradigma "*forza bruta*", cioè dall'impiego di potenti tecnologie hardware. Questo lavoro critica l'approccio dominante ed invoca un paradigma maggiormente orientato all'utente ed all'ingegneria della conoscenza. Viene esplorato il ruolo della metaconoscenza nella valutazione delle posizioni ed evidenziata la necessità di sviluppare adeguate tecniche di planning. Viene descritto un ambiente ideale, chiamato METAChess, in cui l'utente ha il pieno controllo sia sul livello operativo (regole del gioco, interfaccia) sia sul metalivello di pianificazione e valutazione (regole strategiche e tattiche). L'obiettivo è la costruzione di un sistema capace di dialogare su una partita di Scacchi alla maniera degli esperti umani. Anche se il progetto è attualmente nella sua fase iniziale, è stato costruito un prototipo in PROLOG.

Introduzione

Negli ultimi anni le macchine capaci di giocare a Scacchi hanno raggiunto una notevole forza di gioco. Tale miglioramento è dipeso in grandissima parte dall'uso di tecnologie hardware innovative. Anche le proposte più recenti si basano su architetture altamente efficienti [Berliner 86]. Come è noto, la forza di gioco di una macchina che gioca a Scacchi è in proporzione non lineare con la profondità dell'albero di gioco analizzato. Si deduce che, grazie ai prevedibili avanzamenti tecnologici, entro 10 anni la forza di gioco sarà pari, se non superiore, a quella dei migliori giocatori del mondo [Thompson 82].
A nostro avviso la situazione attuale è sconcertante. Lo studio sui giocatori artificiali si è sviluppato come terreno di ricerca riguardo l'Intelligenza Artificiale, e non come sistema di valutazione dell'efficienza di nuove architetture. Un altro punto da notare è che nessun giocatore professionista usa nel suo allenamento le macchine attuali, per quanto potenti ed economiche siano. Le ragioni di questa mancata accettazione possono essere complesse, ma certo si sente la mancanza di interfacce amichevoli e di programmi trasparenti.
In questo lavoro si propone un paradigma alternativo di ricerca, con l'obiettivo di costruire una macchina capace di dimostrare un comportamento esperto e di assistere un professionista nel suo allenamento oppure un ricercatore nello sviluppo di nuove idee. Per 'comportamento esperto' intendiamo la capacità di usare conoscenza strategica su una linea di apertura, di scoprire obiettivi ragionevoli nel mediogioco o nel finale, di commentare alla maniera dei Maestri una

partita di Scacchi. L'ambiente deve comportarsi come un assistente intelligente: l'utente può sia giocare col sistema, sia interrogarlo (chiedendogli per esempio di spiegare le sue mosse evidenziandone la strategia), sia inserire nuova (meta)conoscenza specifica. Quest'ultima possibilità è interessante soprattutto se si vogliono studiare varianti eterodosse del gioco.
L'ambiente di programmazione che abbiamo scelto per l'implementazione è quello dei linguaggi logici, in particolare del PROLOG. Questa scelta è stata determinata sia dalla natura del compito [Aiello 84], sia dai risultati di una precedente esperienza di uno degli autori, che ha implementato un prototipo in MRS, un formalismo basato sul LISP per la costruzione di sistemi esperti [Gaspari 86].
Questo lavoro si compone di tre parti: nella prima si discutono le architetture correnti e se ne evidenziano le debolezze in base al nostro punto di vista. Nella seconda si delinea la struttura del sistema e del prototipo attualmente funzionante. Nelle conclusioni si traccia un confronto tra MRS e PROLOG ed una sintesi dei futuri sviluppi.

1. Struttura di una macchina scacchistica

Una macchina scacchistica normale si compone essenzialmente di tre moduli:

• Interfaccia utente: offre un insieme di funzionalità di comunicazione tra l'utente e la macchina. Le interfacce correnti sono orientate alle comunicazioni di "partita viva": viene mostrato lo stato della scacchiera, il tempo consumato dai giocatori ed alcune informazioni sulle mosse future previste dal programma.
• Generatore di mosse: è il componente "sintattico", che garantisce che la posizione si evolva legalmente. Costruisce l'albero del gioco di profondità N, dove N è scelto dall'utente in funzione della potenza di calcolo offerta dall'hardware.
• Funzione di valutazione: è il componente "semantico", che ingloba la conoscenza scacchistica specializzata. Data una posizione, sceglie una mossa tra quelle legali, usando l'algoritmo Minimax (o meglio, le sue varianti più efficienti). La funzione vera e propria è un polinomio che dà una valutazione statica di una foglia dell'albero di gioco.

In alcuni casi esistono altri moduli, in particolare database di aperture e di finali. In nessun caso questi dati sono integrati con la funzione di valutazione: alla fine di una variante di apertura non viene usata alcuna informazione strategica.

In generale, le macchine scacchistiche sono opache nei confronti dell'utente: non è possibile modificare ne' il generatore di mosse ne' la funzione di valutazione. La prima possibilità sarebbe assai utile per trattare i problemi cosiddetti eterodossi: automatti, aiutomatti, fairy chess (gli Scacchi eterodossi seguono insiemi di regole più o meno differenti da quelli ortodossi). D'altra parte, la funzione di valutazione offre una visione puramente quantitativa dell'andamento del gioco ed è poco utilizzabile da utenti esperti di Scacchi. In

particolare, nei sistemi attuali è impossibile ottenere in termini scacchistici una risposta alla domanda

"perchè hai scelto questa mossa?"

Il processo di generazione di un piano è stato analizzato in un gran numero di testi scacchistici. Si può suddividere in tre fasi: analisi, sintesi e pianificazione [Ponzetto 85]. Durante la prima fase vengono evinte le caratteristiche della posizione (struttura dei pedoni, dislocazione dei pezzi, sicurezza dei Re). Nella fase di sintesi vengono stabiliti uno o più obiettivi in base ai risultati dell'analisi. L'ultima fase, la pianificazione, consiste quindi nella scelta di una sequenza di mosse capaci di realizzare gli obiettivi determinati dalla sintesi.
Le macchine attuali sono capaci solamente di fare analisi più o meno sofisticate e comunque in termini quantitativi. D'altra parte, la conoscenza scacchistica presenta in minima parte natura quantitativa. E' ad esempio estremamente difficoltoso caratterizzare in termini quantitativi la differenza tra una posizione vincente ed una solamente vantaggiosa [Hartston 86]. Inoltre, gli esperti umani fanno largo uso di metaconoscenza durante il gioco: ad esempio, nella valutazione di una posizione, sanno capire se è tempo di calcolare varianti oppure se è meglio affidarsi all'intuizione affinata dall'esperienza.

Non essendo particolarmente interessati alla forza di gioco e quindi all'efficienza del sistema, un ambiente basato sulla programmazione logica è ideale per sviluppare un sistema che replichi la maniera giocare umana. L'equivalenza tra dati e programmi, la possibilità di gestire metaconoscenza, la capacità di estendere le funzionalità del sistema mediante metaprogrammazione, sono tutte caratteristiche che abbiamo utilizzato nella progettazione.

2. Un (meta)ambiente esperto di Scacchi

In questo paragrafo delineiamo i requisiti e la struttura del sistema METAChess, che si trova correntemente in una fase iniziale di implementazione. L'architettura complessiva dell'ambiente consiste di una gerarchia di livelli trasparenti all'utente, che deve poter accedere sia al modulo d'interfaccia, sia ai livelli "profondi", cioè quelli del generatore di mosse e della funzione di valutazione.
Possiamo immaginare la scacchiera come un insieme di celle comunicanti, come in [Berliner 86]. Ciascuna cella esegue continuamente un programma PROLOG, e trasmette una lista di mosse pseudolegali ad un supervisore che raccoglie i messaggi e costruisce l'albero da valutare [Somalvico 82]. Anche se questo schema possiede un alto grado di parallelismo potenziale, non ci siamo prefissi di costruire un generatore ad alto grado di efficienza. Più importante è la presenza di due funzioni di generazione: una di tipo classico, che calcola la lista delle mosse legali in una data posizione, ed una 'goal oriented', il cui risultato è una lista di mosse atte a realizzare un particolare obiettivo. In mancanza di hardware specializzato, la posizione è per ora rappresentata con un insieme di asserzioni

square(I/J,P/C). % *P è nella casa di riga I e colonna J.*

dove P è il tipo di pezzo e C è il colore. Questa teoria iniziale viene arricchita dal generatore con fatti del tipo

occupabile(I_1/J_1,I_2/J_2,P/C). % *P può muovere da I_1/J_1 in I_2/J_2*
attacca(square(I_1/J_1,P1/C1),square(I_2/J_2,P2/C2)). % *P1 attacca P2*
difende(square(I_1/J_1,P1/C1),square(I_2/J_2,P2/C2)). % *P1 difende P2.*

La teoria arricchita dal generatore viene analizzata dalle funzioni di valutazione. L'ambiente offre attualmente tre differenti funzioni:

• *polinomio convenzionale,* che in base a considerazioni di materiale e di mobilità dei pezzi dà una valutazione quantitativa della posizione.
• *funzione di analisi statica e dinamica:* data una posizione del mediogioco viene calcolata una lista di fatti che la caratterizzano. Fatti possibili sono:
"la colonna c è aperta", "la colonna e è semiaperta", "il pedone d5 è debole", "d4 è una casa forte per il Bianco".
Si ottiene così una sorta di 'colorazione' della scacchiera: certe case vengono classificate "critiche". Nella Figura 2. è mostrata la colorazione corrispondente alla posizione mostrata in Figura 1. La densità di colorazione rappresenta l'importanza strategica dell'elemento. La freccia rappresenta un attacco in diagonale.

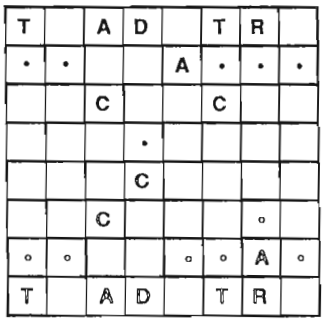


Fig.1. Posizione.

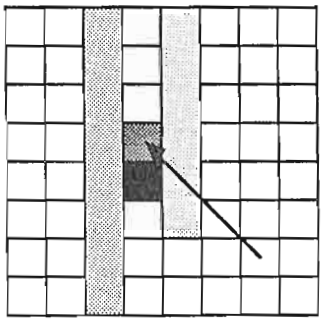


Fig.2. Colorazione.

• *funzione di sintesi:* dati i fatti trovati con l'analisi, si costruisce una lista di obiettivi unitari. Una tavola di consigli analizza il pattern delle case critiche e, mediante forward reasoning, trova gli obiettivi. Obiettivi possibili sono:
"occupare la colonna c con la Ta1", "attaccare il pedone in d5", "avanzare il pedone e2".

Tale lista di obiettivi può essere usata come supporto di un sistema basato su un "advice language" [Bratko 86]. Con questo tipo di linguaggi l'utente specifica in modo dichiarativo quali idee vanno tentate in certi tipi di posizioni. E' possibile costruire varie "tabelle di consigli" in forma di sistemi di regole, per gestire differenti classi di posizioni; la funzione di sintesi descritta sopra costituisce un insieme di metaregole capaci di selezionare le tabelle

più adeguate.
Questi tre moduli dovranno essere integrati, in modo che i risultati calcolati dall'uno possano essere utilizzati dagli altri. La struttura del polinomio di valutazione può essere alterata dinamicamente in relazione ai risultati dell'analisi; per esempio, il valore materiale di un pezzo cambia in funzione delle caratteristiche della posizione.
I seguenti sono altri moduli del sistema, per ora non implementati:

• *planner*: data una lista di goal unitari, questo modulo ne sceglie un sottoinsieme e chiama il generatore goal-oriented per calcolare una lista di mosse che realizza l'obiettivo. E' certamente il modulo più complesso da costruire, in quanto ingloba la maggior quantità di conoscenza specifica.
• *commentatore*: un sistema esperto capace di commentare una partita di Scacchi alla maniera dei Maestri. In [Michie 81] questa funzionalità è affidata al polinomio standard, in base a tecniche di analisi statistica sull'albero del gioco. In un contesto knowledge based, quale è il nostro, il modulo commentatore può offrire un'interfaccia maggiormente orientata all'utente. Non va inoltre sottovalutato il fatto che la vasta letteratura specializzata esistente permette di eseguire la fase di tuning del sottosistema commentatore con un grande grado di precisione.

3. Conclusioni

Mentre il sistema di regole ufficiali del Gioco è tuttora soggetto a raffinamenti, esiste d'altra parte un insieme di varianti eterodosse, alcune delle quali sono largamente praticate dagli esperti. Il Kriegspiel, per esempio, è una variante nella quale i due avversari si affrontano senza avere informazione completa sullo stato della scacchiera, ma solo sulla posizione dei propri pezzi; quella nemica è solo ipotizzata. Un arbitro gestisce la posizione reale, e controlla la legalità delle mosse. I giocatori debbono dedurre lo stato del gioco in base alle informazioni date dall'arbitro sulla legalità delle loro mosse. L'Analisi Retrograda è un'altra specialità fortemente legata alla logica. Data una posizione, si tratta di dimostrare uno o più teoremi, note alcune premesse [Smullyan 79]. Le più interessanti varianti eterodosse possono agevolmente essere trattate nel nostro ambito, descrivendole ciascuna con una teoria che le formalizza. Tali teorie sono in grado di vedere come sottoteorie i moduli già esistenti.
Questa digressione sui Giochi Eterodossi vuole dimostrare che il Gioco degli Scacchi continuerà ad offrire un campo di ricerca all'Intelligenza Artificiale, anche se alcuni ricercatori dichiarano di essere vicini all'apparente obiettivo finale: costruire un programma capace di battere il Campione del Mondo. Il nostro obiettivo è diverso: costruire un sistema capace di dialogare sugli Scacchi come farebbe un esperto umano. Uno degli obiettivi principali del progetto METAChess è la costruzione di un sistema esperto capace di commentare una partita di Scacchi in modo magistrale. L'ambiente deve anche essere adattabile a contesti differenti dal gioco ortodosso.
La scelta di usare il PROLOG ci è sembrata naturale, soprattutto in seguito all'esperienza avuta da uno degli autori con l'ambiente MRS

[Genesereth 80]. Le principali limitazioni del PROLOG che abbiamo dovuto superare sono l'assenza di meccanismi di forward reasoning e l'impossibilità di partizionare la base di conoscenze in teorie separate. L'implementazione di un meccanismo di inferenza "in avanti" non ha presentato difficoltà, mentre l'utilizzazione del predicato predefinito

record(NomeTeoria, Clausola, Puntatore)

per partizionare la base di conoscenze si è rivelata insoddisfacente. Abbiamo infatti sentito la mancanza di potenti meccanismi di manipolazione di teorie: innanzitutto assert e retract di interi insiemi di fatti, ma anche metateorie per specificare regole di inferenza non standard e meccanismi di combinazione di teorie.

Bibliografia

[Aiello 84] L.Aiello e G.Levi, "The Uses of Metaknowledge in AI Systems, Proc. European Conf. on AI, Pisa, 1984, 705-717.

[Berliner 86] H.Berliner e C.Ebeling, "The SUPREM Architecture: A New Intelligent Paradigm, *Artificial Intelligence*, 28, 1986, 3-8.

[Bratko 86] I.Bratko, "PROLOG and Artificial Intelligence", AddisonWesley, 1986.

[Gaspari 86] M.Gaspari, "Sintesi di obiettivi nel Gioco degli Scacchi, Tesi, Pisa, 1986.

[Genesereth 80] M.R.Genesereth, "The MRS Dictionary", TR.HPP8024, Stanford Heuristic Programming Project, 1980.

[Hartston 86] W.Hartston, Artificial Stupidity, in D.Beal (ed.), Advances in Computer Chess 4, Pergamon, 1986, 52-59.

[Michie 81] D.Michie, Evalutative Comments in Chess, *The Computer Journal*, 24, 1981, 278-286.

[Ponzetto 85] P.Ponzetto e R.Bellin, Test Your Positional Strenght, Batsford, 1985.

[Smullyan 79] R.Smullyan, The Chess Mysteries of Sherlock Holmes, Knopf, 1979.

[Somalvico 83] M.Somalvico, "A Dynamical Notion of Evaluation Function in Chess Playing, Proc. 2nd Conf. on AI and Chess, Milano, 1983, 15-41.

[Thompson 82] K.Thompson, Computer Chess Strenght, in M.Clarke (ed.), Advances in Computer Chess 3, Pergamon, 1982, 55-56.

UN PROGETTO DIDATTICO PER L'INSEGNAMENTO DEL PROLOG NELLA SCUOLA ITALIANA DELL'OBBLIGO

Giorgio CASADEI[†], Antonio MOSCIANESE[†], Aldopaolo PALARETI[‡]

[†]Dipartimento di Scienze Statistiche – Università di Bologna
[‡]Istituto di Informatica – Università di Ancona

RIASSUNTO

In questo lavoro vengono presentate due esperienze in corso nelle scuole dell'obbligo italiane.

La prima è condotta in collaborazione con il comune di Bologna e il Dipartimento di Scienze dell'Educazione dell'Università di Bologna e riguarda la scuola elementare; la seconda viene portata avanti con alcune scuole medie.

I progetti prevedono l'insegnamento del linguaggio PROLOG ai docenti coinvolti, lo sviluppo di alcuni programmi che trattano argomenti attinenti le varie discipline insegnate nella scuola e l'insegnamento agli studenti del linguaggio PROLOG mediante la illustrazione progressiva dei programmi applicativi sopra ricordati.

0 – INTRODUZIONE

Lo scopo che ci si prefigge con il progetto illustrato nel presente lavoro è quello di sperimentare una strategia di introduzione dell'Informatica nella Didattica con particolare riferimento ai seguenti punti:

- valutare il contributo che l'apprendimento di un linguaggio di programmazione fornisce allo sviluppo delle strutture cognitive dei bambini;
- misurare l'efficacia dell'uso di software didattico nell'insegnamento e nell'apprendimento di varie discipline.

In prospettiva il progetto descritto vuole evidenziare l'efficacia e mettere a punto le modalità di utilizzo del PROLOG e di prodotti per ITS (Intelligent Tutoring Systems) in un percorso didattico interdisciplinare di tutto il ciclo della scuola dell'obbligo.

La scelta del PROLOG come linguaggio comune per tutto il progetto è dovuta a varie considerazioni.

Innanzitutto il PROLOG è un linguaggio che può essere usato a vari livelli di complessità, soprattutto a causa del suo aspetto descrittivo per cui si "adatta" con naturalezza alla difficoltà dei vari problemi che si devono trattare; con questa scelta si evita quindi la necessità di ricorrere a

linguaggi diversi nelle diverse fasi di un curriculum co ampio.

Il PROLOG inoltre, avendo un aspetto descrittivo, si adat meglio dei linguaggi procedurali alle esigenze didattiche d bambini, per i quali è fondamentale lo sviluppo delle capac tà cognitive. Questa caratteristica del PROLOG fa sì che bambino diventi un "esploratore" del mondo e immagini model di comportamento che può verificare realmente in quanto gli strumenti per implementarli in modo autonomo.

Un ulteriore merito di questo linguaggio infatti è che sua sintassi è elementare e non richiede un addestramen lungo e fuorviante rispetto agli scopi del progetto didatt co.

Infine non si deve dimenticare che il PROLOG permette realizzare programmi complessi e di affrontare progetti Intelligenza Artificiale risultando in tal modo uno strumen unico per coprire l'intero spettro delle esigenze pos dall'introduzione dell'Informatica nella Didattica.

1 — CURRICULUM PER LE SCUOLE ELEMENTARI

1.1 — L'introduzione dell'Informatica nelle scuole elementa non deve avere come obiettivo specifico l'apprendimento di linguaggio di programmazione, ma quello di indurre un mo generale di affrontare i problemi, favorendo l'apprendimen di modelli cognitivi multidisciplinari. Pertanto l'introd zione degli elementi del linguaggio non deve essere fine a stessa, ma deve integrarsi in modo determinante con trattamento di esempi e casi di prova tratti da altre disc pline e dalle esperienze quotidiane dei bambini.

In questa età gli studenti non hanno ancora acquisito cap cità analitiche tali da permettere la realizzazione di p grammi complessi: sono tuttavia in grado di raccogliere correlare informazioni senza tuttavia saperle coordinare maniera sofisticata.

Pertanto il progetto di introduzione del PROLOG nel elementari qui descritto è orientato alla gestione di sempl ci data base di informazioni e alla descrizione di sempl strategie per giochi tipici di questo livello di età. questo punto di vista è necessario introdurre solo un num limitato di predicati predefiniti (parassiti), in quanto utilizza quasi esclusivamente l'aspetto logico descritt del PROLOG. I parassiti che coinvolgono sideeffect, in par colare quelli per l'input e per l'output, vengono introdo solo per rendere ragione delle interazioni con il mo esterno indispensabili per la soluzione dei casi di pr proposti. È invece fondamentale introdurre i concetti lista e di struttura e realizzare la ripetizione per me della ricorsività. La sintassi da insegnare si limita qui ai seguenti oggetti:

- fatto (oggetto del data base senza variabili);
- clausola (oggetto del data base che coinvolge l'implicazione e l'uso di variabili);

- regola (famiglia di fatti e clausole aventi lo stesso nome);
- operatori aritmetici e "is";
- "not";
- "findall";
- "read", "write" e "nl";
- sintassi delle liste (parentesi quadre e barra).

Come si vede la sintassi richiesta è molto semplice e i simboli che devono essere riconosciuti sono poco più di una decina compresi parentesi e punteggiatura.

1.2 – È opportuno che l'insegnamento di linguaggi di programmazione nella scuola elementare comprenda l'uso di componenti ludiche. Tali componenti devono essere relative sia all'argomento dei problemi trattati sia al modo in cui tali problemi sono affrontati; questo secondo aspetto è più legato alle metodologie didattiche che a quelle informatiche per cui non viene approfondito in questa presentazione.

L'aspetto ludico relativo ai problemi da trattare viene coinvolto scegliendo opportuni giochi di cui si chiede di formalizzare le regole e una opportuna strategia di comportamento.

Le problematiche connesse con tale progetto vengono quindi affrontate in fasi successive, definite nel curriculum, individuate dall'insegnante in modo da guidare l'attenzione degli studenti alla conoscenza delle strategie di gioco; tale conoscenza si evolve con il crescere dell'esperienza che si acquista giocando. A titolo di esempio viene descritto il modulo didattico relativo al gioco della sottrazione, che consiste nel togliere alternativamente da un numero dato (per esempio 21) un intero compreso tra 1 e 5: perde il giocatore che è costretto a togliere l'ultimo numero.

Le fasi individuate dal curriculum sono:

- descrizione e sperimentazione del gioco;
- sua formalizzazione e programmazione;
- introduzione di conoscenza sulla strategia:
  strategia a) sottrazione di un numero a caso;
  strategia b) individuazione della scelta corretta nell'ultima mossa;
  strategia c) individuazione della scelta corretta nella penultima mossa;
  strategia d) analisi della strategia vincente.

1.3 – Un metodo parallelo per introdurre il PROLOG nella scuola elementare consiste nell'utilizzare tale linguaggio come strumento per gestire informazioni provenienti da altre discipline, quali Geografia, Storia e Scienze. In tal modo il PROLOG viene usato come strumento elementare per la gestione di semplici data base relazionali. Ad esempio un curriculum progettato per una applicazione di Geografia è articolato nelle seguenti fasi:

- proposta del problema: le regioni italiane;
- rappresentazione sotto forma di tabelle dei dati;

- costruzione del data base dei fatti, con possibilità di semplici interrogazioni;
- interrogazioni complesse che richiedono la realizzazione di regole;
- individuazione di problemi non risolubili con le tecniche elementari che richiedano per esempio l'uso di "findall";
- discussione sulla organizzazione delle tabelle e del data base, individuazione delle lacune e dei problemi derivanti dalla mancanza di documentazione;
- intrduzione delle regole di normalizzazione, riorganizzazione e ampliamento del data base;
- possibilità di correlazione tra data base diversi (data base di Geografia e di Scienze per lo studio di problemi di habitat).

In questa ottica il PROLOG viene introdotto e usato come strumento concreto e efficace per realizzare applicazioni interdisciplinari.

1.4 — I problemi fondamentali che pongono difficoltà di tipo concettuale e tecnico per gli studenti sono i seguenti.

a) Restituire uno alla volta tutti gli elementi di una lista. Ciò comporta l'introduzione del predicato risoddisfacibile "member" usato sia in forma *check* che *generate*. Tale predicato deve essere completamente giustificato dall'insegnante e appreso dagli studenti.
b) Congiunzione, disgiunzione e negazione di predicati. Tali problemi devono essere risolti introducendo i predicati "and", "or" e "not"; mentre "and" e "or" vanno definiti in forma di regole PROLOG, si preferisce considerare il "not" come un predicato primitivo.
c) Data una domanda risoddisfacibile che restituisce più risposte, creare la lista che le contiene tutte. Volendo evitare l'uso di side-effect e l'introduzione dei predicati "assert" e "retract", si richiede che il predicato "findall" sia un predicato primitivo. L'uso di "findall" deve essere spiegato agli studenti.

1.5 — Un sottoinsieme di questo proetto può essere pensato come un modo facile e efficace per insegnare elementi di logica e teoria ingenua degli insiemi in seconda e terza elementare.

2 — CURRICULUM PER LE SCUOLE MEDIE INFERIORI

2.1 — Anche nelle scuole medie inferiori l'obiettivo primario dell'insegnamento di un linguaggio di programmazione consiste nel favorire l'evoluzione delle capacità intellettuali degli studenti. Rispetto alle scuole elementari, nelle medie è possibile trattare problemi più complessi, utilizzare la sintassi del linguaggio in maniera più completa (particolarmente per ciò che riguarda i predicati deterministici) e affrontare in modo sistematico il problema della ricerca degli errori nei programmi (sia in quelli propri che in quelli scritti da altri).

I problemi che devono essere trattati durante lo svolgimento di questa parte del curriculum devono prevedere due tipi di evoluzione. In certi casi si richiede a gruppi di studenti di sviluppare progetti complessi in maniera del tutto autonoma; in altri casi può essere pedagogicamente più efficace fornire il nucleo già realizzato di un progetto e chiedere agli studenti di completarlo e di adattarlo alle esigenze specificate nel modulo didattico interessato.

Dal punto di vista sintattico gli elementi che si devono introdurre sono, oltre a quelli già illustrati nel paragrafo relativo all'insegnamento elementare:

- "cut" e predicati deterministici;
- "assert" e "retract" per l'uso dinamico del data base;
- predicati per la gestione dei file;
- "op", "arg", "univ" e i concetti generali di operatore e funtore.

2.2 – Dopo le esperienze "ingenue" delle scuole elementari, nelle quali i metodi di trattamento dei problemi sono usati senza essere resi spliciti, nella scuola media è necessario affrontare la difficoltà dovuta alla necessità di esplicitare i metodi generali per il *problem solving*. A tal fine gli aspetti che devono essere affrontati sono il concetto di algoritmo, la programmazione *top-down*, l'analisi strutturata dei problemi e la realizzazione di una documentazione adeguata. Si ritiene quindi utile l'introduzione di alcuni problemi classici quali:

- la torre di Hanoi;
- il problema delle otto regine;
- il "mastermind";
- il "nim".

Rispetto ai metodi per la trattazione sperimentale dei giochi descritta per la scuola elementare, si aggiungono: l'analisi a tavolino dei problemi, per individuare quando possibile soluzioni algoritmiche, l'analisi dell'albero delle possibili mosse, per riconoscere le strategie migliori dal punto di vista probabilistico, la suddivisione del problema in sottoproblemi che siano risolvibili separatamente, in modo da stimolare un lavoro in gruppi coordinati.

Per raggiungere tali obiettivi si deve introdurre nel curriculum la trattazione di elementi del calcolo delle probabilità e di strutture astratte (alberi e grafi) per l'organizzazione delle informazioni.

2.3 – Anche nelle applicazioni interdisciplinari si possono realizzare progetti che prevedono l'uso di metodi e strumenti più sofisticati. Oltre alla naturale evoluzione dei progetti già descritti per la scuola elementare, che consiste essenzialmente nel realizzare data base con maggiori correlazioni tra i dati (eventualmente dotati di un *data base dictionary*), un progetto che può essere iniziato è rappresentato dall'uso di un *parser* elementare. Tale progetto consiste nella disponibilità, garantita a priori dal progetto, di un *parser* che sia in grado di utilizzare un insieme di fatti contenuti nel data base, allo scopo di analizzare da un punto di vista

sintattico semplici frasi del tipo "il gatto rosso mangia l'elefante artico". In tale progetto si affrontano i seguenti problemi:

- aumento del vocabolario del sistema;
- introduzione di regole per il trattamento di parti del discorso inizialmente non gestite;
- comprensione del funzionamento di un *parser* elementare e sua modifica per trattare frasi più complesse;
- individuazione delle incongruità e inesattezze del *parser* (riconoscendo la distinzione tra sintassi e semantica).

2.4 — I problemi concettuali e tecnici che devono essere affrontati sono:

a) concetto di determinismo: uso del "cut", struttura di tipo *case*, definizione del "not" e dell'"if";
b) gestione del data base: uso di "assertz" e "asserta" per simulare strutture di tipo *FIFO* e *LIFO* e definizione del "findall";
c) *tail recursion*: discussione del problema della correttezza dei programmi e dell'efficienza tramite esempi;
d) gestione delle informazioni su memoria di massa e uso del sistema operativo;
e) strumenti per il *debugging*: "trace" e "spy".

3 — PROSPETTIVE

Attualmente sono in corso di sperimentazione alcuni "software didattici" nelle scuole medie (su argomenti di Geometria Piana e Solida) e un corso di insegnamento del PROLOG nelle elementari. Si stanno ioltre mettendo a punto criteri per la valutazione delle esperienze in corso ed è in fase di progetto la realizzazione di altre applicazioni interdisciplinari per le scuole medie (Geografia e Italiano) e quella di altri giochi per l'approfondimento del PROLOG ("mastermind" e "nim").

L'introduzione dell'Informatica fin dalle scuole elementari consente lo svolgimento di un programma che a prima vista può sembrare ambizioso. Uno degli obiettivi del presente progetto è anche quello di verificare quanto è effettivamente insegnabile (e apprendibile) con una trattazione organica.

Attualmente è in atto una riorganizzazione della scuola che prevede di inserire nel ciclo dell'obbligo il primo biennio delle scuole medie superiori. Al termine della sperimentazione prevista dal presente progetto sarà possibile formulare proposte concrete per l'introduzione dell'Informatica anche in queste classi. Si può comunque fin da ora prevedere che, oltre agli aspetti formativi e interdisciplinari, si porrà l'esigenza di introdurre l'Informatica come disciplina autonoma.

SISTEMA ESPERTO PER DIAGNOSI AVARIE IN UN TRASMETTITORE RADAR

Bruno Cardile*, Riccardo Filippi¨, Roberto Lalle¨, Franco Turini°

*Tecsiel S.p.A. - ¨Selenia S.p.A. - °Università di Pisa

SOMMARIO

E' stato realizzato un prototipo di sistema esperto in grado di eseguire la diagnosi delle avarie avvenute in un trasmettitore radar. I dati in ingresso al sistema sono gli stessi disponibili allo specialista incaricato della riparazione. Il funzionamento del sistema è off-line rispetto a quello del trasmettitore. Il linguaggio utilizzato per l'implementazione è il PROLOG e la macchina su cui è stato sviluppato è una SUN 3. Il sistema utilizza una tecnica di diagnosi basata su un modello astratto del dispositivo fisico; la sua architettura comprende, di conseguenza, un modulo di descrizione fisica ed uno di descrizione funzionale. Il sistema permette una diagnosi veloce delle avarie e costituisce una importante base di conoscenza flessibile e facilmente accessibile a chiunque.

1. INTRODUZIONE

La scelta del trasmettitore tra i sottassiemi di un sistema radar su cui fare questa sperimentazione è dovuta alla sua complessità e quindi alle difficoltà che si incontrano per individuare le cause del guasto. Il trasmettitore radar è un apparato elettronico essenzialmente analogico nel quale sono impiegate tensioni molto elevate e sono generate grosse potenze elettriche. Ciò fa si che non si possano eseguire tutti i controlli necessari durante il funzionamento. Tra le avarie più complesse da risolvere ci sono quelle dovute al verificarsi di impreviste scariche elettriche. I valori di tensione elevati generano tali scariche quando si verificano perdite di isolamento in qualche componente.
L'individuazione di tale componente, in generale, non è immediata in quanto non è semplice verificare una perdita di isolamento.

In aiuto allo specialista incaricato di riparare la macchina ci sono dei sensori all'interno del trasmettitore che inviano i loro segnali a carte di elaborazione. In base ai dati ricevuti, queste carte presentano un numero sul BITE (Built In Test Equipment) display, accendono o meno certe lampadine ed inibiscono generalmente il funzionamento del trasmettitore. La diagnosti-

ca cablata, però, solo per certi guasti riesce a fornire l'informazione sufficiente all'utente ad individuare il componente che è andato in avaria. Nella maggior parte dei casi le informazioni fornite sono dati indicanti l'area del trasmettitore in cui è avvenuto il guasto.

E' stato scelto di realizzare un sistema che avesse a disposizione le stesse informazioni disponibili attualmente allo specialista per poterne sfruttare l'esperienza. E' stata esclusa qualsiasi interazione diretta con il trasmettitore.

Le prestazioni di questo prototipo sono all'esame degli specialisti ed i risultati che fornisce sono ritenuti industrialmente soddisfacenti.

2. CARATTERISTICHE GENERALI DEL SISTEMA ESPERTO

La caratteristica principale di questo sistema risiede nella particolare tecnica di diagnosi impiegata. Essa consiste nel ragionare a partire da una descrizione fisica e funzionale del trasmettitore. Tale scelta, anche se comporta una maggiore complessità computazionale rispetto ad un approccio basato su "conoscenza compilata", permette di affrontare guasti per i quali i sintomi non sono nè esplicitamente previsti nè univoci. Questo è ciò che accade per il trasmettitore dove per i guasti più complessi è impossibile associare a determinati malfunzionamenti esattamente le parti da sostituire. L'architettura è stata ideata cercando di tenere separati il più possibile i diversi livelli di conoscenza forniti al sistema e tenendo presente i requisiti di modificabilità ed estendibilità richiesti per applicazioni di questo tipo. Anche se l'obiettivo di questa attività è stato quello di valutare l'applicabilità e l'efficacia di un sistema esperto diagnostico in campo radar, l'architettura del sistema vuole rappresentare una proposta sufficientemente generale da essere estesa ad altre realizzazioni di ES di "fault diagnosis".

3. IL PROCESSO DI DIAGNOSI

Il processo può essere diviso in due passi successivi; il primo, indirizzato dal numero presente sul BITE display individua l'area del trasmettitore in cui è avvenuto il guasto. Il secondo, analizzando il modello fisico e richiedendo altre informazioni all'utente cerca di individuare il componente guasto. Il sistema individuata l'area cui appartiene il componente in avaria, sceglie tra due possibili strategie di diagnosi. In un caso per ciascuno dei componenti presenti nell'area in questione chiede informazioni all'utente sui segnali presenti in ingresso ed in uscita. Se c'è discrepanza tra tali dati e le specifiche

di funzionamento contenute nel modello funzionale dell'apparato, il componente è indicato come guasto ed il processo di diagnosi si ferma, altrimenti passa ad un altro componente. Poichè, però, nel caso del trasmettitore non è possibile per tutti i componenti conoscere i segnali di ingresso e di uscita per dedurne lo stato di funzionamento, il sistema è stato dotato della capacità di eseguire una strategia di diagnosi basata su ragionamento approssimato. Tale strategia consiste nel comunicare all'utente i componenti ritenuti sospetti ordinati valutando opportunamente le loro caratteristiche di affidabilità, reperibilità, semplicità di sostituzione e costo dell'operazione.

4. DESCRIZIONE FUNZIONALE DEI MODULI DEL SISTEMA

L'architettura del sistema Fig. 1 è costituita da un insieme di moduli ad ognuno dei quali è associata una determinata funzionalità, nel seguito brevemente descritta.
Il "modulo raccolta sintomi" raccoglie i dati resi disponibili dalla diagnostica cablata.
Tali dati trasmessi al "modulo di controllo della diagnosi" servono ad individuare il sottoassieme nel quale si può ritenere confinato il guasto. L'interprete del modello esegue la diagnosi analitica utilizzando le descrizioni strutturali e funzionali implementate nei moduli di descrizione fisica e funzionale.
Al "modulo di filtraggio" e informazioni statistiche è demandata la funzione di prendere in ingresso una lista di componenti e restituirne un'altra ottenuta dalla prima ordinando i componenti in base alle loro caratteristiche di affidabilità, costo, semplicità di sostituzione.
Il "modulo di guida alla riparazione" contiene conoscenze di tipo euristico circa le azioni da intraprendere una volta che un componente è riconosciuto guasto.

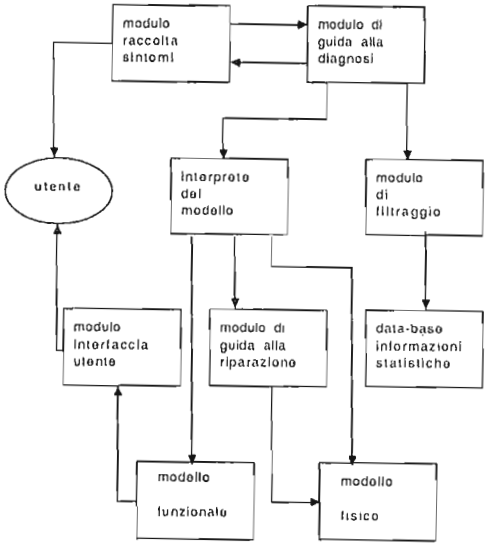


fig. 1

5. CENNI SULLE SCELTE IMPLEMENTATIVE

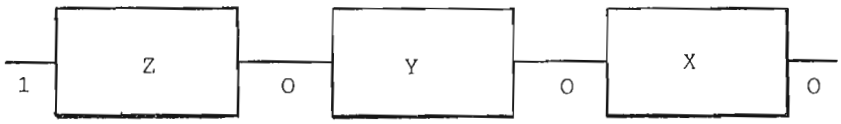
Il modello fisico è stato implementato mediante la definizione di predicati del tipo:

```
connesso_avanti(comp1,comp2).
connesso_dietro(comp3,comp4).
```

La struttura implementata per realizzare il modello fisico è dunque un grafo orientato in cui i nodi rappresentano i componenti e gli archi le connessioni fisiche tra di essi.
Per ciò che riguarda il modello funzionale è stato scelto di realizzare una astrazione sullo stato dei componenti all'interno del dispositivo fisico. Essa consiste nel rappresentare i valori presenti in ingresso ed in uscita ai componenti facendo uso di una semplice codifica binaria.
La procedura di diagnosi è definita dai seguenti predicati:

```
path(stop):-!.
path(X):-guasto(X),ripara(X).
path(X):-connesso_avanti(X,Y),path(Y).
guasto(X):-legaingr(X,Y),legausc(X,Z),!,Y/=Z.
```

L'esecuzione della procedura di diagnosi può essere vista, quindi come la generazione da parte del sistema, di un modello del dispositivo e del suo funzionamento schematizzabile dal seguente esempio:



In questo caso il sistema ha generato, nell'ordine, i componenti X,Y,Z ed ha assegnato loro una codifica degli ingressi. In particolare 0 indica che il valore misurato non è quello corretto, 1 il viceversa. In questo caso il sistema riconosce il componente Z come responsabile del guasto, poichè pur ricevendo in ingresso i valori attesi, presenta in uscita valori non conformi alle sue specifiche di funzionamento.
Poichè i vari pezzi di conoscenza, che costituiscono l'architettura del sistema, sono stati assemblati in una struttura a moduli comunicanti tra loro, è stato implementato un meccanismo di comunicazione di tipo "blackboard" basato sull'uso dei predicati assert e retract.

6. ESEMPIO DI DIAGNOSI

Nella Fig. 2 è indicato il processo di diagnosi relativo alla presentazione del numero 26 sul BITE display. Le risposte dell'utente sono precedute da : . Il sistema chiede una serie di

informazioni relative a componenti individuati usando il modello fisico. Da tali dati e dall'uso del modello funzionale suggerisce la sostituzione dell'alimentatore. Rimanendo l'avaria il sistema suggerisce la sostituzione dei carichi di tale alimentatore per individuare la causa del corto circuito.

7. CONCLUSIONI

Per concludere la descrizione delle scelte adottate nella realizzazione del sistema esperto proposto, è opportuno valutare se ed in che termini è stato raggiunto l'obiettivo di realizzare una architettura sufficientemente modulare e flessibile.

Gli aspetti caratterizzanti il sistema sono essenzialmente due: il primo è quello di aver suddiviso la conoscenza in moduli separati interagenti tra loro; il secondo è rappresentato dall'aver dotato il sistema di strategie diagnostiche essenzialmente indipendenti dal particolare dispositivo che è oggetto dell'applicazione.

L'organizzazione, che si è voluta dare alla conoscenza del sistema, garantisce l'espandibilità della sua struttura complessiva mediante l'aggiunta di nuova conoscenza ed inoltre rende possibile modifiche della funzionalità in modo che esse restino confinate all'interno del modulo in cui sono state definite e non abbiano alcun impatto sul resto dell'architettura.

Per quanto concerne il secondo aspetto, cioè quello riguardante la realizzazione delle procedure diagnostiche, la loro caratteristica principale è quella di essere procedure generali la cui funzione consiste nel "navigare" su un opportuno modello del dispositivo.
In tal modo appare del tutto naturale l'applicazione di una architettura così concepita alla diagnosi di dispositivi di diversa natura, previa una opportuna formalizzazione del modello (fisico e funzionale) del nuovo apparato. Tale caratteristica di flessibilità è un requisito auspicabile anche all'interno della nostra applicazione dato che i radar sono essenzialmente dei prototipi e come tali soggetti ad evolvere nel tempo.

Altro aspetto coinvolto molto da vicino nella realizzazione delle procedure diagnostiche, è quello della "potenza diagnostica" conferita al sistema. A tal proposito si può affermare che, al contrario del sistema basato su "conoscenza compilata", un sistema di diagnosi guidato dal modello ha la possibilità di operare, e quindi di dare una risposta, in presenza di qualsiasi configurazione di sintomi in ingresso. Chiaramente l'affidabilità delle sue risposte dipende esclusivamente dall'accuratezza con cui il modello è stato formalizzato. Nel nostro modello del dispositivo è stato scelto di inserire la descrizione dei componenti dell'apparato sostituibili in blocco, di conseguenza il sistema risulta in grado di isolare il guasto all'interno di

sottoinsiemi più o meno grossi, di cui è possibile suggerire un insieme sufficientemente ristretto di azioni da intraprendere per la riparazione.

Questo tipo di architettura del sistema esperto ha avuto influsso anche sul sistema di spiegazione, che, invece di limitarsi, come avviene di solito, a parafrasare le regole Prolog applicate, è centrato sul modello e permette di evidenziare quali sono i componenti inquisiti e perchè. Il modo di spiegazione è realizzato con la ormai tradizionale tecnica della meta-programmazione.

```
?- expert.
Numero di bite presente sul BITE DISPLAY?
:26
AVARIA ALIMENTATORI +-15v O 28v A LIVELLO MASSA.
E' accesa la lampadina avaria TX ?
: si.
Inserisci carta estensione su carta_log_generale3
Sono presenti 28v tra pin 59 e massa ?
:no.
Sono presenti +15v tra pin 37 e mas. -15v tra pin 47 e mas.?
:si.
E' bruciato il fusibile_f13_PCDR?
:no.
E' presente tens. 115v su A5TB1 (1-2)?
:si.
Sostituisci il componente: alim_28v_A5
E' ancora presente il guasto?
:si.
Sconnetti uno per volta i seguenti componenti
che possono aver causato un corto circuito:
***********************************
pilota_tx_A4
carta_log_generale2_A14A1A1A4
***********************************
Alla fine se il guasto è ancora presente controlla se c'è
cortocircuito nelle connessioni.
HO FINITO LA MIA DIAGNOSI
```

Fig. 2

BIBLIOGRAFIA

/1/ D. Waterman: "A Guide to Expert System" Addison-Wesley 1985.

/2/ F. Hayes-Roth, D. Waterman, D. Lenat: "Building Expert Systems" Addison-Wesley 1983.

/3/ W. Clocksin, C. Mellish: "Programming in Prolog" Springer-Verlag 1981.

/4/ M.R. Genesereth: "The use of design descriptions in automated diagnosis" Artificial Intelligence 24 (1984) 411-436.

/5/ R. Davis, H. Shrobe: "Representing structure and behaviour of digital hardware" IEEE Computer (1983) 75-82.

/6/ R. Davis: "Reasoning from first principles in electronic troubleshooting" Int. J. Man-Machines Studies 19 (1983) 403-423.

/7/ R. Davis: "Diagnostic reasoning based on structure and behaviour" Artificial Intelligence 24 (1984) 347-410.

/8/ W. Thompson, P. Johnson, J. Moen: "Recognition-based diagnostic reasoning" Proceedings 8 IJCAI (1983)236-238.

/9/ M. Billmers, M. Swartwour: "AI-SPEAR: computer failure analysis tool" Proceedings 6 European Conference on Artificial Intelligence (1984) 65-73.

/10/ G. Vesonder, S. Stolfo, J. Zielinski, F. Miller: "ACE: an expert system for telephone cable maintenance" 8 IJCAI (1983) 116-121.

/11/ R. Cantone, F. Pipitone, W. Lander: "Model-based probabilistic reasoning for electronics troubleshooting" 8IJCAI 207-211.

/12/ G. Scerch: "High power solid state modulator for coherent agile microwave amplifier" 11 Modulator Symposium Conference Board 1973.

/13/ M. Fox, S. Lowenfeld, P. Kleinosky: "Techniques for sensor-based diagnosis" Proceedings 8 IJCAI (1983) 158-163.

ESTA: UN SISTEMA ESPERTO PER LA VERIFICA DI REGOLE DI "DESIGN FOR TESTABILITY"

Paolo Gianoglio
Laboratorio Intelligenza Artificiale - Csi-Piemonte
C.so Unione Sovietica 216 - 10134 Torino - Tel. (011) 3307387

Renato Gianoglio - Centro Ricerche Fiat
Strada Torino 50 - 10098 Orbassano (To) - Tel. (011) 9023260

SOMMARIO

Il testing dei circuiti VLSI sta diventando sempre più importante; diverse metodologie di progetto che garantiscono chip "facilmente testabili" sono state introdotte. La verifica manuale di testabilità è però difficile per la complessità dei circuiti; si rende quindi necessaria ed essenziale una verifica automatica. In questo articolo viene presentato un prototipo di S.E. chiamato ESTA (Expert System for Testability Automation) per la verifica di un circuito secondo determinate regole di progetto. La complessità del dominio ha richiesto l'utilizzo di più paradigmi di rappresentazione della conoscenza, opportunamente integrati ed un meccanismo di controllo ad hoc. Il lavoro, sia pur rivolto ad una particolare applicazione, dimostra come è possibile definire ed implementare uno schema di rappresentazione che combina in modo flessibile Frame e Regole di Produzione in ambiente Prolog.

1. INTRODUZIONE

La crescente complessità dei circuiti integrati impone l'uso di tecniche automatiche di controllo del funzionamento sempre più complicate. Con l'avvento dei circuiti integrati ad alta integrazione (VLSI) si sono quindi introdotte delle tecniche di progetto che permettono un facile accesso alle varie parti di un VLSI e inoltre garantiscono a priori che tutta una serie di problemi tecnici, come ad esempio ritardi di logiche, etc., possano non essere presi in considerazione quando si seguono certe metodologie di progetto. E' così nata la Design For Testability (DFT) (Will-80). I metodi esaustivi adottati finora per realizzare strumenti di verifica automatica delle regole di progetto di DFT (WiPa-82), sono computazionalmente inefficienti. L'approccio che si propone in questo articolo consiste nell'emulare la conoscenza ed il comportamento di un esperto di testabilità con uno strumento CAD (AgJs-84), (Agra-84), (Hors-84), (AbBr-85).

Questo lavoro è stato sviluppato come tesi di laurea presso il Dip. di Inf. dell'Università di Torino ed il Dip. di Automatica ed Inf. del Politecnico di Torino.

2. IL SISTEMA ESTA

Esta è un sistema Ibrido "Frame + Regole di Produzione" (FiKe-85) ed è stato sviluppato in Prolog (ClMe-81), sia per le caratteristiche del dominio, sia per la possibilità di rapido prototyping offerte da questo ambiente.
Il Frame (Mins-75) è utilizzato per la descrizione gerarchica del circuito mentre nel formalismo a Regole di Produzione è stata rappresentata la conoscenza della Design For Testability. L'attivazione dei Frames e l'esecuzione delle Regole di Produzione sono operazioni gestite da un meccanismo di controllo (strategia di controllo) completamente separato dalla conoscenza sul dominio ed indipendente dal dominio (CGGP-86b).

3. DESCRIZIONE DEI CIRCUITI

In ambiente CAD l'hardware viene descritto mediante gli Hardware Description Languages (BaUe-85), i quali possiedono una notevole capacità espressiva e chiarezza. Tuttavia gli HDLS non possono essere utilizzati con tecniche di Intelligenza Artificiale poichè la loro semantica è operativa, orientata alla simulazione e non supporta il ragionamento. Per questo motivo è stata scelta una forma predicativa per la descrizione dell'hardware ed in particolare le clausole di Horn. Non solo, la verifica di testabilità richiede una descrizione dell'hardware a livello principalmente strutturale. Una rappresentazione sotto forma di fatti si presenta adeguata.
In fig. 2 sono elencati i predicati che descrivono i componenti elementari: reti combinatorie (NET) e shift-registers (SRL). Il nome del predicato identifica il tipo di componente e i suoi argomenti stabiliscono le connessioni tra i componenti. Altri predicati distinguono particolari input/output.
La fig. 3 illustra come si combinano le clausole Prolog che descrivono i componenti primitivi con i frames per mantenere la descrizione modulare e gerarchica, riportando un circuito (comparator) costituito da due istanze dello stesso circuito composto (register). In particolare i predicati "value" contengono i nomi delle linee di I/O e la metodologia secondo la quale il circuito è stato progettato; segue quindi la descrizione del circuito mentre il predicato "a-kind-of-list" è il puntatore alla descrizione del circuito composto (register).

4. DESCRIZIONE DELLE REGOLE DI PROGETTO

Delle diverse metodologie per la testabilità dei circuiti elettronici è stata studiata in particolare la LSSD (Level Sensitive Scan Design) (Will-80), introdotta dall'IBM, perchè più significativa interessante. Le regole di progetto LSSD sono del tipo if "cond"

then "action" dove "cond" definisce le condizioni di applicabilità della regola e cioè descrive una determinata configurazione di componenti elementari, mentre "action" è il test sui collegamenti tra componenti elementari.
Queste regole sono state trasposte in forma di clausole Prolog che ritornano vero se la corrispondente regola della DFT è violata. E' possibile riscrivere la regola in modo da ottenere una risposta affermativa solo quando la corrispondente regola della DFT è soddisfatta. Questo approccio è molto più complesso e riduce la leggibilità dell'output dell'analisi.
In fig. 4 è riportato un esempio di regola LSSD in forma estesa, schematica e in clausole Prolog. La clausola "lssd22" individua la particolare configurazione descritta nelle condizioni di applicabilità, mentre la clausola "case" implementa il test. Facendo backtracking, attivato con l'esecuzione del goal "fail", l'interprete trova tutte le istanze dei predicati NET e SRL, individua cioè tutte le occorrenze della particolare configurazione e quindi localizza nel circuito, i punti dove è necessaria la verifica della regola. E' evidente l'inefficienza di un approccio algoritmico che si muove nella descrizione del circuito in modo cieco e verifica la testibilità in modo esaustivo.

5. STRATEGIA DI CONTROLLO INDIPENDENTE DAL DOMINIO

Per implementare il meccanismo di attivazione dei frames sono state individuate due possibili alternative: il preprocessing e una strategia indipendente dal dominio. La scelta si è orientata ad una strategia di controllo indipendente dal dominio, implementata in un modulo separato e che opera direttamente sulla base di conoscenza. Tale strategia, essendo indipendente dal dominio, può essere facilmente adattata a vari campi di applicazione e ci permette di ipotizzare utilizzi futuri del sistema ESTA in altri dominii. A differenza dell'approccio con il preprocessing, il controllo e la conoscenza specifica del dominio non vengono mescolate.
La conoscenza sulla quale la strategia di controllo opera direttamente deve essere rappresentata nel formalismo riportato in fig. 1 e che ricorda quella proposta da Minsky. Nel caso specifico, la procedura Range fa il test sul valore dello slot. La procedura Default recupera un valore di default e lo assegna allo slot. La If-needed permette la costruzione automatica degli oggetti del dominio senza l'interazione con l'utente. La procedura Working-on manipola la conoscenza memorizzata in uno slot. Le procedure X-evaluation valutano i fatti in condizioni di incertezza. Infine lo slot A-Kind-Of definisce le relazioni gerarchiche tra gli oggetti del dominio.
Un frame visto come nodo di un albero di frames viene attivato in fase di percorrimento in discesa dell'albero ed in fase di risalita. Come nodo di un grafo può essere attivato più volte poichè cammini differenti possono portare allo stesso nodo. E'

ragionevole pensare che in attivazioni successive si debbano attivare sequenze di componenti diverse. In fase di risalita può essere, ad esempio, sintetizzata la spiegazione della soluzione e come questa è stata determinata.
E' l'utente che definisce le sequenze, essendo legate al dominio, e le riporta singolarmente nei predicati:

slot-activation (frame, num. seq., lista comp. frame).

La strategia di controllo ricorda qual è la visita corrente al frame e la corrispondente sequenza da attivare nel predicato:

next-slot-activation (frame,num. seq.,ultima seq.).

Quindi, per attivare un frame, la strategia di controllo legge il "num. seq." dal predicato "next-slot-activation", recupera la corrispondente sequenza dal predicato "slot-activation" e attiva successivamente ogni elemento della "lista comp. frame". Al termine imposta "num. seq." al suo successivo valore.
Il percorrimento dei frames avviene secondo diverse politiche (depth-first, breadth-first e con euristica). Questo è possibile mantenendo a run-time la lista di frames da visitare che viene percorsa opportunamente con l'ausilio di un puntatore. Ad esempio, caricando la lista in testa ed indirizzando sempre il primo elemento si ha la visita depth-first.

6. CONCLUSIONI E SVILUPPI FUTURI

ESTA è stato utilizzato per verificare diversi circuiti. In fig. 5 sono riportati i risultati sperimentali ottenuti su VAX. Attualmente ESTA gira anche su P.C. Dalla nostra esperienza l'approccio alla DFT con le tecniche di I.A. si è dimostrato promettente e può essere di valido aiuto al progettista di VLSI.
L'attenzione prestata in fase di progettazione ed implementazione della strategia di controllo ha permesso di costruire un Frame System con caratteristiche superiori a quelle richieste dal particolare dominio della verifica di testabilità che suggeriscono l'applicazione dello strumento per altri aspetti del CAD elettronico.
Sviluppi futuri consistono nell'integrare il sistema con un traduttore da linguaggi HDL nel formalismo Prolog-like e una interfaccia grafica per realizzare così uno strumento completo per il CAD elettronico.

7. RIFERIMENTI BIBLIOGRAFICI

(WiPa-82) T.W. Williams, K.P. Parker "Design for Testability - a Survey" IEEE Trans. on Comp., C-31, 1, Jan. 1982, 2-15.
(Will-80) T.W. Williams, "Design for Testability" NATO Adv. St. Inst. on Des. Aids for VLSI Circ., Urbino, Italy, 1980.
(ClMe-81) W.F. Clocksin, C.S. Mellish "Programming in Prolog" Springer-Verlag, 1981.
(Pere-84) F. Pereira "C-Prolog User's Manual", EdCAAD, Feb. 1984.

(BaUe-85) M.R. Barbacci, R. Uehara "Computer Hardware Description Languages: the Bridge between Software and Hardware" IEEE Computer, 18, 2, Feb. 1985, 6-8.

(Hors-84) P.W. Horstmann "A Knowledge-Based System Using Design For Testability Rules" 14th Int. Conf. on Fault Tolerant Computing, June 20-22 1984, Kissimmee, USA, 278-284.

(AgJS-84) V.D. Agrawal, S.K. Jail, D.M. Singer "A CAD System for Design for Testability" VLSI Design, October 1984, 46-54.

(Agra-84) V.D. Agrawal "Automation in Design for Testability" Proc. 1984 Custom Integrated Circuits Conference, 159-163.

(AbBr-85) M.S. Abadir, M.A. Breuer "A Knowledge-based system for designing testable VLSI chips" IEEE Design, Aug 1985, 56-68.

(Mins-75) M. Minsky "A Framework For Representing Knowledge" in The Psychology Of Computer Vision, McGraw-Hill.

(Fike-85) R. Fikes, T. Kehler "The Role Of Frame-Based Representation In Reasoning" Comm. ACM, 28, 9, Sept. 1985, 904-920.

(CGGP-86a) P. Camurati, P. Gianoglio, R. Gianoglio, P. Prinetto "ESTA: un sistema esperto per la verifica di regole di DFT" I.R. DAI/CAD n. 22/86 Dip. Aut. e Inf., Politecnico di Torino.

(CGGP-86b) P. Camurati, P. Gianoglio, R. Gianoglio, P. Prinetto: "Un nucleo di sistema esperto indipendente dal dominio" I.R. DAI/CAD n. 23/86 Dip. di Aut. e Inf., Politecnico di Torino.

8. APPENDICE

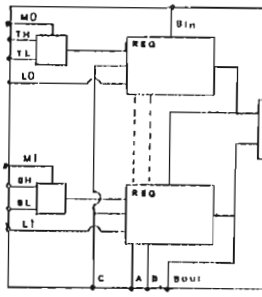
```
frame(frame).
  triggers(frame) :- espr. bool.
  triggers-evaluation(frame, procedura).
slot(frame, slot).
  range(frame, slot) :- espr. bool.
  range-evaluation(frame, slot, procedura).
  default(frame, slot) :- body.
  default-evaluation(frame, slot, procedura).
  if-needed(frame, slot) :- body).
  if-needed-evaluation(frame, slot, procedura).
  working-on(frame, slot) :-  body.
  working-on-evaluation(frame, slot, procedura).
a-kind-of-mode(frame, depth/width/heuristic/best-first).
a-kind-of-list(frame, lista frame in relaz. a-kind-of).
```

- Figura 1 -

```
net(circuito,[in dati],[in clock],[out dati],out clock).
srl(circuito,in dati,in clock,in scan,in clock,in clock,out dati).
blk(circuito,[in dati],[in clock],[out dati]).
pic(circuito,in clock).
scanin(circuito,in dati).
scanout(circuito,out dati).
```

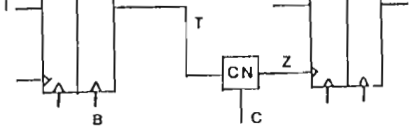
- Figura 2 -

```
frame(comparator).
value(comparator,input,[th,tl,sh,sl,sin,m0,m1,l0,l1]).
value(comparator,inputc,[c,a,b]).
value(comparator,output,[m,e,l,sout]).
value(comparator,dft-type,lssd).
pic(comparator-db,c).
pic(comparator-db,a).
pic(comparator-db,b).
scanin(comparator-db,sin).
scanin(comparator-db,sout).
net(comparator-db,[m0,th,tl],[*],[o0],*).
net(comparator-db,[m1,sh,sl],[*],[o8],*).
net(comparator-db,[a0,sout],[*],[m,e,l],*).
blk(comparator-db,register,[o8,a0,l1],[c,a,b],[sout]).
blk(comparator-db,register,[o0,sin,l0],[c,a,b],[a0]).
a-kind-of-list(comparator-db,[register]).
```



- Figura 3 -

CONDIZIONE DI APPLICABILITA': "due SRL e una rete connessi
che l'in del secondo SRL è controllato dall'out del primo S
TEST: "l'in di clock del primo SRL deve essere uguale ad
della rete affinchè la regola LSSD22 non sia verificata".




```
lssd22(Exp) :-
    net(Exp,[T|[]],[C|[]],_,Z),
    srl(Exp,_,Z,_,_,_,_),
    srl(Exp,I,_,_,_,B,T),
    case(Exp,B,C),
    fail.
```

```
case(Exp,B,C) :-
    pic(Exp,B),
    pic(Exp,C),
    B = C,!.
```

- Figura 4 -

DIVISORE	56	clausole	24ms	(consult)	1.4
CELLA PRIOR.	80	"	40ms	"	5.3
CONTR. VIDEO	219	"	215ms	"	250
AM2901	110	"	47ms	"	97
AM2910	130	"	64ms	"	130
ESTA	300	"	381ms	"	

- Figura 5 -

Parsificazione Parallela in Concurrent Prolog

Giuseppe Giandonato, Corrado Moiso
CSELT
Centro Studi E Laboratori Telecomunicazioni
via Reiss Romoli 274 - 10148 Torino

1. Introduzione

Questo lavoro presenta una versione parallela di una variante dell'algoritmo di Earley [Earley70], per la parsificazione di stringhe in linguaggi generabili da grammatiche context-free. La descrizione dell'algoritmo e` data in uno stile di programmazione *logico-concorrente*, e la corrispondente implementazione, di cui vengono forniti alcuni risultati ottenuti in ambiente Logix [Shapiro86], e` illustrata in Concurrent Prolog (CP), [Shapiro83].

La motivazione principale di questo "esercizio" era quella di verificare la congenialita` del CP, come piu` noto rappresentante della ormai ricca schiera di linguaggi basati sul paradigma logico-concorrente (Parlog, GHC, DL, ...: tra questi il *Flat CP* e` ad oggi l'unico ad essere disponibile in un non rudimentale ambiente di programmazione), al ruolo di linguaggio di *rapid prototyping* per algoritmi paralleli. Tra gli usi possibili rivendicati per il CP, ossia:

- linguaggio di programmazione, destinato ad essere implementato sulle macchine target (eventualmente parallela, come nel progetto Bagel dello stesso Shapiro) per eseguire l'algoritmo in esso codificato;
- linguaggio intermedio, tipicamente per la compilazione in esso di linguaggi della classe Prolog (e` questo come noto l'approccio dell'ICOT con il Flat GHC);
- linguaggio di specifica eseguibile, caratterizzato contemporaneamente da un sapore dichiarativo e da una interpretazione a processi;

e` cioe` il terzo ad essere cosiderato in questo lavoro. In questo tipo di utilizzo in effetti i linguaggi di tipo CP offrono il vantaggio di una notevole concisione, derivata dalla progenitura logica attraverso, ad esempio, il meccanismo di matching/unificazione (e` interessante notare come linguaggi appositamente ideati per prototipazione di sistemi concorrenti, come il recente FP2 [Jorrand86], utilizzino anch'essi tale meccanismo), unitariamente ad una semantica molto semplice e flessibile negli aspetti di comunicazione/sincronizzazione (in particolare e` possibile simulare agevolmente e mischiare message-passing sincrono ed asincrono).

L'occasione e` stata fornita dalla necessita` di validare e valutare in termini di prestazioni, sia asintotiche che sperimentali, un particolare schema di parallelizzazione per l'algoritmo di Earley, che, a differenza di altri noti schemi basati su un numero di processori dipendente dalla lunghezza della stringa d'ingresso, e` in grado di sfruttare un parallelismo fisico corrispondente alla dimensione della grammatica. Cio` puo` risultare particolarmente vantaggioso in applicazioni quali l'analisi sintattica superficiale di linguaggio naturale dove la lunghezza della stringa di simboli in ingresso, in tal caso parole, e` solitamente piccola (tra 10 e 30), mentre il numero di produzioni puo` essere dell'ordine anche di qualche migliaio.

Nei paragrafi seguenti viene derivato e specificato in CP un algoritmo parallelo, corrispondente alla forma cosiddetta *weakened* dell'algoritmo originale di Earley, in cui l'analisi procede come evoluzione delle comunicazioni in una rete (isomorfa alla grammatica) di processi (eventualmente corrispondenti anche uno-a-uno con i processi della macchina fisica sottostante) cooperanti attraverso canali asincroni unidirezionali. E` mostrato come la complessita` nel tempo di tale algoritmo, anche quando eseguito su un singolo processore, e` lineare nella dimensione

della grammatica, in contrasto con quello originale il cui comportamento ha una dipendenz quadratica da tale parametro, rinforzando cosi` l'argomento a favore dello schema parallelizzazione basato su produzioni invece che su simboli d'ingresso.

2. Algoritmo di Earley

Una *grammatica context-free* e` una quadrupla G=(NT,T,Prod,S), dove:

- NT e` un insieme di simboli detti *non-terminali*, dènotati da lettere maiuscole;
- T e` un insieme di simboli detti*terminali*, denotati da lettere minuscole (sequenz possibilmente vuote, di simboli terminali e non-terminali sono denotati da lettere greche);
- Prod e` un insieme di coppie ordinate (P,α), chiamate *produzioni*, indicate come *P<--α* dove P e` il simbolo non-terminale definito dalla produzione;
- S e` un distinto simbolo appartenente ad NT, chiamato simbolo iniziale (o *start symbol*).

L'insieme di produzioni induce una relazione di derivabilita` (==>) tra sequenze di simbo terminali e non-terminali: sia $\alpha A\beta$ una stringa di simboli terminali e di simboli non terminali e s A<--γ una produzione, allora $\alpha A\beta ==> \alpha\gamma\beta$.

L(G) e` l'insieme di tutte le stringhe a1...an di simboli terminali tali che S==>*a1...an (dov ==>* e` la chiusura transitiva e riflessiva della relazione di derivabilita`).

Data una grammatica *G*, *l'algoritmo di Earley* e` un metodo per riconoscere l'appartenenz di una stringa data ad *L(G)* . Esso e` basato sul concetto di *item.*: un item e` una coppia ordina [P<--α.β,i] dove *P*<--αβ e` una produzione di G e *i* e` un indice (0≤i≤n). L'algoritmo calco una successione di insiemi Ij (0≤j≤n) contenenti tutti e soli gli item [P<--α.β,i] che soddisfano seguenti condizioni:

$$0 \le i \le j \text{ and } S ==> {}^*a_1 \ldots a_i P \text{ and } \alpha ==> {}^*a_{i+1} \ldots a_j$$

Una stringa e` riconosciuta appartenere a *L(G)* se e solo se un item della forma [S<--α. ,0] e` I_n.

Un insieme Ij e` inizializzato dall'operazione *Scanner* ed e` chiuso rispetto alle operazi *Predictor* e *Completer*.

Scanner inizializza Ij (j>0) in base agli item in I(j-1):

se [A<--α.Xβ,i] e` in I(j-1) e $X=a_j$ allora [A<--αX.β,i] e` inserito in Ij.

L'insieme I0 e` inizializzato inserendo per ogni produzione *S*<--γ l'item [S<-- .γ,0].

Predictor riguarda gli item in Ij della forma [A<--α.Bβ,i]:

inserisce in Ij [B<--.γ,j] per ogni produzione *B*<--γ, in modo tale da calcolare tutti i valo tali che B ==>* a(j+1)...ak

Completer riguarda gli item della forma [B<--γ. ,k]:

per ogni item [A<--α.Bβ,i] in Ik inserisce in Ij l'item [A<--αB.β,i]

Sia *M* un indice della dimensione della grammatica *G* e *N* la lunghezza della stringa, allora, utilizzano strutture ad accesso diretto (per esempio array), la complessita` in tempo e` $O(M^2$ altrimenti risulta essere $O(M^2N^4)$.

2.1 Parallelizzazione dell'algoritmo di Earley

Nella versione originale l'algoritmo di Earley e` strettamente sequenziale: eseguen pipeline la computazione degli insiemi Ij si ottiene uno scarso grado di parallelismo.

Il risultato migliore nel campo della parallelizzazione` e` legato ad una var

dell'algoritmo chiamata *weakened Earley's algorithm* [Chang84], la quale si basa sulla costruzione di una matrice triangolare {t(i,j) | 0≤i≤j≤n} di insiemi di *dotted-rule* A<--$\alpha.\beta$, dove A<--$\alpha\beta$ e` una produzione della grammatica: t(i,j) contiene $A$<--$\alpha.\beta$ se e solo se $\alpha$==>*ai+1...aj. Una stringa a1...an e` generata dalla grammatica se e solo se una dotted-rule del tipo S<--γ. appartiene a t(0,n).

Utilizzando questa variante dell'algoritmo Chang e Fu riuscirono a realizzare un versione parallela la cui complessita` in tempo e` $O(N)$, avendo a disposizione $O(N^2)$ processori. Questo risultato e` stato possibile perche` la computazione degli insiemi

t(j,j)={ A<--.α | A<--α e` una produzione}

risulta dipendere unicamente dalla grammatica (la creazione di questi insiemi sostituisce il passo di *Predictor*) e non dagli altri insiemi della *j-esima* colonna (che a loro volta dipendono dagli elementi della colonna *(j-1)-esima*). In questo modo, pur computando piu` item, l'algoritmo *weakened* , computando un item del tipo [A<--.$\alpha$,i,i] prima che sia richiesto, possiede un maggior parallelismo di quello del'algoritmo originale: si puo` dire che lo schema computazionale originale *demand-driven* e` stato trasformato in uno*data-driven* .

2.2 Una nuova visione dell'algoritmo

In questa sezione sara` descritta una nuova visione parallela del weakened Earley's algorithm dove il ruolo strutturante e` realizzato dalle produzioni della grammatica, invece che dalla stringa da riconoscere.

Ad ogni produzione A<--X1...Xn e` associato un sistema di processi (agente) con il compito di computare l'insieme

{[A<--X1...Xk.X(K+1)...Xn,i,j] | X1...Xk==>*a(i+1)...aj}

Per uniformita` di definizione chiameremo *item* anche le triple [A<--$\alpha$.$\beta$,i,j] composte da una dotted-rule e da due indici. Il riconoscimento di una stringa lunga $N$ ha successo se e` generato un item della forma [S<--α. 0,N].

L'insieme computato dalla (rete di processi associati alla) produzione A<--$X1...Xn$ e` inizializzato con {[A<--.X1...Xn,i,i] | 0≤i≤N} ed e` completato con le seguenti due operazioni:

- *Scanner*: se l'insieme contiene [P<--X1...Xk.X(k+1)...Xn,i,j], dove X(k+1) e` un simbolo terminale, e se X(k+1)=a(j+1), allora si inserisce [P<--X1...X(k+1).X(k+2)...Xn,i,j+1];
- *Completer*: se l'insieme contiene [P<--X1...Xk.X(k+1)...Xn,i,j], dove X(k+1) e` un simbolo non-terminale, e se un item della forma [X(k+1)<--$\alpha$. ,j+1,h] e` stato inserito nell'insieme associato ad una produzione che definisce X(k+1), allora si inserisce nell'insieme [P<--X1...X(k+1).X(k+2)...Xn,i,h].

Anche in questa visione non c'e` una operazione corrispondente a quella di *Predictor*, poiche` non si aspetta una richiesta per inserire in un insieme un item della forma [A<--.α,i,i].

E` possibile provare che un item [P<--$\alpha$.$\beta$,i,j] e` inserito nell'insieme associato a P<--$\alpha\beta$ se e solo se α==>*a(i+1)...aj; ne segue la correttezza dell'algoritmo, poiche` una stringa a1...an e` riconosciuta appartenere alla grammatica se e solo se e` stato generato un item della forma [S<--$\alpha$.0,n], cioe` se e solo se S ==> α ===>* a1...an.

Per realizzare efficientemente in un sistema distribuito l'operazione di *Completer* e` necessario che al momento dell'inserzione in un insieme di un item della forma [A<--$\alpha$.,i,i] esso sia reso noto agli agenti che computano gli altri insiemi. Questo puo` essere realizzato sia inviandoli con messaggi sia memorizzandoli in una memoria comune: nel seguito sara` adottato il secondo metodo piu` compatibile con il sistema di comunicazione via stream del Concurrent

Prolog.

3. La realizzazione in Concurrent Prolog

E` stata sviluppata una implementazione in *Concurrent Prolog* (CP) [Shapiro83] della versione inerentemente parallela dell'algoritmo di Earley descritta nella sezione precedente.

Seguendo lo stile tipico dello stile di programmazione CP la cooperazione tra processi e` stata realizzata per mezzo di stream, cioe` sequenze finite o infinite di dati, mentre il comportamento (possibilmente non deterministico) dei processi e` descritto per mezzo di insiemi di regole CP (cioe` clausole di Horn con guardie): ogni regola definisce una possibile trasformazione locale del processo, che produce/consuma elementi sugli stream.

Il sistema di processi associato ad una produzione *P<--X1...Xk* consiste in una rete lineare di *k* processi, uno per ogni simbolo Xi, che lavorano in pipeline. Questi processi comunicano per mezzo di canali strutturati come un insieme (chiamato*bundle*) di N+1 stream (numerati da 0 a N), dove N e` la lunghezza della stringa in esame.

Questa realizzazione si basa sul seguente invariante:

> l'indice *i* prima o poi sara` inserito nello stream *j-esimo* del bundle che collega i processi associati a Xh e a X(h+1) se e solo se l'item [P<--X1...Xh.X(h+1)...Xk,i,j] e` inserito nell'insieme associato alla produzione *P<--X1...Xn*.

Per realizzare facilmente l'operazione di *Completer* associamo ad ogni simbolo non-terminale P un bundle, chiamato P-bundle. Per ogni P il corrispondente P-bundle e` il *merge* di tutti i bundle di output dei sistemi di processi associati alle produzioni che definiscono P: lo stream j-esimo del bundle risultante dal merge di due bundle B1 e B2 e` il merge tra i due stream *j-esimi* di B1 e di B2. La presenza dell'indice *i* nello strem *j-esimo* di un P-bundle corrisponde alla generazione di un item della forma [P<--a. ,i,j]. Ne segue che il riconoscimento di una stringa lunga *N* ha successo se l'indice *0* e` presente nello stream *N-esimo* del bundle associato allo start symbol della grammatica.

La fase di inizializzazione degli insiemi di item ha come corrispettivo l'inizializzazione dei bundle di input dei sistemi di processi associati alle produzioni: lo stream *j-esimo* ($0 \leq j \leq N$) e` uno stream finito il cui unico elemento e` l'indice *j*.

La rete associata ad una produzione e` costituita da processi di due tipi chiamati rispettivamente *p-terminal* e *p-nonterminal* : del primo tipo sono i processi corrispondenti ai simboli terminali, mentre del secondo sono quelli corrispondenti ai simboli non-terminali.

Il comportamento di un processo *p-terminal* riproduce l'esecuzione dell'operazioni di *Scanner*. Un tale processo, corrispondente ad una occorrenza del simbolo terminale *a* , oltre allo stream di input e a quello di output ha come parametro la stringa da parsificare e il simbolo terminale *a* : se *a* e` uguale all'*i-esimo* elemento della stringa in esame, esso inserisce nel bundle di output come stream *(i+1)-esimo* lo stream *i-esimo* del bundle di input, mentre in tutte le altre posizioni inserisce uno stream vuoto.

```
p-terminal(In,A,Str,[[]|Out]) :- p-terminal1(In?,A,Str,Out?).
p-terminal1([Xn|[]],A,Str,[]).
p-terminal1([Xn|In],A,[T|Str],[Xn|Out]) :- In ≠ [] | p-terminal1(In?,A,Str,Out?).
p-terminal1([Xn|In],A,[A'|Str],[[]|Out) :- In ≠ [], A≠A' | p-terminal1(In?,A,Str,Out?).
```

Il comportamento di un processo *p-nonterminal* riproduce l'esecuzione dell'operazioni di *Completer*. Un tale processo, corrispondente ad una occorrenza del simbolo non-terminale *P*, oltre allo stream di input e a quello di output ha come parametro P-bundle (il bundle associato a *P*): l'*i-esimo* stream del bundle di output e` costuito dal merge degli stream *j-esimi* tali che l'indice *j* occorre nello stream *i-esimo* del P-bundle.

```
p-nonterminal(In,[],[]).
p-nonterminal(In,[Yj|NT],[Zj|Out]):-  p-nonterminal1(In?,Yj?,Zj),
                                      p-nonterminal(In?,NT?,Out?).
```

```
p-nonterminal1(In,[],[]).
p-nonterminal1(In,[N|Y],Out) :-  sel-n(N,In?,Xn), merge(Xn?,Out1?,Out),
                                 p-nonterminal1(In?,Y?,Out1?).
```

Una notevole parte del calcolo e` realizzata da processi di *merge* . Per evitare le ripetizioni di occorrenze di uno stesso indice negli stream (in presenza, per esempio, di grammatiche ambigue) questi processi non devono semplicemente intrallacciare non-deterministicamente i due stream di input, ma devono anche eliminare le ripetizioni di uno stesso indice nello stream di output. Tale operazione e` realizzata conservando l'insieme di tutti gli elementi inseriti nello stream di output: un indice in uno stream di input viene inserito in quello di output (e nell'insieme) solo se non e` gia` presente in quest'ultimo. In seguito si faranno delle considerazioni sulla realizzazione di tale insieme e sulla sua influenza nella valutazione della complessita` dell'algoritmo.

```
merge(X,Y,Z) :- merge(X?,Y?,empty-set,Z).
merge([],[],P,[]).
merge([X|XT],Y,P,[X|Z]) :- not in(X,P)| insert(X,P,P1), merge(XT?,Y?,P1?,Z).
merge(X,[Y|YT],P,[Y|Z]) :- not in(X,P)| insert(X,P,P1), merge(X?,YT?,P1?,Z).
merge([X|XT],Y,P,Z) :- in(X,P)| merge(XT?,Y?,P?,Z).
merge(X,[Y|YT],P,Z) :- in(Y,P)| mergeX?,YT?,P?,Z).
```

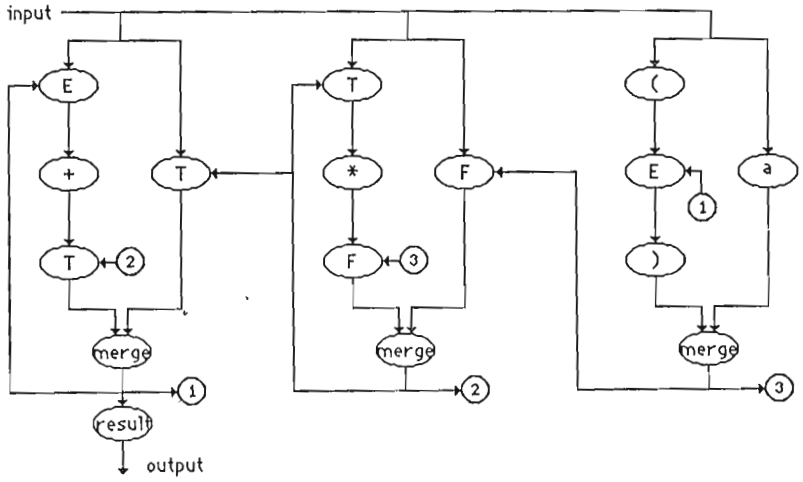
L'esecuzione del programma e` divisa in due fasi: fase di creazione della rete (CR) e fase di esecuzione dei processi (EP) che la compongono. La prima fase costruisce la rete di processi corrispondente alla grammatica rispetto a cui si vuole eseguire il riconoscimento: si costruiscono i sistemi di processi corrispondenti alle produzioni, i processi di merge per realizzare i P-bundle per ogni simbolo non-terminale *P*, e il processo *result* che ricerca nell'ultimo stream del S-bundle (dove *S* e` lo start symbol) l'indice *0* ; inoltre si inizializzano i canali di comunicazione tra i processi come bundle a N+1 elementi. La seconda fase concerne l'evoluzione dei processi che costituiscono la rete.

E` opportuno notare che la topologia della rete di processi e` indipendente dalla stringa da parsificare, mentre la costruzione della rete dipende da essa solo per quel che riguarda l'inizializzazione dei canali di comunicazione tra i processi

Nella figura e` riprodotta la topologia della rete corrispondente alla grammatica:

E <-- *E+T*	*T* <-- *T*F*	*F* <-- *(E)*
E <-- *T*	*T* <-- *F*	*F* <-- *a*

dove *E* e` lo start symbol e {*+*,*\**,*(*,*)*,*a*} e` l'insieme dei simboli terminali.



4. Correttezza

Il comportamento dei processi *p-terminal* e *p-nonterminal* garantisce la validita` dell'invariante su cui si basa l'implementazione proposta. Da questo segue che e` corretto affermare che una stringa lunga *N* appartiene al linguaggio generato dalla grammatica se e solo se nello stream *N-esimo* del S-bundle occorre l'indice *0*.

Possono sorgere pero` alcuni problemi nel caso in cui la stringa non appartenga al linguaggio; infatti, siccome in generale la rete di processi non sa riconoscere quando tutti gli item veri sono stati generati, ci puo` essere una situazione di *deadlock*.

Questa situazione si puo` evitare solo se gli stream nei bundle sono chiusi quando tutti gli item veri sono stati generati. Godono di questa caratteristica, per esempio, le reti corrispondenti a grammatiche le cui produzioni hanno come ultimo termine un simbolo terminale. Questa pero` non e` una limitazione eccessiva perche` per ogni grammatica priva di produzioni vuote esiste una equivalente grammatica di questo tipo che genera lo stesso linguaggio, come puo` essere arguito osservando che tali grammatiche sono simili alle grammatiche in forma normale di Greibach, ma con i corpi delle produzioni invertiti.

Anche altre grammatiche, che corrispondono a classi di piu` complessa caratterizzazione, sono deadlock-free, come ad esempio quella proposta nell'esempio della sezione precedente.

5. Calcolo della complessita`

Per valutare l'algoritmo di parsificazione proposto ne calcoleremo la complessita` in tempo non solo rispetto alla lunghezza della stringa da riconoscere, ma anche rispetto alle dimensioni della grammatica. Infatti, siccome lo scopo prefissatoci e` il riconoscimento di sentenze in (un sottoinsieme del) linguaggio naturale, le grammatiche rispetto cui si esegue il riconoscimento possono avere delle dimensioni superiori a quelle delle stringhe da riconoscere anche di due o tre ordini di grandezza.

Allo scopo di esaminare la complessita` dell'algoritmo, data una grammatica G=(NT,T,Prod,S), introduciamo le seguenti dimensioni:

- nt = |NT|: numero dei simboli non terminali;
- P = |Prod|: numero delle produzioni;
- N: lunghezza della stringa da parsificare;
- K=|{A<--a.b|A<--ab e` una produzione}|: numero delle doted-rule;
- ft: occorrenze dei simboli terminali nelle parti sinistre delle produzioni;
- fnt: occorrenze dei simboli non-terminali nelle parti sinistre delle produzioni;
- M=max(nt,P,ft,fnt);

E` immediato verificare che vale la seguente uguaglianza: $K=ft+fnt+P$.

La valutazione della complessita` in tempo considera come unita` di misura il singolo passo di risoluzione (riduzione) di un atomo (processo) nel goal (rete). Infatti analizzando le clausole del programma e` ragionevole pensare che il tempo di esecuzione di un passo di riduzione sia indipendente sia dalla grammatica che dalla stringa: (1) le teste di tutte le clausole sono lineari nei parametri di input, per cui, durante l'unificazione, non si eseguono uguaglianze di due termini composti, e (2) il numero di clausole del programma e` indipendente sia dalla grammatica che dalla stringa da parsificare. Inoltre con i recenti metodi di implementazione dei linguaggi logici concorrenti la scelta del processo da riscrivere e` indipendente dal numero di processi nella rete.

In base a quaste considerazioni la complessita` dell'algoritmo e` determinato dal numero di riscritture eseguite, nel caso peggiore, nei due passi *CR* e *EP*.

Prima di passare all'esame della complessita` dei singoli processi e delle fasi dell'esecuzione, si deve notare che in uno stream di un bundle ci possono essere O(N) elementi (al piu` ce n'e` N+1), essendo state eliminate le ripetizioni.

Il processo principale nella rete e` *merge* il quale ha una complessita` pari a:
T(merge) = O(N)*T(eliminazione delle ripetizioni): infatti vi sono O(N) chiamate ricorsive del

processo (una per ogni elemento negli stream su cui eseguire l'operazione di merge) per ognuna delle quali occorre eseguire il controllo se l'elemento e` gia` stato inserito nello stream di output. Per agevolare i discorsi che verranno fatti nel seguito definiamo T(eliminazione delle ripetizioni) come $O(N^{er})$: infatti tale operazione e` al piu` dipendente dal numero di elementi che possono essere presenti in uno stream.

I processi che compongono gli agenti hanno, rispettivamente, la seguente complessita`:

- *p-terminal*: un processo di questo tipo esegue una operazione di tempo costante per ogni stream del bundle di output, quindi:

  T(p-terminal)=O(N)

  inoltre in una rete vi sono O(ft) processi p-terminal;
- *p-nonterminal*: un processo di questo tipo si decompone in tanti processi p-terminal1 quanti sono gli stream nel bundle di output; siccome tali processi si decompongono al piu` in tanti processi merge quante sono gli elementi in uno stream, si ha:

  $T(\text{p-nonterminal})=O(N^2)*T(\text{merge})$

  inoltre in una rete vi sono O(fnt) processi p-nonterminal;

In una rete vi sono anche i processi che eseguono il merge per ottenere i bundle associati ad un simbolo nonterminale e il processo *result* ; quest'ultimo ha complessita` O(N), dovendo percorrere uno stream al massimo lungo N+1. Per quanto riguarda i processi di merge tra bundle essi sono tanti quante le produzioni della grammatica, cioe` O(P), e ognuno di essi origina un processo merge per ogni stream nel bundle; ne segue che la loro complessita` totale e`: T(merge-bundle)=O(P*N)*T(merge).

La complessita` della fase EP e` quindi la somma di tutte le complessita` dei processi che formano la rete: $T(EP) = O(ft)*O(N) + O(fnt)* O(N^2)*T(\text{merge}) + O(P*N)*T(\text{merge}) + O(N)$ e cioe`: $T(EP) = O(fnt*N^{3+er}+P*N^{2+er}) = O(M* N^{3+er})$.

Non avendo analizzato nei particolare la fase della creazione della rete, indicheremo solo la complessita` totale di questa fase:

$T(CR)=O(N*(K*nt))+O(nt*(P+fnt))=O(M*N)+O(M^2)=T(\text{canali})+T(\text{topologia})$.

Il primo addendo corrisponde alla inizializzazione dei canali di comunicazione con bundle di N+1 elementi, mentre il secondo corrisponde alla creazione della topologia della rete. E` opportuno fare notare che la causa della proporzionalita` al quadrato della dimensione della grammatica e` dovuto esclusivamente alla creazione delle connessioni dei processi p-nonterminal con i bundle associati ai simboli nonterminali.

Per concludere l'analisi della complessita` occorre stimare il valore di *er* : se l'insieme degli elementi gia` inseriti nell'output di un processo *merge* e` realizzato con una lista allora il tempo impiegato per controllare la presenza di un elemento e` proporzionale alla dimensione dell'insieme (cioe` *er=1* ). In questo caso la complessita` globale della parsificazione sarebbe:

$T(\text{parsing})=T(CR)+T(\text{canali})+T(\text{topologia})=O(M*N^4)+O(M^2)$.

Se l'insieme fosse rappresentato per mezzo della sua funzione caratteristica realizzata per mezzo di un array (cioe` *er=0* ). Avendo quindi a disposizione gli array la complessita` sarebbe:

$T(\text{parsing})=O(M*N^3)+O(M^2)$,

che e` quella dell'algoritmo di Earley originale.

Da questo confronto si nota come l'introduzione degli array nella programmazione logica (e in quella funzionale) e` necessaria per migliorare le prestazioni di un algoritmo. Sfortunamente normalmente questa introduzione crea dei problemi nella implementazione di questi linguaggi: per mantenere la *referential transparency*, caratteristica del paradigma dichiarativo, una struttura, e quindi anche un array, deve essere copiato ogni volta che un suo elemento e` stato copiato, aumentando nuovamente la complessita` di esecuzione. In [Hudak85] si propone un metodo per limitare le copie: un array non e` copiato, ma localmente modificato, se non vi sono piu` referenze alla sua vecchia versione. Secondo questo metodo gli array usati come funzioni caratteristiche all'interno dei processi di *merge* possono essere localmente modificati, per cui il parametro *er* risulta essere realmente *0* .

6. Compilazione della grammatica

Descrivendo la fase di costruzione della rete di processi si era fatto notare che solo la fase di inizializzazione dei bundle dipende dalla lunghezza della stringa da esaminare: infatti la topologia della rete dipende unicamente dalla grammatica. Siccome e` la costruzione di tale struttura che causa la proporzionalita` al quadrato rispetto alla dimensione della grammatica, si e` introdotta una fase di *compilazione* di una grammatica in una struttura dati, i cui costituenti individuano i processi presenti nella rete con i relativi canali di comunicazione (indicati per mezzo di variabili logiche). Tale fase compilativa ha come complessita` T(topologia), cioe` T(compilazione)=T(M^2).

In questo modo data la struttura di una grammatica compilata, la fase di esecuzione EP deve essere preceduta da una fase di instanziazione della grammatica, in cui si crea un processo per elemento della struttura e si inizializzano i canali di comunicazione; la complessita` di tale fase risulta essere: T(istanziazione)=T(processi)+T(canali)=O(M)+O(M*N). Quindi la complessita` della parsificazione di una stringa lunga N rispetto ad una grammatica di dimensione N, precedentemente compilata, e`:

T(parsificazione)=T(istanziazione)+T(EP)=O(M*N)+ O($M*N^{3+er}$)=O($M*N^{3+er}$)

cioe` lineare rispetto alla grammatica.

7. Risultati di alcune prove

Il programma descritto e` stato sviluppato e provato in flat CP, sotto il sistema Logix, sviluppato all'Universita` di Rehovot [Shapiro86] dal gruppo di Shapiro. Il programma CP e` stato trascritto in uno flat CP, eliminando, secondo la tecnica propostata da Shapiro, le guardie contenenti predicati definiti dall'utente.

Sono stati eseguiti alcuni test sull'efficienza del metodo di parsificazione descritto, utilizzando alcune grammatiche (tratte da [Tomita85]) che descrivono un sottoinsieme del linguaggio naturale.

Nella seguente tabella e` riportato il numero di riduzioni per riconoscere tre stringhe (S1, S2 e S3) rispetto a tre grammatiche (G1, G2 e G3) precedentemente compilate; le tre stringhe considerate erano lunghe rispettivamente 5, 8 e 11, mentre i parametri M, che indicano la dimesione delle tre grammatiche, erano, rispettivamente 6, 8 e 68.

reductions	S1	S2	S3
G1	1662	2117	2740
G2	2141	3003	4415
G3	12340	16622	23103

Per tutte e tre le stringhe i risultati confermano l'andamento lineare del numero di riduzioni rispetto alle dimensioni della grammtica.

La valutazione della complessita`, effettuata nei paragrafi precedenti, e` una misura delle

esecuzioni in un ambiente sequenziale, che non tiene conto del potenziale parallelismo presente sia nella visione dell'algoritmo che nella sua descrizione in CP. Nella seguente tabella e` riportato un indice del parallelismo presente nei riconoscimenti precedentemente considerati: tale indice e` la media del numero di riduzioni eseguibili in parallelo ed e` calcolato come il rapporto tra il numero di riduzioni e il numero di cicli dall'emulatore del Flat CP in Logix eseguiti durante il riconoscimento.

reds/cycle	S1	S2	S3
G1	21	18	18
G2	24	30	36
G3	37	50	69

Da questi dati si puo` notare che il parallelismo viene incrementato sia aumentando la dimensione della stringa che quella della grammatica, perche` aumentano il numero di processi CP; inoltre si ha un parallismo maggiore anche se la grammatica e` piu` ambigua, poiche` vi e` un maggior numero di item veri e, quindi, ci sono piu` indici negli stream di comunicazione.

8. Conclusioni

L'utilizzo del CP e` risultato essenziale per ottenere in brevissimo tempo e con una elevata confidenza il prototipo necessario per verificare l'idea base, mediante esempi di prove, e per valutarne approssimativamente le prestazioni al variare dei parametri d'interesse. Si ritiene che con linguaggi concorrenti convenzionali lo sforzo necessario sarebbe stato di un ordine di grandezza superiore. Particolarmente apprezzabile a questo riguardo e` la capacita` del CP (in contrasto ad es. del citato FP2) nel trattare processi dinamici (vedi fase di costruzione della rete corrispondente alla grammatica) e la facilita` con cui e` espresso il comportamento di tipo dataflow tra processi (essenzialmente statici nel nostro caso, in fase di esecuzione).

Si e` anche verificato come sulla specifica in CP sia possibile valutare le prestazioni asintotiche assumendo il passo di riduzione come unita` elementare (verificando ovviamente che esso non dipenda dai parametri valutati), comparando poi le indicazioni cosi` ottenute con indicazioni (numero totale di riduzioni, riduzioni/ciclo) fornite automaticamente dall'esecutore Logix ed ottenendo cosi` in modo estremamente rapido la conferma (o meno) dell'idea base: nel nostro caso la parallelizzazione per produzioni comunicanti a stream senza ripetizioni con complessita` lineare O(M).

Ringraziamenti

Questo lavoro e` stato parzialmente finanziato dal Progetto ESPRIT N. 415 (Parallel Architectures and Languages for Advanced Information Processing: a VLSI-directed approach).

Riferimenti

Chang Y. and K. Fu (1984), Parallel parsing algorithms and VLSI implementations fo syntactic pattern recognition, IEEE Transactions on Pattern Analysis and Machin Intelligence, vol. PAMI-6, no. 3, pp. 302-313.

Earley J. (1970), An efficient context-free parsing algorithm, Comm. of the ACM, vol. 13, no 2, pp. 94-102.

Hudak P. and **A. Bloss** (1985), Avoiding copying in Functional and Logic languages, Proc 12th annual ACM Symposium on Principles of Programming Languages, New Orlean (Louisiana), pp. 300-313.

Jorrand Ph. (1986), Functional Parallele Programmation based on term substitution, Proc AIMSA'86, Varna (Bulgaria), North-Holland Publishers.

Shapiro E. (1983), A subset of Concurrent Prolog and its interpreter, ICOT Technical Report TR-003.

Shapiro E. (1986), Logix reference manual, Univ. of Rehovot.

Tomita M. (1985), Efficient Parsing for Natural Language: a fast algorithm for practica systems, Kulwer Academic Publishers.

Un Ambiente Avanzato di Basi di Dati per Gestire Oggetti Attraverso la Programmazione Logica (*)

C. Del Gracco†, M. Dispinzeri‡, A. Mecchia‡,
P. Naggar†, C. Pizzuti‡, D. Sacca'‡

SOMMARIO

Viene descritto un ambiente avanzato di basi di dati, basato sulla combinazione della tecnologia delle basi di dati relazionali e dei principi dele programmazione logica. Tale ambiente permette di gestire oggetti complessi, tra cui sono definite diversi tipi di associazioni (in particolare, le gerarchie di istanza e di sottoclasse e la aggregazione) e di inferire le proprieta' dei singoli oggetti attraverso l'uso di tali associazioni. La programmazione logica e' opportunamente estesa per permettere di far riferimento a tutte le proprieta' degli oggetti, incluse quelle inferite. L'ambiente fa parte di un sistema per la gestione di basi di dati e di conoscenza, il cui obiettivo principale e' l'integrazione di un linguaggio per la rappresentazione degli conoscenza, orientato ad oggetti, e le basi di dati.

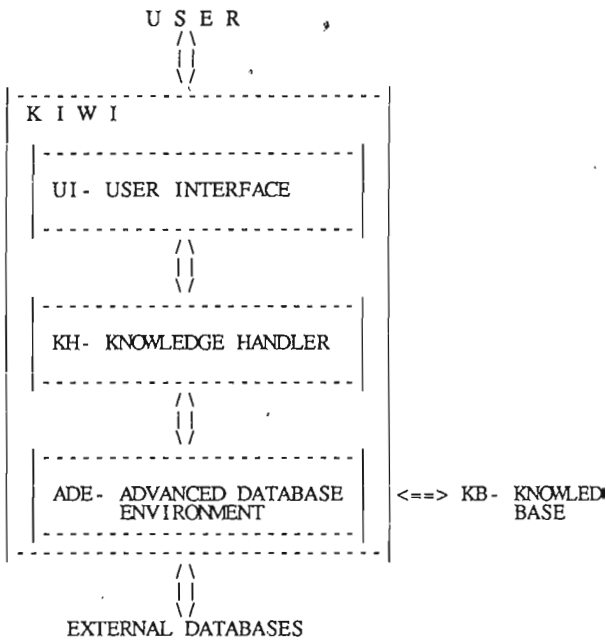
1. Introduzione

Lo scopo del progetto KIWI (*Knowledge-based User-friendly System for the Utilization of Information Bases*) e' quello di progettare e implementare un prototipo che permetta all'utente di sviluppare applicazioni basate sulla conoscenza utilizzando un certo numero di basi di dati esterne (vedere [KIWI1] e [KIWI2] per una descrizione generale del sistema KIWI). Tali applicazioni saranno scritte usando un linguaggio per la rappresentazione della conoscenza e supportate da un ambiente avanzato di basi di dati, che e' responsabile di gestire la base di conoscenza del sistema KIWI e di ritrovare e inferire fatti dalla base di conoscenza e dalle basi di dati esterne. Il sistema KIWI mette inoltre a disposizione dell'utente metodologie e strumenti tali da permettergli di individuare le informazioni volute e a ritrovarle dalla base di conoscenza e dalle basi

† ENIDATA, via Chianesi 110, Roma, Italy
‡ CRAI, c.da S. Stefano, Rende (CS), Italy
(*) *Questo lavoro e' stato svolto nell'ambito del progetto KIWI (ESPRIT No. 1117)*

di dati esterne attraverso una interfaccia user-friendly.

Il sistema KIWI consiste di tre livelli come nella seguente figura:



La *User Interface* (**UI**) assiste l'utente nell'interazione con il sistem ([UI1],[UI2]). Il *Knowledge Handler* (**KH**) implementa il linguaggio della rap zione della conoscenza OOPS, che e' basato essenzialmente sulla program orientata agli oggetti ed e' arricchito da altri potenti paradigmi quali la pro zione basata sulle regole e la programmazione logica ([KH1],[KH2]). L' *Database Environment*, (**ADE**), e' basato sulla combinazione delle tecnolo basi di dati relazionali e della programmazione logica per una impleme efficiente dei concetti di OOPS e per la gestione della *Base di Conoscen* ([ADE]). Inoltre, ADE fornisce l'interfaccia con le basi di dati esterne.

In questo articolo descriviamo come la base di conoscenza e' organizzata da ADE. Essa e' costituita da oggetti aventi diversi tipi di associazioni tra *aggregazione, isa e istanza.* Tali oggetti e le associazioni fra essi sono rappre termini di relazioni ed esistono opportuni meccanismi che, tenendo conto delle zioni, completano l'insieme delle informazioni relative ad essi. Inoltre, il li per la manipolazione e la interrogazione di tali oggetti e' basato sulla program logica che, estremamente potente per la manipolazione delle basi di dati re viene estesa per tenere conto degli oggetti ([REI], [HEN], [GAL], [PAR]

[ULL], [SAC], [ZAN]).

Il presente lavoro e' organizzato come segue: nella prima sezione viene definita la base di conoscenza in termini di oggetti e delle associazioni fra essi allo scopo di progettare gli schemi di relazione che li rappresentano. Nella seconda sezione viene descritto il linguaggio di interrogazione di tali oggetti, che e' una estensione del linguaggio logico di interrogazione delle basi di dati di tipo relazionale.

2. La Base della Conoscenza di ADE

Nel KH l'utente rappresenta la sua conoscenza (*conoscenza definita*) attraverso i concetti di OOPS, un linguaggio che raggruppa paradigmi provenienti da diverse tecniche per la rappresentazione della conoscenza. La conoscenza definita puo' essere arricchita con la *conoscenza preesistente* contenuta nelle basi di dati esterne e rappresentata secondo i modelli logici propri delle basi di dati.

ADE ha il compito di memorizzare la conoscenza definita nella KB, di manipolarla in maniera efficiente e di integrarla con quella preesistente, fornendo un semplice formalismo capace di unificare le differenti rappresentazioni. Tale formalismo e' basato sulla estensione delle basi di dati relazionali attraverso il paradigma della programmazione logica, e consiste essenzialmente nel rappresentare gli oggetti della base di conoscenza in termini di relazioni, accanto alle quali viene definito un insieme di regole che permettono di derivare nuove informazioni.

Gli oggetti che ADE memorizza nella KB sono di 3 tipi diversi:

- *oggetti diretti*

  essi consistono di un *nome* e partecipano a tre tipi di associazioni con altri oggetti. La prima associazione e' *feature* che permette di descrivere le proprieta' dell'oggetto, dove ogni proprieta' e' un riferimento ad un altro oggetto (tale associazione realizza l'astrazione di *aggregazione*, che permette di collegare un oggetto alle sue parti componenti). Le altre due associazioni, *instance* e *isa* connettono l'oggetto con altri oggetti. In particolare, l'associazione *instance* realizza l'astrazione di *classificazione* che permette di associare un oggetto ad un altro oggetto che ne costituisce il tipo (si dice che il primo e' istanza del secondo); l'associazione *isa* realizza l'astrazione di *generalizzazione* fra due oggetti, secondo cui se *A ISA B* allora ogni istanza di *A* e' anche istanza di *B*.

- *oggetti esterni*

  essi corrispondono ai record delle basi di dati esterne. Dalla definizione di ogni record si estraggono le caratteristiche di tali oggetti, cioe' il *nome* dell'oggetto che e' il nome del record e i nomi dei campi del record con i loro tipi.

- *oggetti predefiniti*

  essi sono oggetti che non hanno bisogno di essere definiti in quanto la loro definizione e' gia' contenuta in ADE. Tali oggetti sono *STRINGA*, *INTERO*, tutte le stringhe e tutti gli interi.

Il nome di un oggetto e' unico ed e' definito dall'utente nel caso degli oggetti diretti.

La *base della conoscenza*, (B_c), e' costituito da:

$$B_c = < O, ISA, INSTANCE, FEATURE, VALUE, D\_FEATURE, D\_ISA, D\_INSTANCE >$$

le cui componenti sono descritte di seguito.

(a) La componente O e' uguale a:

$$D\_O \cup B\_O \cup \{nil\}$$

cioe' e' l'insieme dei nomi di tutti gli oggetti:

- D_O e' l'insieme dei nomi degli oggetti diretti
- B_O e' l'insieme dei nomi degli oggetti predefiniti

L'introduzione dell'elemento *nil* sara' chiarrita in seguito.

(b) La componente *ISA* e' definita come:

$$ISA \subseteq D\_O \times D\_O$$

Se una coppia $(o,o') \in ISA$ allora o e' un 'sub-oggetto' di o', cioe' ogni istanza di o e' anche istanza di o'. *ISA* contiene quelle coppie (o,o') definite esplicitamente dall'utente.

(c) La componente:

$$D\_ISA \subseteq D\_O \times D\_O$$

e' la chiusura transitiva della *ISA*, cioe' se $(o,o') \in ISA$ allora $(o,o') \in D\_ISA$ e se $(o,o') \in D\_ISA$ e $(o',\hat{o}) \in D\_ISA$, allora $(o,\hat{o}) \in D\_ISA$.

(d) La componente *INSTANCE* e' definita come:

$$INSTANCE \subseteq D\_O \times D\_O$$

Se la coppia $(o,o') \in INSTANCE$ allora o e' una istanza di o'. *INSTANCE* contiene quelle coppie (o,o') definite esplicitamente dall'utente.

(e) La componente *D_INSTANCE* e' definita come un sottoinsieme:

$$D\_INSTANCE \subset O \times O$$

contenente quelle coppie (o,o') tali che:

- se $(o,o') \in INSTANCE$, allora $(o,o') \in D\_INSTANCE$

- se $(o,\hat{o}) \in D\_INSTANCE$ e $(\hat{o},o') \in D\_ISA$, allora $(o,o') \in D\_INSTANCE$
- se o' e' l'oggetto predefinito *INTERO* e o e' la denotazione di un intero, allora $(o,o') \in D\_INSTANCE$.
- se o' e' l'oggetto predefinito *STRINGA* e o e' la denotazione di una stringa, allora $(o,o') \in D\_INSTANCE$.
- se $(o,o') \in D\_INSTANCE$ allora anche $(nil,o') \in D\_INSTANCE$.

Si noti che *D_INSTANCE* contiene la chiusura della *INSTANCE* rispetto alla *D_ISA* per quanto riguarda gli oggetti diretti e descrive l'associazione *instance* esistente fra gli oggetti predefiniti. Inoltre, introduce l'elemento *nil* come istanza di qualsiasi oggetto che ha istanze.

(f) La componente *FEATURE* e' definita come:

$$FEATURE \subset D\_O \times LABEL \times LEVEL \times DOMAIN$$

dove

- *LABEL* e' un insieme di nomi tale che $LABEL \cap O = \emptyset$
- *LEVEL* e' l'insieme di numeri interi non negativi
- $DOMAIN \subseteq D\_O \cup \{STRINGA, INTERO\}$

Tale insieme contiene le associazioni *feature* definite esplicitamente dall'utente. In particolare se $(o,l,lv,o') \in FEATURE$, allora per l'oggetto o e' stata definita una feature di nome l e di livello lv, il cui tipo e' o'.

Le features a livello 0 di un oggetto o costituiscono il meccanismo con cui e' realizzata l'aggregazione. Le features a livello maggiore di 0, rappresentano uno strumento per permettere di definire l'aggregazione per classi di oggetti in relazione di istanza con o. In particolare, una feature di livello 1 per o significa che tutte le istanze di o erediteranno quella feature a livello 0. Allo stesso modo, una feature di livello 2 per o, significa che tutte le istanze di o erediteranno quella feature a livello 1, e quindi, le istanze delle istanze di o erediteranno la stessa feature a livello 0, e cosi' via per i successivi livelli.

(g) La componente *D_FEATURE* descrive, oltre alle features direttamente definite, anche quelle derivate. Formalmente, *D_FEATURE* e' definito nel seguente modo:

$$D\_FEATURE \subset D\_O \times LABEL \times LEVEL \times DOMAIN$$

dove (o,l,i,d) e' in *D_FEATURE* se:

- $(o,l,i,d) \in FEATURE$
- $(o,\hat{o}) \in ISA$, $(\hat{o},l,i,d) \in D\_FEATURE$ e $i > 0$
- $(o,\hat{o}) \in INSTANCE$, $(\hat{o},l,i+1,d) \in D\_FEATURE$ e $i \geq 0$

(h) La componente *VALUE* descrive i valori assegnati dall' utente alle features a livello 0 degli oggetti. Tenendo conto della ereditarieta' delle features, l' utente

puo' dare un valore ad una feature f di un oggetto o anche se f non e' stata esplicitamente definita per o. Inoltre, ad un oggetto possono venire associate piu' features con nome e livello uguali e domini diversi. Siano $(o,f,i,d_1),..,(o,f,i,d_n)$ tali features. Ad esse in realta' corrisponde una unica feature avente come dominio $d_1 \cap .. \cap d_n$, dal momento che una feature e' univocamente determinata dall'oggetto a cui appartiene, dal nome e dal livello. Quindi, affinche' la base di conoscenza si mantenga consistente, gli unici valori v che una siffatta feature puo' assumere sono tali che $v \in d_1 \cap .. \cap d_n$. Formalmente:

$$VALUE \subset D\_O \times LABEL \times (D\_O \cup (B\_O - \{STRINGA, INTERO\}) \cup \{nil\})$$

dove $(o,f,v) \in VALUE$ se per ogni $(o,f,0,d_i) \in D\_FEATURE$, $v \in d_1 \cap .. \cap d_n$

Si noti che nel caso in cui l'intersezione dei domini e' vuota, l'unico valore che la feature puo' assumere e' *nil*.

3. Implementazione della Base di Conoscenza

ADE si serve di una base di dati relazionale per memorizzare la base di conoscenza. Lo schema logico di tale base di dati e' disegnato in modo da contenere un insieme minimo di informazioni da cui ADE sia in grado di derivare la descrizione completa della base di conoscenza. Vedremo, infatti, che ADE e' provvisto di un insieme di regole che gli consentono tale derivazione.

Lo schema logico relazionale corrispondente alla base di conoscenza e' costituito dalle seguenti relazioni:

(1) OGGETTO(nome). Essa contiene l'elenco dei nomi di tutti gli oggetti ad eccezione di quelli appartenenti a *B_O*.

(2) FEATURE(nome_oggetto,nome_feature,livello,dominio). Essa contiene le descrizioni delle features definite da utente per i diversi oggetti.

(3) INSTANCE(nome_istanza,nome_oggetto). Essa contiene le dichiarazioni fatte dall'utente della esistenza di una gerarchia di classificazione fra due oggetti.

(4) ISA(nome_sub,nome_sup). Essa contiene le dichiarazioni fatte dall'utente della esistenza di una gerarchia di generalizzazione fra due oggetti.

(5) VALUE(nome_oggetto,nome_feature,valore). Essa contiene i valori assegnati dall'utente alle features di livello 0 definite esplicitamente per un oggetto.

(6) una relazione *nome_oggetto*$(f_1,..f_n)$ per ogni oggetto diretto di nome *nome_oggetto* che ha features $(f_1,..f_n)$ a livello 1 o definite o ereditate attraverso la associazione *instance*. Lo schema di tale relazione e' determinato da ADE sulla base delle features degli oggetti ed e' costituito da un attributo per ogni feature a livello 1 piu' un ulteriore attributo, di nome *self*, il cui dominio e' l'insieme dei nomi delle istanze dell'oggetto. Ogni tupla di questa relazione corrisponde ad una istanza dell'oggetto associato alla relazione.

4. Il Linguaggio di Interrogazione di ADE

Il linguaggio di interrogazione della base di conoscenza, e' basato sulla programmazione logica e costituisce una estensione del linguaggio logico di interrogazione delle basi di dati di tipo relazionale, introdotto in [ULL].

Il nostro linguaggio consiste di:

- un predicato associato ad ognuna delle relazioni dello schema logico che implementa la base di conoscenza. Ognuno di tali predicati ha il nome della relazione associata e un numero di argomenti pari al numero di attributi delle corrispondenti relazioni.
- i predicati binari D_ISA e D_INSTANCE e il predicato quaternario D_FEATURE associati rispettivamente alle componenti *D_ISA*, *D_INSTANCE* e *D_FEATURE* della base di conoscenza.
- predicati associati agli oggetti con un numero di argomenti pari al numero di features ereditate attraverso le associazioni *isa* e *instance*.
- predicati di confronto ($=$, $<$, $\leq$, $>$, $\geq$, $\neq$).
- un predicato variabile *S.nome variabile*. Tale predicato ha un numero variabile di argomenti. Esso puo' venire associato da ADE dinamicamente a piu' oggetti, per cui le n-ple di valori che lo rendono vero, sono ricavate al momento di tali associazioni.
- predicati temporanei. Essi possono essere introdotti per facilitare la risoluzione di una query e sono definiti in termini di altri predicati. Il numero di argomenti e' quindi definito dall'utente. L'insieme delle n-ple di valori che li rendono veri, sono ricavate al momento della risoluzione di una query.

Esistono due modi di specificare gli argomenti dei suddetti predicati: il primo, detto *posizionale*, consiste nello sfruttare l'eventuale ordinamento degli argomenti nel predicato, il secondo, detto *nominale*, consiste invece nello sfruttare il nome degli argomenti. Questa distinzione nasce dalla constatazione che esistono dei predicati la cui struttura e' ben nota per cui e' possibile stabilire un ordinamento degli argomenti. Per altri invece, in particolare per i predicati associati agli oggetti diretti, parlare di ordinamento degli argomenti ha poco senso, in quanto il processo di *ereditarieta'* determina dinamicamente la struttura del predicato. Analogamente, poiche' la struttura del predicato variabile e' nota solamente al momento della risoluzione della query, l'unico modo per specificarne gli argomenti e' quello *nominale*.

Una query nel nostro linguaggio e' espressa mediante una coppia:

$$<LP,C>$$

dove *LP* e' un insieme, eventualmente vuoto, di clausole logiche e *C* e' una congiunzione di predicati rappresentante un goal da risolvere attraverso le clausole in *LP* e i fatti della base di conoscenza. Una generica clausola in *LP* e' costituita da una coppia *<head,body>* dove *head* e' un predicato temporaneo e *body* e' una congiunzione di

predicati. Ogni predicato *P* in *C* che non e' *head* di regole in *LP*, deve allora essere uno dei predicati definiti sopra.

Per ogni predicato a cui non e' associata una relazione, viene introdotta da ADE una regola in *LP*, in modo che sia chiaro il procedimento di deduzione. In particolare, per i predicati *D_ISA*, *D_INSTANCE* e *D_FEATURE* le regole sono fissate a priori e sono le seguenti:

D_ISA(X,Y):–ISA(X,Y).

D_ISA(X,Y):–D_ISA(X,Z),D_ISA(Z,Y).

D_INSTANCE(X,Y):–INSTANCE(X,Y).

D_INSTANCE(X,Y):–D_INSTANCE(X,Z),D_ISA(Z,Y).

D_FEATURE(X,Y,Z,T):–FEATURE(X,Y,Z,T).

D_FEATURE(X,Y,Z,T):–D_FEATURE(W,Y,Z+1,T),Z≥0,D_INSTANCE(X,W).

D_FEATURE(X,Y,Z,T):–D_FEATURE(W,Y,Z,T),Z>0,D_ISA(X,W).

Per i predicati associati agli oggetti, le regole sono determinate da ADE tenendo conto delle associazioni *isa*. Infatti, supponiamo per esempio che un oggetto *P* con features f e g di livello 1, sia in relazione *isa* con un oggetto *Q* avente features h ed i di livello 1. Per *P* e *Q* saranno definite due relazioni: P(self, f, g) e Q(self, h, i) mentre, poiche' *P isa Q*, i predicati associati ad essi sono: *P(self,f,g,h,i)* e *Q(self,h,i)*. Per il predicato *P(self,f,g,h,i)* ADE aggiunge ad *LP* la regola:

P(x:self,y:f,t:g,z:h,w:i):-P(x:self,y:f,t:g),Q(x:self,z:h,w:i).

Diamo ora una serie di esempi di uso del linguaggio di interrogazione.

Esempio 1.

Supponiamo di avere le seguenti relazioni:

OGGETTO
nome_oggetto
Persona
Letteratura
Libro
Giornale

INSTANCE	
nome_istanza	nome_oggetto
Libro	Letteratura
Giornale	Letteratura

FEATURE			
nome_oggetto	label	livello	dominio
Persona	nome	1	STRINGA
Letteratura	titolo	2	STRINGA
Letteratura	editore	2	STRINGA
Libro	autore	1	Persona

In aggiunta, si supponga che la relazione OGGETTO contenga ulteriori oggetti che non sono stati listati per brevita'. Tali oggetti sono istanze degli oggetti citati. Per brevita' sono anche omesse le relazioni associate agli oggetti Persona, Letteratura e Libro.

Volendo conoscere i titoli e gli editori di tutto cio' che e' presente in letteratura (libri e giornali), la query dovra' essere formulata nel modo seguente:

<∅ ,{INSTANCE(x,'Letteratura'),S. x(y:titolo,z:editore)}>.

o equivalentemente:

<∅ ,{Letteratura(x:self),S. x(y:titolo,z:editore)}>.

il predicato variabile S. x viene associato a tutti gli oggetti istanze dell'oggetto Letteratura. Se invece si vuole conoscere l'autore ed il titolo di cio' che e' presente in letteratura, allora, nella query

<∅ ,{INSTANCE(x,'Letteratura'),S. x(y:titolo,z:autore)}>.

il predicato variabile viene associato solo all'oggetto Libro. Se inoltre si vuole conoscere il nome dell'autore si potra' formulare la query nel modo seguente :

<∅ ,{INSTANCE(x,'Letteratura'),S. x(y:titolo,z:autore),S. z(t:nome)}>.

Esempio 2.

Si supponga che esistano le seguenti relazioni:

Persona		
self	nome	eta
Rossi	Paolo Rossi	32
Bianchi	Marco Bianchi	26

Impiegato			
self	salario	societa	manager
Rossi	1600000	IBM	Bianchi
Bianchi	1200000	IBM	nil

associate agli oggetti *Persona* e *Impiegato*, mentre le relazioni *FEATURE* e *OGGETTO* sono state omesse per ragioni di brevita`. La query:

```
<{ Vip(x) :- Impiegato(x:nome,y:salario),y > 3.000.000
     Vip(x) :- Impiegato(x:nome,y:salario,z:manager),
                Impiegato(z:self,y':salario),y > y' },
          {Impiegato(x:nome,'IBM':societa),Vip(x)}>.
```

ritrova tutti gli impiegati della societa' IBM che hanno un salario maggiore di 3.000.000 oppure maggiore del salario del loro manager. Si noti l'utilizzo del predicato temporaneo Vip la cui vita e' limitata esclusivamente alla risoluzione della query.

5. Conclusioni

In questo lavoro abbiamo discusso l'organizzazione di una base di conoscenza costituita da oggetti e dalle associazioni fra essi. Si e' visto come tali oggetti possano essere rappresentati da relazioni e interrogati da un linguaggio che costituisce una estensione del linguaggio logico di interrogazione delle basi di dati relazionali. Sono in corso di definizione la formalizzazione di tale linguaggio e la sua estensione in modo che includa anche comandi di aggiornamento degli oggetti. Naturalmente, una tale estensione comporta l'introduzione di vincoli di consistenza che devono essere rispettati dalla base di conoscenza.

Tale linguaggio costituira' l'interfaccia fra KH e ADE. Sara' poi compito di ADE trasformare le richieste espresse dal KH in richieste espresse mediante un insieme di clausole di Horn generalizzate che possano essere facilmente interfacciate alle relazioni della base di conoscenza. Le richieste trasformate saranno quindi compilate secondo metodi il cui scopo e' quello di ottenere una implementazione efficiente delle query.

6. Bibliografia

[ADE] M. Dispinzeri, A. Mecchia, G. Rossi, D. Sacca`, N. Spyratos, "Definition and Design of the Advanced Database Interface", Report B5-C5, ESPRIT Project

641, November 1985.

[GAL] H. Gallaire, J. Minker, J.M. Nicolas, "Logic and Data Bases. A Deductive Approach", *Computing Surveys* , Vol. 16, n.2, (1984)

[HEN] L. Henschen, S. Naqvi, "On Compiling Queries in Recursive First-Order Data Bases", *JACM 31*, 1, (1984)

[JAR] M. Jarke, J. Clifford, Y. Vassiliou, "An Optimizing Prolog Front-end to a Relational Query", *Proc. ACM SIGMOD Conference on Management of Data* , (1984).

[KIWI1] D. Sacca, D. Vermeir, A. D'Atri, A. Liso, G. S. Pedersen, J.J. Snijders, N. Spyratos, "Description of the Overall Architecture of the KIWI System", *ESPRIT'85: Status Report of Continuing Work, The Commission of the European Communities (Eds.)*, Elsevier Science Publishers B.V. (North-Holland), 1986, 685-700.

[KIWI2] A. D'Atri, P. Naggar, G. S. Pedersen, D. Sacca, J.J. Snijders, N. Spyratos, D. Vermeir, "Preliminary Description of the KIWI Architecture", Report A1, ESPRIT Project 1117, June 1986.

[KH1] D. Vermeir (editor), "OOPS: a Knowledge Representation Language", Report B4-C4, ESPRIT Project 641, November 1985.

[KH2] D. Vermeir (editor), "Requirements Documentation of the Knowledge Handler (Main Features)', Report B3, ESPRIT Project 1117, October 1986.

[PAR] D.S. Parker, et al., "Logic Programming and Databases", *Proc. First International Workshop on Expert Database Systems*, Kiawah Island, S.C., (Oct. 1985)

[REI] R. Reiter, "Deductive Question Answering in Relational Data Base", in *Logic and Data Base*, H. Gallaire and J. Minker (eds.), Plenum Press, New York, (1978)

[SAC] D. Sacca, C. Zaniolo, "The generalized counting method for recursive logic queries", *Proc. 1st International Conf. on Database Theory*, Rome, 1986.

[UI1] A. D'Atri, M. Lenzerini, P. Naggar, A. Paoluzzi, A. Pelaggi, A. Marchetti-Spaccamela, "Design of the User Interface to the KIWI System", Report B2-C2, ESPRIT Project 641, November 1985.

[UI2] A. D'Atri, A. Paoluzzi, R. Bettacchi, A. Leone, "Box Management System: A graphical User Interface to Knowledge and Data Bases", Report D2/1, ESPRIT Project 1117, October 1986.

[ULL] J.D. Ullman, "Implementation of Logical Query Languages for Databases", *TODS 10*, 3, 1985, pp. 289-321.

[ZAN] C. Zaniolo, "The Representation and Deductive Retrieval of Complex Objects", *Proc. VLDB Conf.*, Stockholm, (Aug. 1985).

LA PROGRAMMAZIONE LOGICA IN AMBIENTI DI SVILUPPO SOFTWARE: UN DATA BASE DI PROGETTO LOGICO

Patrizia ASIRELLI*, Fabio GIANNINI+, Elena GRIFONI+, Paola INVERARDI*

+ ESTEC, Noordwijk, Olanda
* I.E.I.-Istituto di Elaborazione dell'Informazione -CNR, Pisa

RIASSUNTO

Questo articolo e' un estratto dal lavoro [Gia 86]. L'obbiettivo centrale di tale lavoro era quello di mostrare come l'utilizzo della Programmazione Logica potesse fornire un approccio positivo ed innovativo nel campo degli Ambienti di Sviluppo Software in particolare proponendo un Data Base Logico come Data Base di Progetto.
Il motivo per cui la ricerca si e' indirizzata verso l'applicazione della Logica nel campo dei Data Base di Progetto (DBP) [Ga1 84, Ga2 78, Obe 85, CSE 83], nasce da una parte dalla peculiarita' delle informazioni necessarie per supportare il ciclo di vita del software, che richiede al DBP funzionalita' specifiche che lo distinguono dai tradizionali Data Base, e dall'altra da un insieme di caratteristiche tipiche della Logica che risultano particolarmente adatti ai DBP.
La ricerca si e' sviluppata nel seguente modo : sono stati analizzati e definiti con precisione i requisiti piu' significativi di un DBP; in base a tali requisiti, una volta descritto un semplice esempio di Ambiente di Programmazione per un linguaggio Ada-like, e' stato disegnato il relativo DBP, secondo il modello Entity/Relationship [Che 76].
Tale disegno di DBP e' stato poi implementato sia con OMS, il Sistema di Gestione per Basi di Dati di Progetto di PCTE [ESP 85], sia con EDBLOG, un sistema di gestione per Basi di Dati Logiche [Asi 85, Mau 85].
Il confronto delle due implementazioni con i requisiti e quello diretto fra esse ha confermato i vantaggi dell'approccio logico rispetto a quello tradizionale (se si tralasciano problemi di efficienza).
Inoltre la realizzazione del DBP mediante la Logica ha permesso di estendere ed adattare lo stesso modello Entity/Relationship.
In questo estratto si ánalizzano i motivi per cui si e' realizzato il DBP mediante un Data Base Logico, tralasciando sia l'analisi del DBP tradizionale (OMS), sia gli aspetti di implementazione del modello logico.
In particolare si esaminera' come la base logica permetta al DBP di rispondere in modo completo a tre requisiti essenziali e particolarmente complessi (indicati con "Data Model", "Configurazione", "Storia") richiesti ai database di progetto. Questi requisiti saranno ampiamente discussi.
Infine nell'Appendice si mostreranno le definizioni complete della Configurazione e della Storia e come esse si applicano su un programma Ada completo.

1. PERCHE' REALIZZARE UN DBP LOGICO

Tradizionalmente il Sistema di Gestione per DBP presente in un Ambiente di Programmazione fornisce un Livello di Gestione della Base di Dati (LGDB).
Al di sopra di questo livello, facendo riferimento alla classica stratificazione virtuale di un ambiente di programmazione e.g. [US1 80], troviamo i vari "tool" che utilizzano quello che viene fornito dal LGDB, ovvero primitive per l'accesso/manipolazione dei dati, o meglio veri e propri linguaggi per l'interrogazione e la modifica del Database.
Qualsiasi funzione complessa che interessi il DBP viene eseguita all'interno del tool per essere utilizzata dal tool che la esegue oppure per realizzare un ulteriore strato virtuale di livello superiore

Questo e' quanto accade ad esempio in PCTE, dove il gestore della base di dati di progetto e' OMS che rende disponibili una serie di primitive di interfacciamento con i tool superiori, senza tuttavia fornire alcuna funzionalita' di complessita' elevata. In altri termini, OMS fornisce una serie di primitive di basso livello utilizzabili per realizzare svariate politiche di gestione. Tale scelta e' ovviamente legata al ruolo di OMS, che si propone come un kernel di ambiente di programmazione e come tale deve, quindi, supportare la maggiore flessibilita' possibile nella definizione degli strati superiori di software che un utente puo' voler realizzare. Tale flessibilita' viene pagata in termini della primitivita' delle funzionalita' offerte. Infatti, anche la maggiore efficienza della base di dati puo' venir compromessa, quando si consideri l'insieme del software necessario per poter offrire livelli di funzionalita' paragonabili a quelli offerti da una base di dati logica, come per es. la possibilita' di definire vincoli di integrita'.
Realizzare un DBP con un Database Logico permette di definire un Livello funzionale di Base, su cui costruire le applicazioni, ben piu' elevato rispetto a quello fornito dai Data Base tradizionali, senza tuttavia diminuire la flessibilita' del sistema.
Si tratta in pratica di elevare il LGDB sino a coprire una serie di applicazioni altrimenti demandate al Livello Tool. Per esemplificare, si puo' fare riferimento alla stratificazione dell'ambiente introdotta da Stoneman [US1 80], che distingue i tre livelli KAPSE, MAPSE, APSE: realizzare un DBP logico permette di fondere i livelli KAPSE e MAPSE senza perdita di generalita'.
L'innalzamento del LGDB viene ottenuto essenzialmente attraverso lo sfruttamento di due caratteristiche dei DB Logici : Deduzione e Trattamento dei Vincoli di Integrita'.
La possibilita' di definire Regole Deduttive amplia la Potenza Espressiva del Database; consente di estrarre nuove informazioni che non erano state esplicitamente introdotte, semplificando inoltre la definizione e la comprensione di alcune relazioni.
Le Regole Deduttive vanno oltre i semplici fatti presenti nel DB, definendo l'informazione ad un livello superiore; l'utilizzo di Regole Deduttive permette in definitiva una diversa rappresentazione dell'informazione separando quelli che sono i fatti esplicitamente presenti (e necessari) nel DB, sotto forma di asserzioni, dai fatti che invece non hanno bisogno di essere presenti, in quanto deducibili tramite regole.
Il trattamento dei Vincoli di Integrita' in un DB Logico, a prescindere da problemi di efficienza, sicuramente riducibili e comunque intrinseci alla complessita' dell'operazione, risulta facilmente integrabile nel DB stesso. Questo, grazie alla possibilita' di sfruttare i medesimi meccanismi di dimostrazione utilizzati per le Query al DB. Inoltre tali vincoli vengono espressi semplicemente come formule del Primo Ordine.
Un'altra caratteristica interessante dei DB Logici e' quella dell'uniformita' dei linguaggi necessari nelle varie fasi. Infatti si ha sempre a che fare con formule del Primo Ordine, sia per quanto riguarda l'inserzione dei fatti, sia per la definizione delle Regole, sia come linguaggio di interrogazione che come linguaggio di definizione.

2 - IL DBP LOGICO E I REQUISITI

Per poter confrontare l'approccio proposto con l'approccio classico sono stati presi come riferimento alcuni requisiti che, come risulta in letteratura [Obe 85], [CSE 83], sono da considerare caratterizzanti di cosa si debba intendere per database di progetto. In questo paragrafo descriveremo tre dei requisiti considerati, dal nostro punto di vista i piu' significativi, e vedremo come il DBP Logico risponda ad essi.

2.1 - DATA MODEL

La peculiarita' delle informazioni che devono essere trattate da un data base di progetto pone anche dei problemi sulla adeguatezza dei piu' noti modelli

semantici sui quali basare un DBP. La problematica e' ben nota in letteratura,e.g. "non esiste un Data model adatto a rappresentare gli specifici contenuti e le relazioni tra i componenti di un DBP (si ritiene comunque che il modello Relazionale e quello Entity/Relationship siano i migliori tra quelli esistenti, pur se non sufficienti)" [Obe 85]; "infatti, poiche' in un Ambiente di Sviluppo Software esiste la necessita' di esprimere relazioni piu' complesse di quanto accada in un ambiente tradizionale, in un DBP esse sono numerose e fortemente interallaciate [CSE 83]".
Tale complessita' richiede dunque un maggiore potere espressivo da parte del Data Model che deve rappresentarle, senza per altro forzare politiche che potrebbero rivelarsi inadeguate.
In generale, definire un DB mediante la Logica, non significa definire anche un nuovo tipo di Data Model. Percio', quello che ci siamo proposti di fare e' stato di realizzare il modello Entity/Relationship su un DB Logico, mostrare come il modello ottenuto possa essere considerato una estensione del Modello E/R stesso e come esso sia aperto ai possibili sviluppi verso cui si sta indirizzando la ricerca nel campo dei Data Model.
Realizzare il Modello E/R su un DB Logico, significa fornire al Data Model una base formale che e' quella della Logica del Primo Ordine. In questo modo non si introduce un nuovo Data model. Tuttavia, definire il Modello E/R con un sistema matematico, estende il Data Model stesso con gli strumenti e le tecniche matematiche associati; si forniscono una notazione e delle definizioni formali per i concetti del Data Model; si forniscono strumenti per analizzare proprieta' (essenzialmente strutturali) delle applicazioni e per risolvere alcuni problemi teorici, come la dimostrazione della consistenza del DB. Inoltre un Data Model basato sulla Logica del primo ordine, si pone in una ottica nuova rispetto ai Data Model tradizionali : in questi ultimi, infatti, i tipi-oggetti sono rappresentati dalle loro istanze (questo discorso vale non solo per i Modelli Classici, ma anche per l'E/R, in cui esistono gli Entity Set, ma di fatto il DB e' costruito sulle istanze di tali insiemi) e le modifiche sono realizzate esclusivamente aggiungendo o estraendo fatti. Con un Data Model basato sulla Logica, invece, gli Oggetti sono rappresentati come "Termini", e gli Attributi, Relazioni e Vincoli come "Formule ben formate".
Vi e' dunque un DB di fatti su cui vengono valutate le fbf per rispondere alle Query.
Le modifiche all'applicazione si realizzeranno quindi aggiungendo fbf, ma non alterando necessariamente il DB dei fatti grazie alle Regole Deduttive [Bro 84].
Infine, un Data Model Logico e' aperto di fronte ai possibili sviluppi della ricerca nel campo dei Data Model: concetti come *Relativismo Semantico*, *Astrazione*, *Deduzione*, sono particolarmente adattabili ai DB Logici e possono quindi essere sviluppati in modo semplice e naturale.
Un altro argomento di ricerca nel campo dei Data Model e' quello dell'*Implementabilita'*. In [Bro 84] si fa notare come soltanto i Modelli Classici (Relazionale, Reticolare e Gerarchico) siano stati implementati, mentre quasi niente e' stato fatto con quelli Semantici (fra cui il Modello E/R). Nel lavoro da cui abbiamo estratto questo articolo si e' presentata proprio l'implementazione del Modello E/R con base Logica.
Concludendo, uno dei risultati piu' significativi che si ottengono fornendo una base Logica al Data Model, e' quello di *realizzare* un modello formalmente definito e tuttavia non vincolante rispetto ai nuovi sviluppi del campo.

2.2 - CONFIGURAZIONE

Una delle funzionalita' piu' significative che un DBP deve offrire e' quella di rendere disponibili meccanismi in supporto alla configurazione. Come configurazione intendiamo un insieme di partizioni di un progetto, in opportune versioni, che formano un sistema completo : nel caso specifico dello sviluppo di un progetto software in un linguaggio Ada-like, una configurazione e' un insieme di "moduli compilati" in una opportuna versione-opzione (versioni diverse di uno stesso modulo significano realizzazioni diverse, con un diverso algoritmo, della stessa funzione; le opzioni sono intese come opzioni di compilazione).
Questo insieme di moduli compilati forma un programma completo. Scendendo piu' in dettaglio, una configurazione sara' composta da :

1) un modulo compilato "main";
2) un insieme di moduli compilati legati al "main" in quanto sue "with" (unita' importate) o "subunit" (unita' ottenuto sviluppando il programma top-down);
3) ricorsivamente, tutti i moduli compilati legati ai moduli finora trovati perche' loro "with" o "subunit".
Come si vede dalla definizione, la complessita' della raccolta dei moduli che formano una configurazione e' notevole, ed e' quindi fondamentale l'aiuto di meccanismi automatici . Tali meccanismi dovrebbero svincolare l'utente dal tenere traccia di tutte le relazioni all'interno di un programma ed assicurare una corretta configurazione. Tuttavia dovrebbe anche essere permessa all'utente la definizione di schemi alternativi di configurazione secondo criteri legati ai particolari requisiti del momento [Obe 85].
Nel nostro database logico si e' automatizzata la costruzione e la verifica delle configurazioni mantenendo un'ampia flessibilita' nella loro definizione.
Utilizzando EDBLOG (vedi appendice per una breve descrizione del sistema), per implementare la configurazione si e' fatto uso delle *transazioni* (insieme di operazioni elementari eseguite atomicamente se vengono verificate pre e post condizioni) e delle *regole deduttive;* con l'utilizzo ricorsivo delle prime si costruisce una relazione, conftype(main, X, caratteristica), che lega ad un "main" tutte le Compilation Unit X che hanno una caratteristica comune. Questa caratteristica e' definita tramite fatti o regole deduttive.
Poiche' le Compilation Unit possono essere in piu' versioni-opzioni, a partire da uno stesso "main" si possono costruire configurazioni diverse.
Una volta scelto il tipo (caratteristica) di configurazione che si vuole realizzare, occorrera' ricercare tra le varie versioni-opzioni quelle che rispondono alla caratteristica richiesta dalla Configurazione.
Per questo si associa ad ogni Compilation Unit un predicato, "type", col quale si lega la caratteristica al nome della Compilation Unit.
Ad esempio, una CU puo' avere tipo "Last" (indica che si tratta dell'ultima versione-opzione del modulo), oppure "tested" (indica che la CU e' stata testata).
Vediamo un esempio :
supponiamo di avere un modulo A, nelle versioni-opzioni A.1.1., A.1.2., A.2.1. e supponiamo che sia :

```
type(cuc(a,1,1),bad).
type(cuc(a,1,2),tested).
type(cuc(a,2,1),last).
type(cuc(a,2,1),quick).
```

(e' interessante notare che una stessa Compilation Unit Compilata [cuc] , come A.2.1., puo' avere piu' di un tipo).
Nel configurare un programma contenente il modulo A dovremo stabilire un criterio di scelta tra le diverse versioni del modulo.
Ad esempio, potremmo volere una configurazione composta da tutti moduli testati, nel qual caso sara' A.1.2. a farne parte, mentre se volessimo una configurazione composta dalle ultime versioni di ogni modulo la nostra scelta cadrebbe su A.2.1. .
Il predicato "type" puo' essere definito oltre che da fatti, anche da regole deduttive.

```
type(X, not_bad) <- type(X, tested).
type(X, not_bad) <- type(X,last).
```

In questo caso una configurazione con caratteristica "not_bad", sarebbe composta da Compilation Unit "tested" o, in mancanza di queste, dalle "last" (a causa dell'ordine di valutazione delle clausole in Prolog).
Di seguito la prima clausola della configurazione :

```
1) config(A,conf) <- type(A,conf), compiled(Z,A), main(Z)
                                    |forall(deleteconf(A,X,conf)),
                                    config1(A,Z,conf).
```

L'implementazione completa della configurazione e la sua interazione con un esempio di programma Ada, sono mostrate nell'Appendice.

2.3 - STORIA

Un DBP deve fornire meccanismi efficienti per la collezione, il mantenimento

del mantenimento dei dati sono dipendenti dalla metodologia e non dovrebbero essere limitate dal Data Base.
Dovrebbe essere possibile sia automatizzare che lasciare al controllo dell'utente tali meccanismi; infatti e' desiderabile la collezione automatica della Storia per garantire la completezza dell'informazione raccolta, tuttavia deve essere possibile interrompere tale raccolta per evitare la saturazione durante le attivita' di sviluppo [Obe 85].
Interpretiamo il concetto di Storia secondo la definizione di [US1 80] :
"la Storia memorizza il modo in cui un oggetto e' stato prodotto e contiene tutte le informazioni necessarie e rilevanti per la sua produzione".
Nella nostra realizzazione al fine di mantenere la definizione data e di soddisfare i requisiti richiesti (in particolare automatizzazione e, contemporaneamente, controllo da parte dell'utente) si sono definiti due tipi di Storia :
- Storia Generativa
- Storia Documentazione

2.3.1 - STORIA GENERATIVA

Questo tipo di storia e' particolarmente legato alla struttura del programma nel linguaggio di programmazione dell'ambiente (nel nostro caso Ada).
Le caratteristiche essenziali sono :
- La Storia Generativa e' l'insieme di informazioni <u>necessarie</u> alla creazione di un oggetto (le "with clause", il "padre", la "library", il "sorgente");
- le informazioni relative alla generazione devono essere mantenute per tutta la vita di un oggetto;
- gli oggetti riferiti nella Storia generativa di un oggetto non possono essere cancellati.

Per tali caratteristiche si e' scelto che la Storia generativa venga raccolta e protetta automaticamente.
Essa viene espressa mediante regole deduttive, ad esempio :

history(X,Y) <- compiled(Z,X), compiled(W,Y), libsec(Z,W).

(la Storia gen. dell'oggetto X, che e' un compilato, e fatta anche dalla sua Library Y, sempre che l'oggetto X sia una Secondary (e questo viene controllato dalla relazione "libsec").
compiled(Z,X), compiled(W,Y) sono necessarie perche' la relazione "libsec" e' stata data sui sorgenti.
La Storia gen. e' fatta dai compilati, infatti la storia di un sorgente si limita alla fase di Editing, e quindi non e' di grande rilevanza). Questa definizione di Storia si ferma al primo passo (ad esempio da' le "with", ma non le "with" delle "with").Sarebbe possibile definirla come transazione, ricorsivamente, e dare quindi tutto l'albero-storia sopra l'oggetto.
Da questa definizione di Storia gen. si possono ad esempio derivare :
1) tutti gli oggetti che costituiscono la Storia gen. della Compilation Unit Compiled cuc(a,1,2) ; sara' infatti sufficiente fare una Query del tipo :

history(cuc(a,1,2), Y) <--

2) tutti gli oggetti di cui la Compilation Unit Compiled cuc(a,1,2) e' Storia gen. ; questa volta la Query sara' del tipo :

history(X,cuc(a,1,2)) <--

Vediamo adesso alcuni esempi di utilizzo della Storia gen. :
- puo' essere utilizzata, in modo trasparente all'utente, per l'operazione di ricompilazione (ricompilare un modulo significa anche ricompilare, o almeno rendere obsoleti, tutti quei moduli di cui esso e' Storia gen.);
- puo' essere utilizzata nell'operazione di cancellazione :
un oggetto puo' essere cancellato solo se non appartiene alla Storia gen. di alcun altro oggetto.
- puo' essere utilizzata per dare un vincolo globale sulle date di creazione degli oggetti, secondo il principio che, se un oggetto e' Storia gen. di un altro oggetto, esso deve essere stato creato per primo:

history(X,Y) <-- date(X) > date(Y).

L'implementazione completa della Storia gen. e' descritta nell'Appendice.

2.3.2 - STORIA DOCUMENTAZIONE

E' l'insieme di informazioni <u>rilevanti</u>, ma non necessarie nella produzione di

un oggetto.
Ad esempio :
- la data di creazione di un oggetto,
- il tipo di compilatore usato,
- informazioni sulla creazione nel tempo delle Configurazioni,
- la documentazione.
Si e' scelto di lasciare all'utente la definizione di questo tipo di Stor
considerando che la cancellazione o l'assenza di informazioni di questo gen
non compromette le funzionalita' del sistema.

3 - CONCLUSIONI

Applicare i concetti della Programmazione Logica ai DBP ha significato an
formalizzare un'area che puo' essere considerata intermedia tra Linguaggio
Programmazione e Ambiente di Sviluppo; proprio per questo, generalmente, e
non e' formalizzata all'interno del linguaggio anche se fa parte della
descrizione informale.
Nel nostro lavoro, infatti, sono stati definiti con la Logica del Pri
Ordine i concetti relativi alla strutturazione di un programma A
(compilation unit, subunit, with clause, program library, ...) e alle reg
di compilazione e ricompilazione.
Inoltre sono stati formalizzati concetti tipici degli Ambienti di Svilup
come quello della configurazione, del linkaggio e della storia.
Tali definizioni sono direttamente eseguibili.
Si noti infine la semplicita' dell'implementazione (la configurazione,
esempio, si limita a 7 clausole).

RIFERIMENTI

[Asi 85] Asirelli, P., De Santis, M., Martelli, M.,Integrity Constraints in
Logic Data Bases, *Journal of Logic Programming*, Vol. 2, no. 3, Oct. 1985.

[Bro 84] Brodie, M.L., On the Development of Data Models, *Conceptual Modelling*, ed. Springer Verlag, New York, 1984.

[Che 76] Chen, P.P., The Entity-Relationship Model. Toward a Unified View o
Data, *ACM Transaction on Database Systems*, Vol. 1, March 1976.

[CSE 83] CSELT, Imperial College, Imperial Software Technology, Logics SA/N
"Requirements for Software Engineering", *Final Report*, June 1983.

[ESP 85] ESPRIT , PCTE : A Basis for a Portable Common Tool Environment,
Functional Specification, *Third Edition* 1985.

[Gal 84] Gallaire, H., Minker, J.,Nicolas, J.M., Logic and Database : A
Deductive Approach, *Computing Surveys*, Vol. 16, no. 2, June 1984.

[Ga2 78] Gallaire, H.,Minker, J., Nicolas, J.M., An Overwiev and Introduct
to Logic and Databases,*Logic and Databases* , ed. Gallaire Minker 1978.

[Gia 86] Giannini, F., Grifoni, E., Programmazione Logica in Ambiente di
Sviluppo Software : Data Base Logici come Data Base di Progetto, *Tesi di laurea, Corso di Laurea in Scienza dell'Informazione*, Pisa 1986.

[Mau 85] Mauro, F., Basi di Dati Logiche : un Approccio al Trattamento del
Transazioni, *Tesi di Laurea,Corso di Laurea in Scienza dell'Informazione* ,
Pisa 1984/85.

[Obe 85] Oberndorf, P.A., Penedo, M.H., Future Ada Environment Workshop,
Summary of Project Database Working Group Discussion, *ACM Ada Letters* 19

[US1 80] US Dod, Requirements for Ada Programming Support Environments, Stoneman, February 1980.

APPENDICE

1- EDBLOG

Una base di dati deduttiva, e' definita come un insieme di fatti (componente *estensionale*) e un insieme di regole (componente *intensionale*),e puo' essere vista come una teoria del prim'ordine. Data la interpretazione procedurale delle clausole Horn e posto che i fatti e le regole della base di dati siano espressi con clausole Horn, possiamo considerare una base di dati come un programma logico. E' dunque possibile trattare i problemi relativi alle basi di dati come problemi dei programmi logici in generale.
DBLOG [Asirelli 85] e' un DBMS, in cui la base di dati e' un programma logico piu' un insieme di formule che esprimono i vincoli di integrita' di due tipi:*Vincoli d'Integrita'* e *Controlli* .
Oltre al controllo dei vincoli DBLOG provvede a gestire la base di dati mediante le operazioni elementari di modifica (aggiunta e cancellazione di fatti, regole, vincoli e controlli) e di interrogazione.
La versione estesa con le transazioni (EDBLOG) [Mauro 85] include l'interprete per il linguaggio delle transazioni, basato anch'esso sulla logica, che permette di condizionare la esecuzione delle transazioni a pre-condizioni (controlli specifici della transazione in oggetto) e post-condizioni (controlli relativi alla situazione del DB dopo la esecuzione della transazione). L'uso di pre e post condizioni nella definizione delle transazioni, consente di separare i controlli globali del database da quelli relativi a particolari operazioni, riducendo cosi' il numero di controlli di consistenza che devono essere fatti periodicamente.
Riassumendo, EDBLOG risulta cosi' composto:

1) teoria delle clausole Horn, costituita da:
1.1 *un insieme di fatti* , in forma di clausole Horn unitarie (Componente Estensionale del DB) ;
1.2 *un insieme di regole*, in forma di clausole Horn definite non unitarie (Componente Intensionale del DB);
2) *un insieme di vincoli di integrita'* (integrity constraints o IC) che sono formule del tipo:

$$A \rightarrow B_1,\ldots,B_n .$$

la cui interpretazione informale e' che ogni volta che **A** e' vera allora B_1 e . ...e B_n, devono essere veri;
3) *un insieme di controlli*, che sono formule del tipo 2) oppure :

$$A_1 \wedge \ldots \wedge A_m \rightarrow B_1,\ldots,B_n$$

$$\rightarrow B_1,\ldots,B_n$$

$$A_1 \wedge \ldots \wedge A_m \rightarrow$$

4) teoria che definisce le operazioni elementari di modifica a 1) mediante le operazioni:

insert_f (A)
delete_f(A)

dove **A** e' un fatto od una regola di 1).
5) insieme di transazioni definte come:

$$A \leftarrow Pr_1,\ldots, Pr_l \mid B_1,\ldots,B_m \mid Ps_1,\ldots, Ps_n$$

4) e 5) possono anche essere considerate amalgamate, in cui i vari **Pr** e **Ps** sono precondizioni e postcondizioni da verificare in 1) (goal da eseguire), ed il cui corpo $B_1,\ldots,B_m$ e' costituito da operazioni elementari di modifica a 1) e/o da altre transazioni definite sempre in 5). L'interpretazione di un goal e' fatta nella teoria 5) (4 e 5 amalgamate) e prosegue in 1) o 4) (4 e 5 amalgamate) a seconda che si debbano valutare pre/postcondizioni di una transazione e/o operazioni elementari.

2 - IMPLEMENTAZIONE DELLA "CONFIGURAZIONE".

L'operazione che implementa la Configurazione e' realizzata mediante un gruppo

di transazioni. La caratteristica comune a tutte le Compilation Unit Comp
che costituiscono la Configurazione e' la "caratteristica", cioe' il seco
argomento della transazione principale (config(main,caratteristica)) e
coincide con il "type" delle singole cuc.
Una stessa cuc, inoltre, puo' avere piu' di un tipo (ad esempio puo' esse
"last" e "tested") e quindi e' garantita una grande flessibilita' e poten
espressiva.

1) *config(A,conf) <-- type(A,conf), compiled(Z,A), main(Z)*
| forall(deleteconf(A,X,conf))[1], *config1(A,Z,conf).*

note :
1. "forall" e' un costrutto EDBLOG che esegue l'operazione tra parentesi p
tutti i valori della variabile non istanziata (in questo caso X).
"deleteconf" e' una clausola di transazione definita successivamente,
cosi' come "config1".

2) *config1(A,X,conf) <--*
cxcl(X,Y)1,compiled(Y,W),typeW,conf),not(conftype(A,W,conf)
|insert_f(t,conftype(A,W,conf))[2]*,config1(A,Y,conf)*[3],
config2(A,Y,conf), config1(A,X,conf).

note : la prima clausola della transazione "config1" si muove sul ramo del
"context clause"; 1. Y e' una "with" di X; 2. "insert_f " e' l'operazione
EDBLOG che inserisce un fatto nel Data Base. In questo caso si inserisce i
fatto "conftype(...)" che associa al "main" A, compilato, i moduli che
compongono la Configurazione; 3. si cercano le "with" e le "subunit" di ti
"conf", compilate, per la Context Clause Y.
La transazione "config1" e' chiamata ancora per continuare a cercare le
"with" e le "subunit" di X, mentre la clausola "config2" cerca le "second
delle "library" su cui e' chiamata.

3) *config1(A,X,conf) <-- sepsub(X,Y)*[1], *compiled(Y,W),*
type(W,conf),not(conftype(A,W,conf))| insert_f(t, conftype(A,W,conf)),
config1(A,Y,conf)),config1(A,Y,conf),config1(A,X,conf).

note : la seconda clausola della transazione "config1" si muove sul ramo d
"subunit";
1. Y e' una "subunit" di X.
Non occorre richiamare config2 perche' le subunit sono secondary e quindi
hanno secondary.

4) *config1(A,X,conf) <-- |.*

note :
la terza clausola della transazione config1 e' quella terminale.

5) *config2(A,Y,conf) <-- libsec(Y,B) | config(A,B,conf).*
6) *config2(A,Y,conf) <-- |.*

note :le clausole di transazione config2 vengono chiamate per distinguere
casi in cui le "with clause", che sono library abbiano secondary o meno.

7) *deleteconf(A,X,conf) <-- conftype(A,X,conf) |*
delete_f(conftype(A,X,conf),1).

note : la transazione "deleteconf", cancella la precedente configurazione
ha la stessa caratteristica ("conf") della configurazione appena eseguita
successo. "delete_f" e' una operazione EDBLOG che cancella un fatto. "1"
significa che viene cancellato esclusivamente il fatto e non le regole che
derivano (infatti non ce ne sono).

3 - IMPLEMENTAZIONE DELLA "STORIA GENERATIVA".

history(X,Y) <-- compiled(Z,X), compiled(W,Y), libsec(Z,W).
history(X,Y) <-- compiled(Z,X), compiled(W,Y), sepsub(Z,W)[1].
history(X,Y) <-- compiled(Z,X), compiled(W,Y), cxcl(Z,W)[2].
history(X,Y) <-- exconf(Y,X)[3].
history(X,Y) <-- compiled(Y,X)[4].
history(X,Y) <-- pco(Y,X)[5].

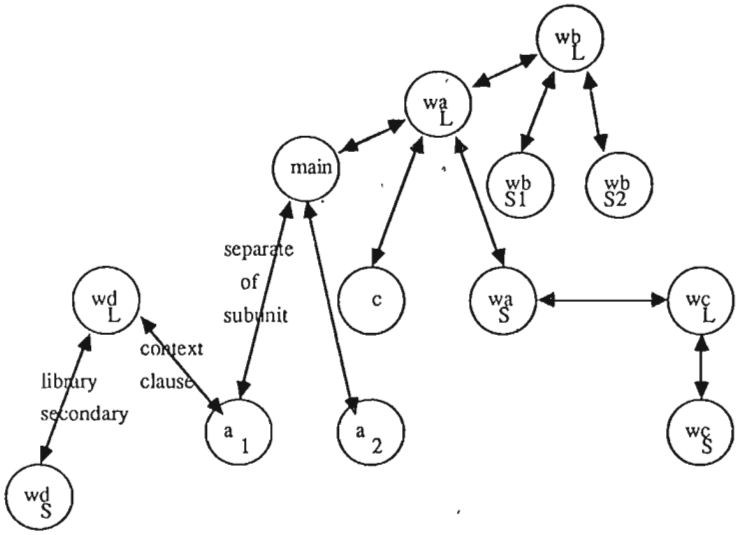
note: 1. storia di un oggetto e' il "padre" (se l'oggetto e' una subunit);
2. storia di un oggetto sono le sue "with clause";
3. storia di un eseguibile sono le cuc di cui e' il linkato;
4. storia di un oggetto e' il suo sorgente;
5. storia di un oggetto e' la Program library cui appartiene.

APPLICAZIONE DELLA "CONFIGURAZIONE" E DELLA "STORIA" AD UN PROGRAMMA Ada COMPLETO.

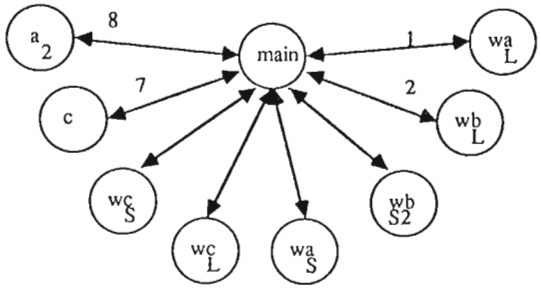
ESEMPIO DI PROGRAMMA Ada:

```
with WA; use WA ;
procedure MAIN is                          (library)
    ...
    procedure A (X : REAL) is separate ;
begin
    ...
    A (X) ;
    ...
    ...
end MAIN ;
-------------------------------------------------------
with WB ;
package WA is                              (library)
    ...
    package C is
    ...
    end C ;
    package body C is separate;
end WA;
-------------------------------------------------------
with WC; use WC ;
package body WA is                         (secondary)
    ...
begin
    ...
end WA;
-------------------------------------------------------
package WB is                              (library)
    ...
end WB ;
-------------------------------------------------------
package body WB is                         (versione "1" della secondary)
    ...
begin
    ...
end WB ;
-------------------------------------------------------
package body WB is                         (versione "2" della secondary)
    ...
begin
    ...
end WB ;
-------------------------------------------------------
procedure WC (X : REAL) ;                  (library)
-------------------------------------------------------
procedure WC (X : REAL) is                 (secondary)
begin
    ...
end WC ;
-------------------------------------------------------
with WD ;                                  (e' la versione "1" della secondary A)
separate (MAIN)
procedure A (X : REAL) is
    ...
begin
    ...
end A ;
-------------------------------------------------------
separate (MAIN)                            (e' la versione "2" della secondary A)
procedure A (X : REAL) is
begin
    ...
end A ;
-------------------------------------------------------
function WD (Y : INTEGER)                  (library)
-------------------------------------------------------
function WD (Y : INTEGER) is               (secondary)
begin
    ...
end WD ;
-------------------------------------------------------
separate ( WA )                            (secondary)
package body C is
begin
    ...
end C ;
-------------------------------------------------------
```

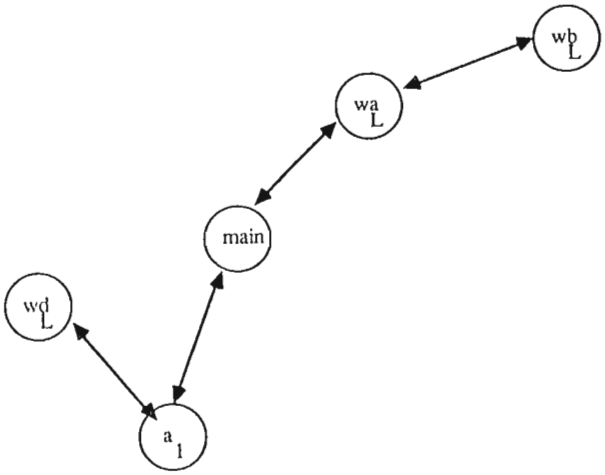
GRAFO DELLE RELAZIONI TRA I MODULI DEL PROGRAMMA DELL'ESEMPIO.



GRAFO DELL' OPERAZIONE "CONFIG(MAIN,LAST)" :



GRAFO DELLA STORIA GENERATIVA DEL MODULO A.1 :



EPSILON:
ARCHITETTURA DI UN KNOWLEDGE BASE MANAGEMENT SYSTEM OBJECT ORIENTED

Patrizia Coscia[+], Paola Franceschi[+], Giorgio Levi[*], Mario Modesti[+], Giuseppe Sardu[+], Carlandrea Simonelli[+], Luigia Torre[+]

[+]Systems & Management SpA, Pisa
[*]Dipartimento di Informatica, Universita' di Pisa

1. Introduzione.

Nel presente lavoro verranno descritte l'architettura e le caratteristiche principali di un *Knowledge Base Management System.*

Tale sistema e' in fase di realizzazione, ed il prototipo utilizza un Prolog ed un Relational Data Base Management System commerciali. L'architettura del prototipo e' mostrata in figura 1. La **Knowledge Base** e' una collezione di *teorie* e relazioni su teorie definite da *links.*

La **Teoria** e' il componente di base della Knowledge base. Le teorie sono simili ai "worlds" in MULTILOG /Kauffmann86a,86b/ e a "unit worlds" e "instances" in MANDALA /Furukawa84/. Vale a dire, le teorie sono composte da una porzione di conoscenza ed una macchina di inferenza (*theory processor*). Una teoria corrisponde ad una porzione di conoscenza e puo' essere associata ad una finestra nell'interfaccia utente grafica. Il theory processor contiene le operazioni per interrogare la teoria, per aggiornare e scandire la teoria, per caricare/scaricare la teoria, considerata come un singolo oggetto atomico. Il theory processor puo' anche contenere **tools** (debugger,tracer, explainator, query_the_user).

La componente di conoscenza in una teoria puo' essere rappresentata utilizzando diversi metodi di descrizione (p.es. tuple di un database, espressioni in algebra relazionale, programmi in vari linguaggi logici). Il theory processor e' definito in modo da supportare il corrispondente linguaggio. I linguaggi logici sono tutti estensioni e/o restrizioni di PROLOG, incluse le funzionalita' di controllo e rappresentazione della conoscenza. Nel caso di programmi scritti in PROLOG, le teorie permettono di definire una strutturazione sul workspace del PROLOG e forniscono un meccanismo di ridenominazione che consente l'overloading di nomi di oggetti definiti in teorie diverse.

Per definire i vari motori inferenziali viene utilizzata una tecnica basata sulla metaprogrammazione. Come si discutera' in seguito, l'applicazione della valutazione parziale ai metainterpreti consente di derivare programmi PROLOG efficienti. Il valutatore parziale in questione /Ghelfo86/ tratta interamente il PROLOG ed ha speciali funzionalita' per gestire efficientemente chiamate a metaprogrammi.

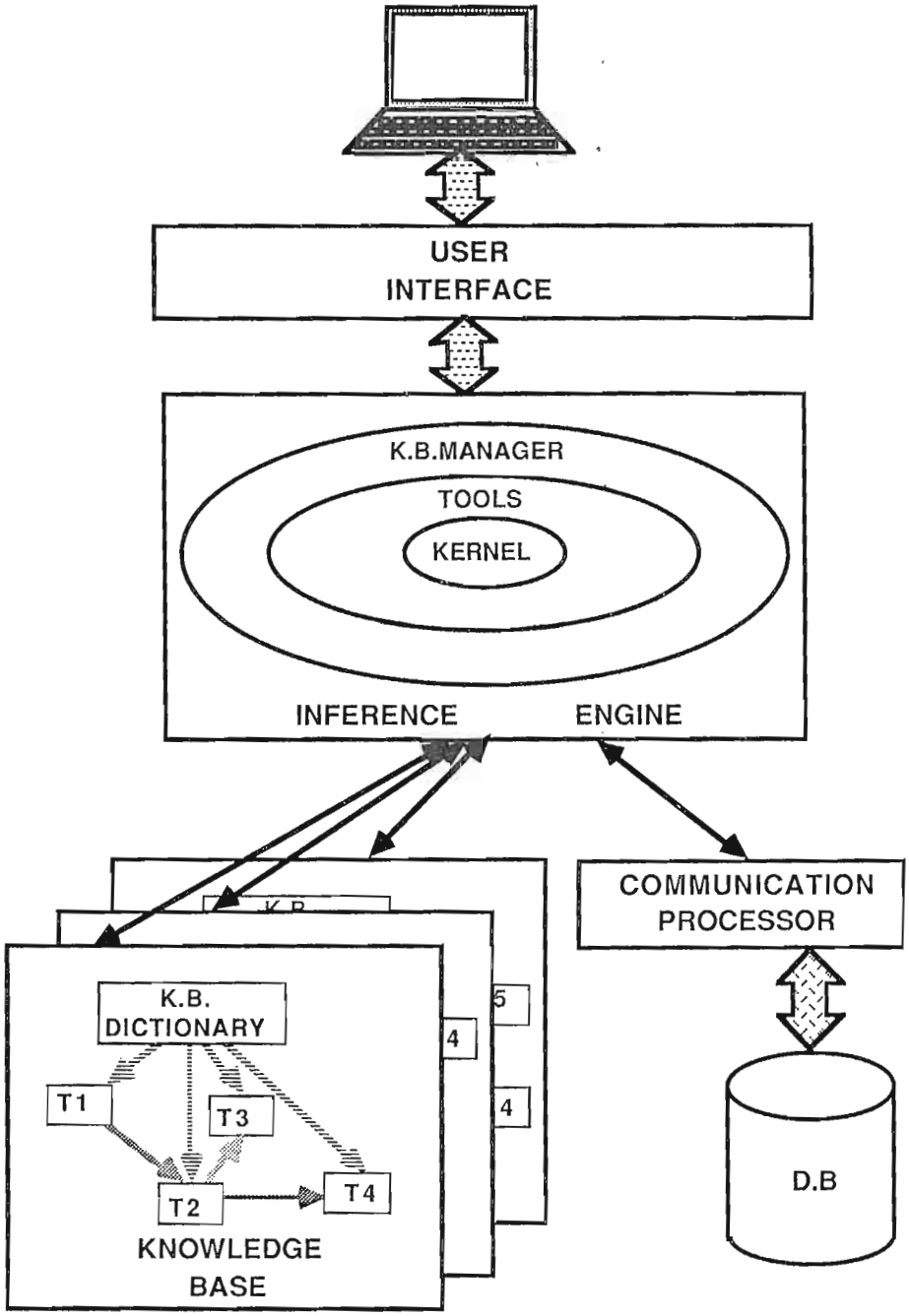

USER
INTERFACE
K.B.MANAGER
TOOLS
KERNEL
INFERENCE ENGINE
COMMUNICATION
PROCESSOR
K.B.
DICTIONARY
T1
T3
T2
T4
KNOWLEDGE
BASE
5
4
4
D.B

figura 1.

Il meccanismo di base di strutturazione della conoscenza e' la definizione di relazioni tra teorie, attraverso la specifica di appropriati **links**. Alcuni links definiscono "nuove" regole di inferenza per una teoria. In tal caso, la regola di inferenza deve essere immersa nel metainterprete di "query" del theory processor. Altre relazioni definiscono vincoli sull'aggiornamento delle teorie e sono trattate da quei componenti del motore d'inferenza che gestiscono l'aggiornamento delle teorie.

Ogni teoria puo' avere diverse teorie derivate, ottenute utilizzando appropriati strumenti di trasformazione. Anche la relazione tra una teoria e le sue teorie derivate e' rappresentata da links ed e' utilizzata per mantenere le versioni derivate consistenti con gli aggiornamenti della teoria originale. Il **Knowledge Base Dictionary** mantiene le informazioni sulle teorie e sui links.

I links piu' rilevanti verranno discussi nel capitolo 3. Nel prossimo capitolo discuteremo brevemente l'approccio metaprogramazione - valutazione parziale; nel capitolo 4 verranno presentate le caratteristiche dell'integrazione con database ed infine nel capitolo 5 verra' descritta l'interfaccia utente.

2. Metaprogrammazione e valutazione parziale.

La potenza della metaprogrammazione nei linguaggi logici e' mostrata dalla struttura del ben noto metainterprete /Sterling84/.

```
1.solve(true) :-!.
2.solve((Q1,Q2)) :-!,solve(Q1),solve(Q2).
3.solve(Q) :-clause(Q,Body),solve(Body).
```

La definizione e' cosi' semplice e breve perche' la maggior parte del lavoro e' compiuta dal sottostante "vero" interprete del livello oggetto. Il metainterprete non definisce esplicitamente le due procedure computazionalmente piu' complesse, cioe' il backtracking e l'unificazione. Il backtracking e', in effetti, implicito nella clausola 3, poiche' l'atomo clause(Q,Body) e' "backtrackable". L'unificazione, d'altra parte, corrisponde direttamente all'unificazione del livello oggetto.

La definizione di metainterpreti *arricchiti* diventa attraente, perche' permette di definire nuove funzionalita' senza modificare il programma (la conoscenza al livello oggetto) e l'interprete del linguaggio (il motore inferenziale). La definizione e' economica e conduce ad interpreti che sono ragionevoli dal punto di vista della performance. I metainterpreti estesi possono definire nuove strategie di controllo, arricchire il linguaggio logico con nuovi costrutti (per esempio, struttura della conoscenza, o incertezze) e le relative regole di inferenza (ereditarieta' o ragionamento approssimato), o definire strumenti di analisi per fornire funzionalita' tipiche dei sistemi esperti (explanation, query_the_user, ecc), o monitoraggio interattivo (debugger, tracer, ecc). Una delle caratteristiche principali della metaprogrammazione e' la capacita' di estendere il linguaggio, il motore inferenziale e l'ambiente, senza modificare i componenti di base

del sistema, cioe' l'interprete e il compilatore PROLOG. Le estensioni definite come metaprogrammi (PROLOG) sono portabili e facili da definire, analogamente a quanto accade negli ambienti LISP. In questo caso, le prestazioni sono piuttosto scarse, se paragonate a quelle ottenibili con un implementazione ad_hoc del nuovo linguaggio/ambiente; tale soluzione, tuttavia, e' molto costosa e, inoltre, non e' necessariamente aperta ad ulteriori estensioni e modifiche. La metaprogrammazione, viceversa, e' semplice, piu' flessibile e pulita, poiche' la conoscenza (le regole nel linguaggio eventualmente esteso) e il motore inferenziale (il metainterprete) sono separate e facili da capire, e tutte le estensioni del motore d'inferenza sono definite chiaramente a meta-livello.

Il vero punto debole della metaprogrammazione e' la performance. Esiste, tuttavia un'interessante tecnica (la valutazione parziale dei metainterpreti) che permette di combinare il basso costo e l'alta flessibilita' della metaprogrammazione con la performance.

Nella valutazione parziale dei metaprogrammi, i valori parziali di input sono chiamate di procedura. La valutazione parziale del metaprogramma M applicato ad una chiamata della procedura P genera una specializzazione di M, che puo' essere vista come una nuova versione di P. La nuova definizione di P e' una procedura P' completamente nuova, che include alcune delle caratteristiche rilevanti di M. Questo permette di sostituire una metachiamata di P (attraverso M) con una chiamata diretta a P', poiche' l'esecuzione diretta di P' e' equivalente all'esecuzione di P attraverso il metainterprete M.

Se M e' il metainterprete puro (senza nuove regole di inferenza o funzionalita' estese), P' e' equivalente a P. Se il metainterprete M contiene regole di inferenza aggiuntive (e le corrispondenti funzionalita' aggiuntive), la valutazione parziale compila le nuove funzionalita' nella procedura P'.

Riassumendo, gli aspetti chiave che rendono interessante la valutazione parziale nel caso di metaprogrammi sono la traduzione delle metachiamate in chiamate dirette e la valutazione, a tempo di compilazione, della conoscenza statica. La valutazione parziale permette quindi di combinare la flessibilita' della metaprogrammazione con l'efficienza.

3. Links e theory processor.

Si possono utilizzare i meccanismi di strutturazione della base di conoscenza, ossia teorie e legami, per definire i processors delle teorie. Un motore inferenziale, definito in una (meta)teoria M1 puo' essere ereditato da una teoria T1 se nel Dizionario della Base di Conoscenza e' definito un link **class**(T1,M1). Il link di tipo "class" definisce una relazione di ereditarieta' fra una teoria a livello oggetto ed una teoria a metalivello che contiene il suo motore inferenziale.

Un motore inferenziale e' definito per mezzo di metaprogrammi, i quali, a loro volta hanno bisogno di un altro meccanismo di inferenza: in altre parole, anche un motore inferenziale e' una teoria a livello oggetto legata da un link di tipo "class" a un'altra teoria a

livello. Il motore inferenziale delle teorie base della gerarchia tta dal link "class" e' il PROLOG esteso con un meccanismo di izionamento del workspace e con il trattamento del link "class". Il "class" puo' introdurre diversi livelli di metainterpretazione, izionando negativamente la performance. Tuttavia i vari livelli di interpretazione possono essere compilati per mezzo della tazione parziale.

e teorie possono comunicare e cooperare sia esplicitamente che per itarieta'. Nel primo caso, una teoria T1 puo' comunicare con una ia T2 per mezzo di un riferimento diretto al motore inferenziale di

Specifici links possono limitare l'accesso diretto alla modalita' **y-only** (possibilita' di chiamare solo il componente di query del ory processor) oppure consentire anche la modalita' **write** sibilita' di modificare la teoria, attraverso il componente di te deltheory processor).

La comunicazione per ereditarieta' e' realizzata definendo un link tipo **inherit** da T1 a T2. Questo link e' interpretato dal query iler di T1 nel seguente modo: se un sottogoal non puo' essere olto in T1, allora e' trasformato in una chiamata al query handler T2.

Nel caso mostrato in figura 2, dove le due teorie condividono lo sso meccanismo di inferenza, il risultato e' essenzialmente reditarieta' della conoscenza a livello oggetto. Il caso in figura 3 tipico invece dell'interazione tra teorie che hanno meccanismi di erenza diversi. Come esempio si puo' considerare il caso in cui T1 un programma logico e T2 e' l'insieme di tuple di un database. Il abase e' visibile al programma logico attraverso una specifica erence engine, cioe' il database query processor.

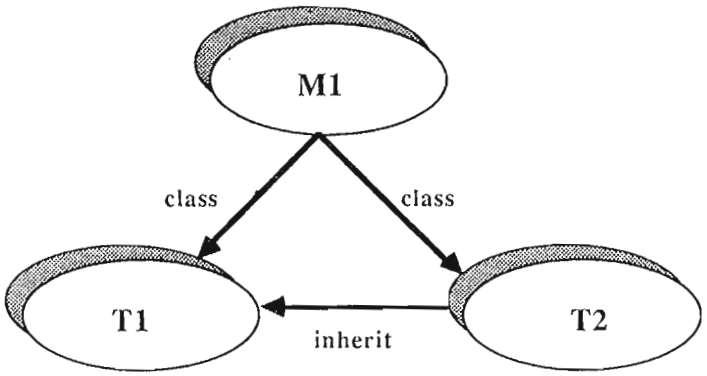


Figura 2.

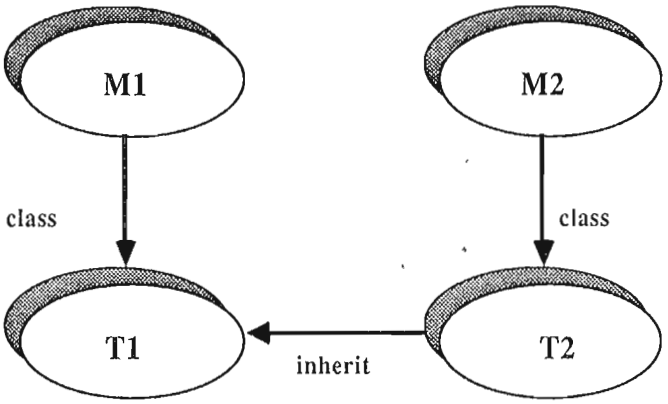


Figura 3.

Consideriamo ora un altro link importante, che condiziona il componente di aggiornamento dei motori inferenziali, cioe' il link di tipo **constraint**. Una teoria T2 puo' definire i vincoli di integrita' per la teoria T1; in tal caso, la teoria T1 dovra' essere legata con un link di tipo "constraint" alla teoria T2 e ogni operazione di aggiornamento su T1 dara' luogo alla chiamata di un apposito componente dell'inference engine di T2 che controlla che l'aggiornamento sia consistente con i vincoli presenti in T2.
Il motore inferenziale di T2 supporta il linguaggio usato per definire i vincoli di integrita'.

Il sistema fornisce diversi tools di trasformazione e di ottimizzazione, per esempio un valutatore parziale di metaprogrammi e i tools di trasformazione usati per l'ottimizzazione degli accessi al database (che includono la valutazione parziale di teorie contenenti programmi che accedono a database e la loro trasformazione in espressioni in algebra relazionale). L'applicazione di un tool di trasformazione P ad una teoria T genera una teoria derivata T', che e' legata a T da un link di tipo **version**.

Consideriamo l'esempio in figura 4, dove T3 e' ottenuta valutando parzialmente la conoscenza in T1 rispetto al metainterprete "demo", che appartiene all'inference engine di T1. T3 e' legata a T1 con il link **version(demo)**. La semantica di tale link ha due aspetti. Quando si trova una chiamata di "demo" applicata a qualche predicato di T1, essa viene convertita nella corrispondente chiamata in T3. Inoltre, quando T1 viene modificata, T3 deve essere mantenuta consistente con le modifiche di T1. Si noti che la compilazione di un livello di interpretazione si riflette nel fatto che l'inference engine di T3 e' ora M2.

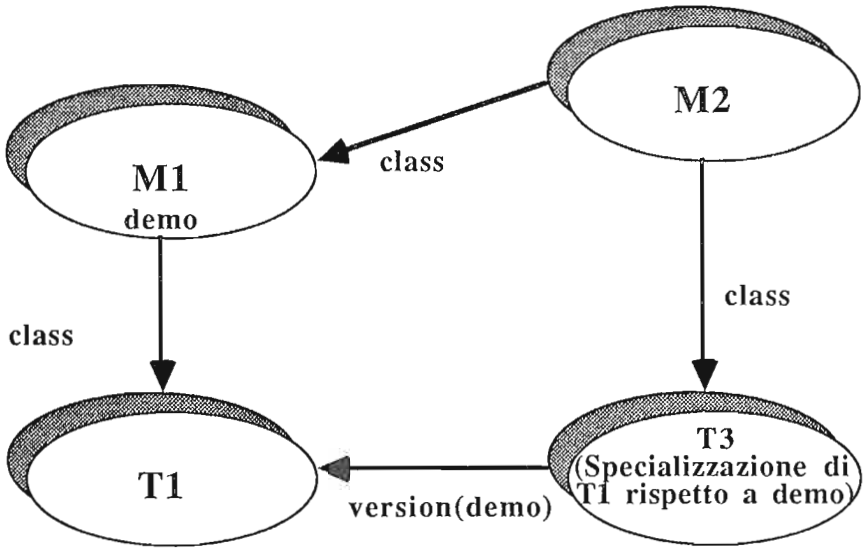


Figura 4.

I links mostrati rappresentano un insieme minimo fondamentale per le scelte architetturali. Altri tipi di link possono essere definiti e interpretati per mezzo di nuovi motori inferenziali, definendo cosi' nuove funzionalita' del linguaggio. Possono essere definiti links specifici per i tools: un esempio, relativo alla valutazione parziale, e' la possibilita'di definire quali dei links esistenti sono visibili alla valutazione parziale. In questo modo si possono rendere i risultati della valutazione parziale indipendenti dal contenuto corrente di certe teorie (tipicamente, teorie contenenti conoscenza incompleta o in evoluzione).

Esempio. Il problema dei tre saggi.

Questo breve esempio vuole mostrare come EPSILON puo' essere utilizzato per strutturare la conoscenza mediante teorie e links nella risoluzione di un problema; mostra come teorie e links possono servire a definire conoscenze separate e conoscenza condivisa; illustra infine come puo' essere trattata la conoscenza dinamica.

Il problema. Ogni saggio ha un cappello. Almeno un cappello e' bianco. Ogni saggio puo' vedere i cappelli degli altri saggi e, alla domanda se il suo cappello e' bianco, rispondera' "si'", oppure "no" comunicando una conoscenza di tipo "non so" agli altri saggi.

La struttura della nostra soluzione e' mostrata in figura 6. La conoscenza generale, condivisa da tutti i saggi, e' contenuta nella teoria **wiseman**. La specifica conoscenza dinamica di ogni saggio e' contenuta in una teoria **view**. Ogni saggio e' rappresentato da una teoria che eredita la conoscenza in **wiseman** e nella corrispondente **view** (link **inherit**). Ogni saggio puo' anche comunicare conoscenza

alle views degli altri saggi (link **write**). Nella figura vi sono due tipi di link inherit, che si distinguono nel fatto di essere o no visibili (ed utilizzabili) per il valutatore parziale. Nell'esempio, i links tra i saggi e le rispettive views non sono visibili, poiche' le views contengono conoscenza dinamica.

Tutte le teorie appartengono alla stessa classe (il link "class" non e' mostrato in figura) ed il loro metainterprete di query e' demo, che controlla i links e, se trova un' link inherit, definisce l'ereditarieta'.

La conoscenza nella teoria **wiseman** e' la seguente:

```
answer(yes) :- i_am(W), white(W),!.
answer(no) :- i_am(W), next(W,WN),next(WN,WNN),
              view(WN,View2), view(WNN,View3),
               assert((white(WN):-not(white(WNN))),View2),
              assert((white(WNN):-not(white(WN))),View3), fail.
answer(no) :- i_am(W), view(W,View1), propagate(W,View1).

white(W) :- next(W,WN), next(WN,WNN), i_am(X), view(X,V),
            not(call(white(WN),V)), not(call(white(WNN),V)).

propagate(W,View) :- clause(white(W),B,View), infer(B), fail.
propagate(W,View).

infer(true) :- !.
infer(not(white(X))) :- view(X,V), assert(white(X),V),!.

next(wiseman1,wiseman2).
next(wiseman2,wiseman3).
next(wiseman3,wiseman1).

view(wiseman1,view1).
view(wiseman2,view2).
view(wiseman3,view3).
```

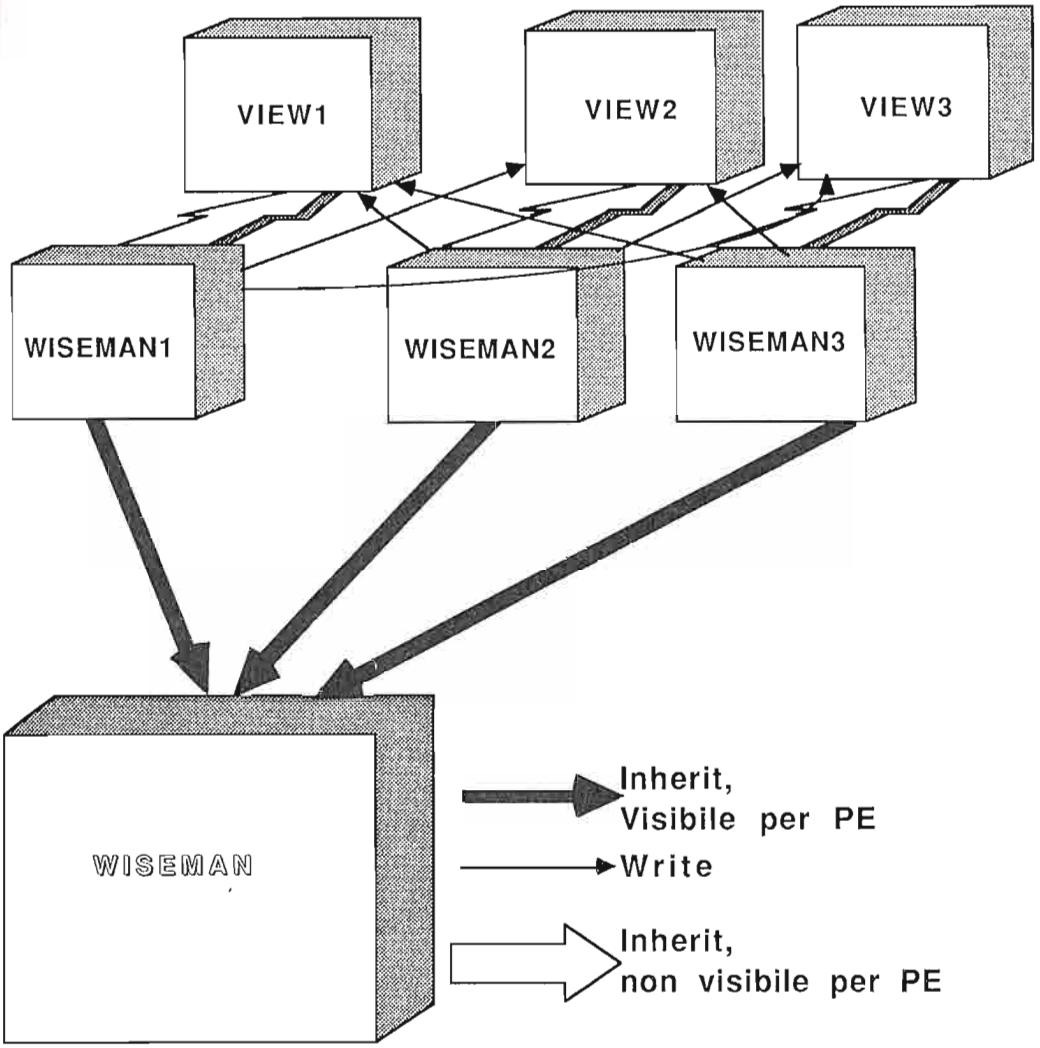


Figura 5.

La teoria contiene della conoscenza sui tre saggi (next), le loro views (view), ed ha essenzialmente due regole. La prima e' la procedura di decisione nel caso "yes" (cioe', gli altri cappelli non sono bianchi). La seconda comunica la risposta "non so" agli altri saggi, aggiungendo nuove regole alle loro views. Le nuove regole sono derivate dal fallimento delle regole esistenti.

Consideriamo lo stato iniziale mostrato in figura 6, dove tutti i cappelli sono bianchi.

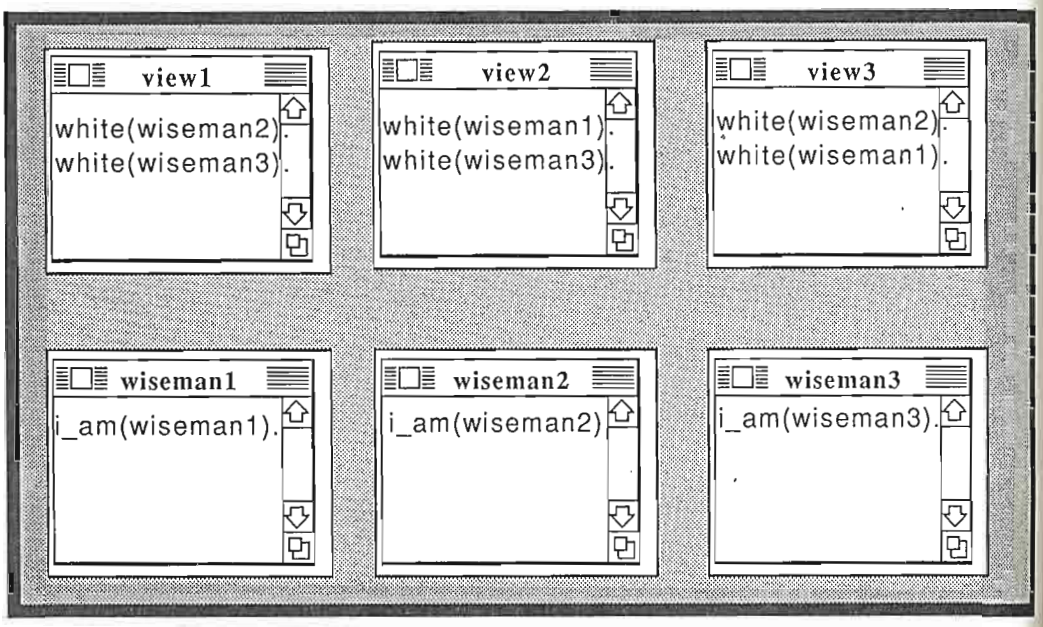


Figura 6.

Ogni saggio puo' essere interrogato eseguendo **answer(X)** nella corrispondente teoria. Per esempio, la query per interrogare i tre saggi uno dopo l'altro e'

demo(answer(X),wiseman1),demo(answer(Y),wiseman2),
demo(answer(Z),wiseman3).

che dara' X=no, Y=no, Z=yes. Lo stato risultante delle teorie e' mostrato in figura 7, dove le teorie **view** contengono anche le nuove regole, inferite dai fallimenti.

Ricordiamo infine che **demo** e' in pratica un metainterprete, che controlla varie proprieta' della knowledge base e, nel caso di links di *tipo inherit, simula l'ereditarieta'* (per esempio, la teoria **wiseman1** eredita **wiseman** e **view1**).

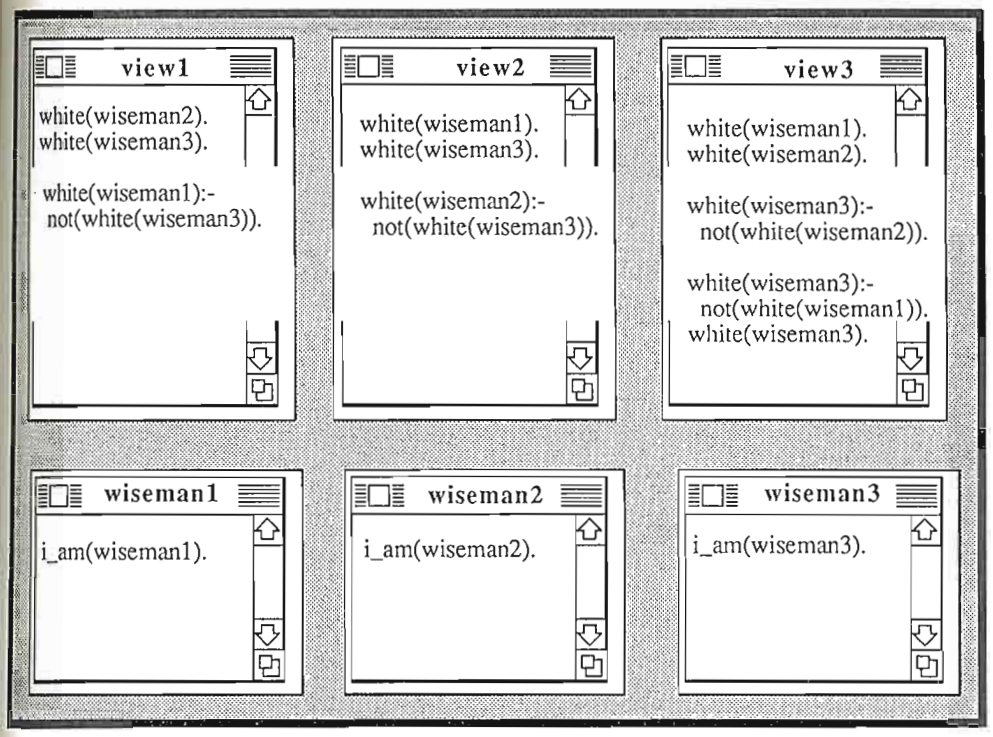


Figura 7.

4. L'integrazione col Database

Il meccanismo di strutturazione in teorie viene utilizzato anche per la gestione di database. In quest'ambito, il meccanismo delle teorie viene impiegato per rappresentare:

- il motore di inferenza del database (*communication processor*);
- teorie dei tipo database, che sono l'immagine PROLOG di tuple memorizzate in un database esterno. Il loro motore inferenziale e' il *communication processor;*
- i dizionari dei dati, ossia la descrizione in PROLOG dei dizionari dei dati dei database presenti nel sistema. Un dizionario dei dati e' legato alla corrispondente teoria database da un link di tipo dict;
- i vincoli di integrita';
- i risultati di interrogazioni sul database. Il programma logico che rappresenta le risposte risultato di una query al database puo' essere memorizzato in una teoria.

Come primo e piu' semplice esempio, consideriamo il caso di interazione secondo il metodo interpretativo. Nella precedente figura 3, si consideri la teoria T1 contenente un programma logico e la

teoria T2 come una teoria database, essendo M1 il motore di infer
della teoria T1 e M2 il database processor. Da notare che in qu
caso non si richiede alcuna meta conoscenza particolare sul datab
poiche' T1 eredita direttamente dal database. Infatti, ogni volta
il query processor di M1 non riesce a risolvere una query in T1, v
attivato il query processor di T2, che reperisce le tuple contenut
T2 che soddisfano la query e le restituisce, sotto forma di progr
logico, in una nuova teoria T3. A questo punto il query processor i
risolve la query in M3.

Questa soluzione e' stata realizzata utilizzando un data
commerciale (Informix), definendo il communication processor co
insieme di predicati per l'accesso e la ricerca sul database.

Nel caso generale, in cui si prevede che la base di conosc
contenga piu' di una teoria database, vengono forniti degli strum
di trasformazione ed ottimizzazione che permettano di ridurre il nu
di accessi al database, raggruppando le query e ritardando l'access
database fino ad avere il massimo numero di variabili istanziate.

5. L'interfaccia utente.

Le interfaccie utente sofisticate e amichevoli giocano un r
sempre piu' importante in sistemi di ogni tipo, ed in partico
quando i sistemi da interfacciare sono complessi e richiedono un
grado di interazione, come nel caso di databases e sistemi basati s
conoscenza.

L'interfaccia utente di EPSILON e' stata realizzata facendo us
un Personal Computer e puo' quindi essere considerata a tutti
effetti come un processo separato che interagisce con il sis
mediante scambio di messaggi. Questo approccio oltre a consentire
di capacita' grafiche esterne a vantaggio dell'efficienza del si
offre i seguenti vantaggi:

- non richiede modifiche o estensioni all'interprete PROLOG;
- permette la definizione di un semplicissimo meccanism
  comunicazione tra l'interfaccia e gli strumenti del si
  EPSILON;
- l'interfaccia e' largamente indipendente dall'interprete P
  utilizzato;
- e' possibile implementare l'interfaccia utilizzando un ligu
  procedurale, che e'piu' adatto a risolvere questo tip
  problemi;
- consente l'uso di un personal computer come periferica dedic
  intelligente.

Questa ipotesi prevede un canale di comunicazione bidirezi
che connette il processo interfaccia col resto del sistema,
flusso di informazioni tra i due avviene utilizzando i linguaggi p
di ciascun componente: l'interfaccia utente invia richieste al
sotto forma di goal PROLOG, e riceve richieste espresse in lingu
grafico.

Oltre a modalita' di editing piu' sofisticate rispetto ai s

PROLOG usuali, l'interfaccia utente mette a disposizione funzionalita' quali:

- creazione di una nuova finestra di tipo standard per le teorie;
- creazione di una finestra di emulazione terminale per i tools;
- visualizzazione del testo del programma logico associato ad una teoria;
- visualizzazione delle informazioni relative ad una teoria.

6.BIBLIOGRAFIA

/Furukawa84/ **K. Furukawa, A. Takeuchi, S. Kunifuji, H. Yasukawa, M. Ohki and K. Ueda**, Mandala: A logic based knowledge programming system, Proc. Int'l Conf. on Fifth Generation Computer Systems (1984), 613-622.

/Ghelfo86/ **S. Ghelfo and G. Levi**, A partial evaluator for metaprograms in a multiple theories logic language. Epsilon Project Report (October 1986).

/Kauffman86a/ **H. Kauffmann and A. Grumbach**, Representing and manipulating knowledge within "worlds". Proc. First Int'l Conf. on Expert Data Base Systems, L. Kershberg, Ed. (1986), 61-73.

/Kauffman86b/ **H. Kauffmann and A. Grumbach**, MULTILOG: MULTIple worlds in LOGic programming, Report (1986).

/Sterling84/ **L. Sterling**, Expert system = Knowledge + Meta-interpreter, Tech. Rep.CS84-17, Weizmann Institute of Science, Rehovot, Israel (1984).