Spedizione in abbonamento postale, Gruppo IV, 1° semestre.
Pubblicità Inferiore al 70%.
N. 35, Gennaio - Aprile 1989
TAXE PERCUE

UNIVERSITA' DEGLI STUDI
DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE

Honeywell Bull
CENTRO STUDI
INFORMATICA
E AUTOMAZIONE

NOTE DI SOFTWARE

DICEMBRE 1988
42/43
MARZO 1989

Note di software è una pubblicazione trimestrale, edita a cura del Dipartimento di Scienze dell' Informazione dell' Università di Milano e dal Centro Studi Informatica e Automazione della Honeywell Bull Italia.

Note di software intende costruire uno strumento per la circolazione di informazioni, idee, metologie ed esperienze nell'area delle tecnologie e della produzione del software.

La collaborazione a **Note di software** è libera. Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Direttore responsabile o con il Comitato di Redazione. In particolare si sollecitano contributi nella forma di monografie, da inviare in triplice copia, e che non dovrebbero superare le cinquemila parole.
I lavori ricevuti per la pubblicazione verranno revisionati dai membri del Comitato di Revisione Scientifica, che possono avvalersi della collaborazione di specialisti del settore. I lavori pubblicati riflettono comunque le opinioni dei loro autori.

Note di software viene distribuito ad Aziende, Enti e specialisti del settore che ne facciano richiesta alla Redazione.

Direttore Responsabile: Franco Filippazzi
Honeywell Bull Italia S.P.A., Centro Studi Informatica e Automazione, via Viola 11 - 20127 Milano

Comitato di Redazione: Stefania Bandini, Francesco Gardin, Roberto Polillo
Università di Milano Dipartimento di Scienzee dell' Informazione
via Moretto da Brescia, 9 - 20133 Milano tel. (02) 2772233

Comitato di Revisione Scientifica: Alberto Bertoni, Gianluigi Castelli, Giovanni Degli Antoni, Giorgio De Michelis, Mario Italiani, Gaetano Aurelio Lanzarone, Giancarlo Martella, Marco Maiocchi, Giancarlo Mauri.

Autorizzazione del Tribunale di
Milano n. 87 del 23 febbraio 1977

Supporto tecnico - gestionale e distribuzione: Domenico Asinelli
Honeywel Bull via Tassoli 6
20154 Milano Tel. 02 - 67792960

UNIVERSITA' DEGLI STUDI
DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE

Honeywell Bull
CENTRO STUDI
INFORMATICA
E AUTOMAZIONE

DICEMBRE 1988
42/43
MARZO 1989

QUESTO NUMERO

Questo numero doppio di Note di Software si apre con un intervento di Gianni Degli Antoni sui mondi artificiali e con un breve commento di P. Morini su un problema che ultimamente ha fatto parlare di sé in moltissimi ambienti della programmazione: si tratta dei "virus", programmi che manifestano un comportamento molto analogo a strutture biologiche infettive.

Seguono poi due lavori di rassegna: il primo, a firma di G. Bottoni, M. Cremonesi e I. De Lotto, fornisce un chiaro quadro dei cambiamenti degli ultimi anni nell'ambito dell'evoluzione delle tecnologie hardware e software degli elaboratori con riferimento al calcolo tecnico-scientifico. Il secondo è presentato da N. Cesa-Bianchi e da C. Ferretti e illustra due noti stumenti software per la simulazione di reti neuronali; si tratta di un argomento di grande attualità per il settore dello studio di modelli connessionisti.

R. Grande introduce poi un argomento che in questi ultimi tempi ha riscosso molti interessi di ricerca: la possibilità di utilizzare il paradigma di programmazione object-oriented come strumento di rappresentazione della conoscenza e come metodologia di programmazione nell'ambito della tecnologia della programmazione logica.

Seguono poi due contributi di autori stranieri: il primo é di J. C. Ebergen e presenta i circuiti delay-insensitive, ovvero i componenti il cui comportamento funzionale é indipendente dai ritardi nella comunicazione o negli elementi. Il secondo tocca un argomento nel settore dell'informatica medica. Esso riporta un' esperienza con un sistema di supporto alla decisione nel campo della chirurgia con particolare riferimento ai problemi di interfaccia uomo-macchina.

G. C. Macchi tratta invece delle "eccezioni", un argomento che partecipa agli aspetti di fault-avoidance e di fault-tollerance che il software deve possedere per far fronte a malfunzionamenti dovuti a errori di progetto o di debugging.

Infine, P. Stofella ci fornisce un resoconto del convegno AICA sulla Elaborazione Parallela tenutosi a Milano nel novembre scorso.

LA REDAZIONE

ARTIFICIAL WORLDS

Gianni Degli Antoni
Dipartimento di Scienze dell' Informazione
Università di Milano

Computers sciences and tecnologies have never been so active as in these days. Industrial and research laboratories seem to produce new ideas at an increased rate. Shows and shops are full of new companies and new products. Competition forces the decreasing of computer prices.
Technicians as well as users are frustrated in triyng to understand an overall behaviour of the technology evolution. The software designer relies on intuition in order to define products and methodologies. In the market old phenomenas are appearing confirming that when it is difficult to make a choice the status symbol can be used with success.

In spite of this tremendous complexity something can be perceived quite cleary while looking at the best promises of our time. The human interface is certainly improving in various directions: mouses, better displays, graphical interfaces, windows, buttons or links and ipermedial interfaces all are diffusing everywhere and now are not proposed only by a limited subset of companies. Moreover large storage read only optical discs are diffusing for new reasons: data transport and realibity. A read only storage device is reliable; it has not to be backed up; software virus can do nothing against it; it is quite cheap to ask for copies from the master. And it is not difficult to organize the work for preparing the master itself. The workstation with a read only optical disc (actually a CD - ROM) becomes similar to a living biological cell where DNA is the read only information controlling the whole cell behaviour.

The art of writing programas is changing too: finally everybody is now understanding that object oriented technology can be extended to our programming languages: some LISP's, C++ and other have incorporated the methodology ideas. Others will do the same.
Meanwhile many are experiencing the advantages of using object oriented programming in the development of complex systems using available languages.
Unfortunately the news in interfaces, optical storage devices and programming are not enough to understand what is going on and what to do. We need a paradigm suitable to drive the technologist as well as the user in designing or using solutions. The paradigm can be perceived from the real world or can be detected from the history of computing. In the real world it is seriously evident that humans act on computers as they act in the reality.

Someone has seriously considered the evidence that men are in front of an artificial reality.
To show such a choice they have created on the video iconic symbols of pieces of the world XEROX, APPLE and others have used such a solution for a limited set of artificial objects.

Since now understanding what's going on, we can abandone these limitations and we can extend our metaphor to the other pieces of the world. Buttons and hypermedial interfaces help us: using a button is like doing something in the world. Representing the physical world on the display is a great cognitive help: everyboby is largely acquainted with reality and has not difficult in recognizing representations and using them.
Representing pieces of the physical world as interacting objects is made easy by object oriented programming. Browsing in a system based on a well known metaphor is easy. Testing, debugging, criticizing or improving such a system can be done easily. Starting from a predefined archetype to every object graphically represented on the display a small piece of software will be associated. Taking into account the various pieces of the artificial world it will be a simple approach to verify the entire system.

An example will help. Let's consider the design of an information system for a Computer Science Department. The department will appear on a map as a button. With other buttons it will be possible to enter in offices, for example to verify the computers available in that place, the people working and other entities (software, books, data, procedures,...). The network connecting the computers will be visible on request as well as the all available details.
To make the system useful some perception has to be given: it will mean that we will see on the display a printer in some office. This requires to add to the models some perception as a consequence.
Such an information system will be probably easy to use for everybody while the expert user will follow short cuts. Artificial reality as a paradigm suitable to design information systems supports strongly the underlyng ideas of packages like HYPERCARD (™), NEW WAVE and others. These packages can be seen as metaphors, or as general pourpose interfaces. It is not difficult to introduce some artificial intelligence techniques into the picture: as intelligence will be required: these will be represented like entities on the display. Some activity will be delegated to agents. The human user will finally decide if the agents have done the task correctly or not.

Now the picture should be clear: every workstation in a network has its own read only data bank corresponding to slowly varying (structural) information well integrated with every day data on volatile storage devices. In front of each display actions can be taken about the artificial as well as about the real world. Interraction of the real world with the user is through percepition as due to networks and sensor. Ipermedial interfaces are ideal for such a picture.
To implement the suggested model object oriented programming is certainly the best approact.
World archetypes or metaworlds are requested in order to implement our artificial worlds just as a CAD system is used to make drawings.
It is not difficult to recognize in today computer tecnnology fragments of the suggested picture: what matter is that such a picture is (consciously or not) driving technology in new direction suitable to put all pieces of the computer world together in a single technology with a degree of realism related to the application field.

NOTE DI SOFTWARE 42/43

VIRUS E COMPUTERS

Paolo Morini
Laboratorio di Informatica Musicale
Dipartimento di Scienze dell' Informazione
Università di Milano

Come molti già sapranno da qualche tempo sono in circolazione alcuni virus che hanno la singolare prerogativa di contagiare i computers invece degli uomini; tuttavia ben pochi sanno come agisce un siffatto virus, e ancora meno sanno come prevenire il contagio e difendersi da eventuali attacchi. E' giunto il momento di fare luce su questo aspetto oscuro del mondo dei computers.

Molti infatti pensano che questi "fantomatici" virus esistano solo nelle parole di persone dotate di notevole fantasia, e che in realtà siano, se pur teoricamente possibili, troppo complessi per essere realizzati; oppure che sono presenti solo in limitati e controllati esperimenti di ricerca (forse militare?) che non hanno alcuna ripercussione sul mondo esterno. Purtroppo la situazione reale è molto differente: i virus sono relativamente facili da costruire e ne esistono già diversi in diffusione rapida sui vari tipi di personal computer. A conferma di ciò vale la comparsa di uno di essi al Dipartimento di Scienze dell'Informazione dell'Università di Milano, dove é stato isolato, studiato e reso inattivo (dall'autore grazie all'aiuto di David Bianchi, Michele Bohm e Alberto Sametti).

Appurato che esistono, non abbiamo ancora spiegato che cosa sono in realtà. Un virus è un pezzo di codice eseguibile che non ha vita autonoma, ma si installa su sistemi e programmi esistenti e ne modifica il comportamento, eseguendo azioni che non ne migliorano affatto il funzionamento e possono essere anche disastrose, come la formattazione del disco, rimozione di files, ecc... Perché questo codice sia considerato un virus deve possedere anche una singolare caratteristica: dev'essere capace di autoinstallarsi su (cioè infettare) un sistema o un programma che ne sia sprovvisto (sano) che viene introdotto nello stesso ambiente in cui è attivo il virus (al contatto).

La scelta del nome appare a questo punto ovvia: la capacità di riprodursi e diffondersi è del tutto assimilabile a quella dei virus biologici. Ma l'analogia non si ferma qui: con una visione certo semplificata ma corretta ogni catena di DNA è un programma che viene eseguito da una cellula, e un virus è un segmento di DNA esterno che si fa eseguire insieme a quello originale modificando l'ambiente che lo circonda. La similitudine è perfetta.
Non è comunque il caso di spaventarsi e buttare i computers dalla finestra: i virus possono essere vinti. Esistono programmi chiamati "vaccini" che trovano ed eliminano i virus, ovvero guariscono i sistemi e i programmi infetti; ma, come tutti i vaccini, sono attivi solo su alcune famiglie di virus; il principio della selezione naturale (altro riferimento biologico!) comporta che i virus posti in circolazione sono sempre più perfezionati e nascosti per aggirare i metodi di difesa degli utenti, a loro volta sempre più progrediti.

A questo punto risulta ovvio che la miglior difesa è la prevenzione e l'informazione tempestiva, in modo da

limitare al minimo il contagio. Appena viene scoperto un virus lo si deve comunicare a tutte le persone che hanno materiale che può essersi infettato: bisogna diffondere la notizia più velocemente di quanto il virus diffonda sé stesso. Contemporaneamente, in collaborazione con persone esperte, bisogna studiarlo allo scopo di capire quali operazioni effettuare per estirparlo o renderlo innocuo. In attesa di questo è bene non utilizzare il materiale che si teme sia infetto; l'isolamento è il solo mezzo sicuro per evitare la diffusione del contagio.

A proposito di diffusione dell'informazione, qualcuno sarà curioso di sapere come funziona il virus in cui ci siamo imbattuti. Come premessa sarà bene specificare che lavora solo in ambiente Macintosh; inoltre ha un'azione solo dimostrativa, ovvero in certe condizioni emette un beep. E' però strutturato in modo tale che se ne viene messa in circolazione una nuova versione con un'azione diversa (verosimilmente più dannosa) si aggiorna automaticamente; è quindi importante rimuoverlo non per i danni che provoca ma per quelli che potrebbe causare.

Vediamo ora nel dettaglio le modalità di contagio (1): l'effetto finale è che un programma sano lanciato attraverso un sistema infetto si infetta a sua volta e viceversa. In particolare il contagio avviene in tre fasi:

1 - Quando viene lanciato un programma infetto la entry point (indicatore della prima istruzione da eseguire) modificata indica che deve essere eseguito il segmento codice [CODE,256], che fa parte del virus; questo controlla che il sistema sia infetto, e in caso negativo lo rende tale. In particolare gli aggiunge la risorsa [INIT,32]. Quindi viene passato il controllo al vero inizio del programma.

2 - Al boot da un sistema infetto viene eseguito il codice della risorsa [INIT,32] che sostituisce alla funzione di ToolBox TeInit il codice contenuto in [nVIR,1].

3 - Quando viene lanciato un programma sano, nella fase di inizializzazione viene quasi sempre eseguita la TeInit; se il sistema è infetto il controllo passerà quindi al codice di [nVIR,1] che dopo aver eseguito la vera TeInit verifica che l'applicazione sia infetta e in caso negativo la rende tale. In particolare viene sostituita la entry point del programma in modo che venga eseguito prima il segmento codice appositamente aggiunto [CODE,256].

Per decontaminare un sistema o un programma infetti bisogna, agendo attraverso un resource editor tipo ResEdit, rimuovere tutte le risorse di tipo nVIR ed anche [INIT,32] dal sistema e [CODE,256] dai programmi; solo per i programmi è inoltre necessario prima prendere nota degli 8 bytes contenuti in [nVIR,2] e copiarli in [CODE,0] a partire dal 16° byte, ricoprendo il contenuto precedente. E' opportuno eseguirne la rimozione dal System, dal Finder e da tutte le applicazioni sul disco in una volta sola, senza uscire dal resource editor (e non dimenticatevi di lui!), quindi riavviare immediatamente la macchina. In altro modo si rischia, lasciandone anche una sola copia circolante, di infettare di nuovo il sistema e tutti i programmi.

(1) Per poter effettuare le operazioni sotto descritte bisogna conoscere il concetto di risorsa tipico di Macintosh; la notazione [XXXX, n] indica il riferimento alla risorsa di tipo XXXX e numero n.

DALLA PROGRAMMAZIONE TRADIZIONALE A QUELLA PARALLELA

G.P. Bottoni , M. Cremonesi, I. De Lotto
CILEA, Segrate (MI) - Università degli Studi, Pavia

RIASSUNTO

L'evoluzione delle tecnologie hardware e software degli elaboratori cambia l'ambiente in cui opera l'analista e il programmatore. Con riferimento al calcolo tecnico-scientifico viene brevemente fatto il quadro dei cambiamenti intercorsi negli ultimi anni e si illustrano i principali problemi incontrati nel programmare i supercalcolatori vettoriali e paralleli.

ABSTRACT

Hardware and software computer technology evolution changes the enviromment in which the programmer and analyst operate.
With reference to the technical and scientific computation changes which took place during the last few years and brief summary is reported and the morin problems faced in the programming of parallel and vectorial supercomputers.

1. INTRODUZIONE

Elaboratori sempre più veloci sono richiesti e messi a disposizione per il calcolo tecnico scientifico nelle più varie aree applicative: dalle nuove discipline "computazionali" quali ad esempio la fisica, la chimica, la biologia, alle applicazioni di ingegneria quali quella strutturale, quella che studia fenomeni di termofluidodinamica o componenti e sistemi elettronici. Le potenze di elaborazione rese disponibili dai calcolatori commerciali attuali si misurano in centinaia di megaflops (milioni di operazioni in virgola mobile al secondo), con un aumento di capacità elaborativa di un fattore almeno 100 negli ultimi dieci anni e con un tasso di crescita attuale di circa un fattore 2 all'anno. Questi progressi sono stati fatti per l'evoluzione delle tecnologie dei componenti, in particolare dei circuiti al silicio con l'integrazione su larga scala, e per le innovazioni di architettura con l'adozione del parallelismo a diversi livelli operativi. La legge empirica di Grosh dice che, nell'ambito di una medesima tecnologia, la velocità di un elaboratore cresce con il quadrato del suo prezzo, il che significa un rapporto prestazioni-costo crescente per l'hardware in modo significativo.

Non è possibile ripetere le stesse considerazioni per il software, dove gli strumenti di sviluppo messi a disposizione e la qualità crescente richiesta per i prodotti non hanno permesso di aumentare la produttività nello sviluppo dei programmi, in media, per più di un fattore 3 nello stesso periodo di tempo; già con macchine puramente scalari e per codici sequenziali si parla di crisi del software, essendo sempre più complesso scrivere, trovare gli errori, correggere e mantenere programmi di grandi dimensioni. Procedendo verso le macchine ad elevate prestazioni di tipo parallelo, questa crisi del software si complica, per la più elevata complessità del processo di programmazione. Questa, in assenza di ambienti di programmazione di livello adeguatamente elevato, sembra fare un salto all'indietro, dovendo il programmatore tener ora conto anche di come scomporre il programma in parti da eseguirsi in parallelo e della architettura della macchi-

na su cui intende lavorare.
In questa breve nota si fanno alcune considerazioni sui problemi ora esposti, richiamando brevemente l'evoluzione degli elaboratori e della loro architettura e le conseguenti implicazioni per il programmatore. Si limita l'analisi ai sistemi per il calcolo scientifico e alle architetture derivate dal modello di Von Neumann (non si considerano quindi le macchine sistoliche, le dataflow machines e le reduction machines).

2. DALL'ARCHITETTURA SERIALE A QUELLA PARALLELA

L'idea di aumentare la potenza elaborativa per un'assegnata tecnologia, ricorrendo all'elaborazione parallela non é nuova. Nel 1920 V. Bush del MIT progettò un calcolatore analogico per risolvere in parallelo equazioni differenziali, nel 1940 Von Neumann dell'Università di Princeton propose una griglia di elaboratori per una soluzione parallela di equazioni differenziali, D. Slotnick dell'Università dell'Illinois progettò nel 1960 l'Illiac IV con 64 processori in parallelo, nel 1980 sono apparse le prime macchine commerciali ad elevato parallelismo (Hypercube, HEP) fino ad arrivare alla Connection Machine nel 1986.

I costruttori di mainframe usano da tempo il parallelismo a diversi livelli dell'architettura per aumentare la potenza elaborativa delle proprie macchine. La riduzione del periodo dell'orologio, legato alla velocità di funzionamento dei componenti elettronici di base, che ha raggiunto in alcune macchine il valore di pochi nanosecondi (CRAY2: tc = 4.1 nsec; Hitachi S820: tc = 4 nsec), non basta più per ottenere prestazioni elevate. Oltre a ciò esiste il limite non ancora raggiunto, ma non così remoto, del tempo di propagazione del segnale fissato dalla velocità finita della luce che dà, per un chip uniprocessore della dimensione lineare di 2.5 cm, una potenza massima dell'ordine di 1 GFLOPS, legata al tempo di attraversamento del chip da parte del segnale. Si usano allora registri veloci sui quali memorizzare temporaneamente gli operandi, rendendo così più breve il tempo di accesso rispetto a quello della memoria ed, estendendo il concetto, si fa ricorso ad una gerarchia delle memorie per ottenere un ottimo funzionale, anche se con un appesantimento del lavoro di gestione. Si adottano unità funzionali multiple, tali da permettere l'esecuzione contemporanea ad esempio di una somma e di un prodotto; si utilizza il prefetching delle istruzioni in modo da sovrapporlo al fetching degli operandi. Si organizza la memoria in banchi indipendenti, con le informazioni distribuite sequenzialmente su di esse in modo da permettere un accesso parallelo a più informazioni consecutive. Si segmentano le unità funzionali, permettendo la sovrapposizione di più operazioni.

Questi interventi si manifestano al programmatore con maggiori prestazioni della macchina, a fronte di una sua più elevata complessità logica e circuitale, ma non gli richiedono particolari attenzioni essendo essi mascherati dai microprogrammi e dal software di base fornito dal costruttore. Nè richiedono ripensamenti degli algoritmi rispetto a quelli per macchine sequenziali più semplici, anche se a volte, per ottenere un'elevata efficienza, si deve tener conto dell'architettura della macchina e degli algoritmi di gestione interna (ad esempio quelli per la gestione della memoria virtuale).

Il salto di qualità nelle prestazioni, ma anche il coinvolgimento dell'utente, si è avuto con l'introduzione commerciale degli elaboratori vettoriali (SIMD) e paralleli (MIMD e multi SIMD). In queste macchine non ci si è limitati a potenziare l'unità aritmetica e la memoria (come nei processori vettoriali), oppure l'unità aritmetica e l'unità logica (come nelle macchine con istruzioni molto lunghe), ma nelle macchine SIMD, pur mantenendo un'unica unità logica, però con tutti gli accorgimenti per renderla veloce, si sono potenziate particolarmente l'unità aritmetica e la memoria, in modo da eseguire la stessa istruzione su più dati in tempi molto brevi; nelle macchine MIMD si sono moltiplicate per n le unità centrali, con riferimento ad un'unica memoria (architetture a memoria condivisa), o gli interi elaboratori, tra loro comunicanti attraverso opportune reti di connessione (architettura con invio di messaggio).

La tecnica correntemente usata per aumentare la capacità elaborativa dell'unità aritmetica é segmentarla in una successione di fasi che operano contemporaneamente su dati diversi (struttura a catena di montaggio (pipeline)). Per caratterizzare questa struttura si può dire che:

- se un'elaborazione può essere decomposta in n fasi e t_q é la durata della fase q-esima, il tempo totale dell'elaborazione risulta:

$$T_t = \sum_{q=1}^{n} t_q \qquad (1)$$

- se il valore massimo di t_q è T_{max} il tempo totale per eseguire K elaborazioni uguali nella struttura pipeline può essere scritto come:

$$T = [T_{max} + \frac{T_t - T_{max}}{k}] K$$

e il tempo medio per elaborazione:

$$T_m = T_{max} + \frac{T_t - T_{max}}{k}$$

Rispetto ad un' esecuzione seriale delle singole fasi, il cui tempo è dato dalla (1) la struttura a pipeline permette un fattore di maggior velocità di calcolo:

$$S = \frac{KT_t}{KT_{max} + (T_t - T_{max})}$$

Nel caso ottimale di $T_{max} = \frac{T_t}{n}$:

$$S = \frac{n}{1 + \frac{n-1}{K}}$$

che tanto più approssima n quanto più elevato è K, il numero di operazioni uguali che si eseguono in serie (in genere K > n).

Se invece la struttura pipeline non è bilanciata, cioè se ad esempio $T_{max} = T_t / 2$

$$S = 2 - \frac{2}{K+1}$$

che al più può essere uguale a due. Una struttura pipeline quindi deve essere bilanciata ed elevato deve essere il numero di operazioni uguali da eseguire. Un' operazione vettoriale, nel gergo di chi lavora con gli elaboratori vettoriali, consiste nell'esecuzione della stessa istruzione su un insieme di dati. Per i calcolatori vettoriali, al fine di ottenere le migliori prestazioni, è quindi necessario caricare l'unità aritmetica con il maggior numero possibile di operazioni vettoriali per le quali l'unità è specializzata e in genere dà prestazioni molto più elevate che nel caso di operazioni scalari (un'istruzione per ogni coppia di dati) e con operazioni vettoriali operanti ciascuna su un elevato numero di dati.

Come succede in ogni fenomeno che in parte si sviluppi alla velocità v_1, e in parte alla velocità v_2, la velocità media v è data da:

$$\frac{1}{v} = \frac{f}{v_1} + \frac{1-f}{v_2}$$

dove f è la frazione del fenomeno a velocità v_1; così per un programma che sia eseguito in parte in modo scalare (f) e in parte in modo vettoriale (1-f), il fattore di accelerazione S_0 per l'utilizzo di un elaboratore vettoriale diviene:

$$S_0 = \frac{T_{(f=1)}}{T_{(0 \le f \le 1)}} = \frac{1}{f + \frac{(1-f)}{S_1}} \qquad (2)$$

dove S_1 è il rapporto tra tempo di esecuzione scalare e tempo di esecuzione vettoriale di un'istruzione. S_1 coincide con S se la stessa unità è usata per i due modi operativi.

Questa relazione mette in evidenza il peso delle parti del programma da eseguirsi in modo scalare, tanto maggiore quanto più elevato è S_1, e in conseguenza l'opportunità che il programma sia vettorizzato al meglio.

L'espressione (2) può essere estesa al caso in cui si valuti il fattore di accelerazione per un programma, una cui parte (f) è eseguita su un solo processore perchè non parallelizzabile e una parte (1 - f) eseguita in parallelo su p processori. In tal caso $v_2 = v_1/p$ e quindi:

$$S_0 = \frac{1}{f + \frac{(1-f)}{p}} \qquad (3)$$

Più in generale, per un programma una cui frazione f_0 non è parallelizzabile e le frazioni f_i tra loro disgiunte sono parallelizzabili su p_i processori, detto O_i l'overhead di sincronizzazione per la frazione i, il fattore di accelerazione diviene:

$$S_0 = \frac{1}{f_0 + \sum_1^m {}_i (\frac{f_i}{p_i} + O_i)} \qquad (3^I)$$

dove m è il numero di frazioni disgiunte parallelizzabili del programma e inoltre:

$$f_0 + \sum_1^m {}_i f_i = 1$$

L'espressione (3) mostra che $S_0 \le p$. Si può allora scrivere: S_0 (1 - a)p, con $0 \le a < 1$. Essendo dalla (3): $S_0 < 1/f$ si ottiene: a $\ge$ 1- 1/fp, che dice che per un assegnato f, a è tanto più elevato, cioé tanto minore é l'efficienza, quanto più grande é p.

Si consideri ad esempio il caso di un algoritmo di moltiplicazione matriciale. Esso è composto da una parte sequenziale per leggere le matrici dalla memoria, che è lineare con l'ordine N della matrice, e da una parte per eseguire il prodotto, che è proporzionale a N^3 In conseguenza: $f(N) = r/N^2$ dove r è l'operazione di lettura misurata in passi di moltiplicazione matriciale. Dall'eq. 3 si ottiene:

$$(1 - a)p = \frac{1}{f(N) + \frac{(1-f(N))}{p}}$$

dove

$$a = \frac{1}{1 + \frac{N^3}{r(p-1)}}$$

cioè più N è grande, più è elevata l'efficienza a pari numero di processori. In altri termini i vantaggi più

significativi si ottengono in presenza di calcoli onerosi, e cioè, per garantire una assegnata a:

$$N > \sqrt{\frac{rp}{a}}$$

Il programma non solo deve permettere l'individuazione delle fasi disgiunte parallelizzabili, ma anche esser scritto per trovare la più elevata efficienza possibile per ciascuna fase parallelizzabile.
Gli esempi portati sono elementari, ma mettono in evidenza come le architetture vettoriali e quelle parallele coinvolgono il programmatore assai più delle altre innovazioni architetturali, richiedendogli in generale una conoscenza dell'elaboratore su cui opera approfondita e specialistica.

3. PROGRAMMAZIONE VETTORIALE

Come si è già detto, nel seguito si fa riferimento solo al calcolo scientifico. Questo è caratterizzato in genere da un'elevata percentuale di operazioni in virgola mobile e da programmi usualmente codificati in Fortran. Si fa inoltre riferimento a macchine general purpose, adatte cioè per una discreta generalità di problemi.

Una macchina vettoriale è di solito basata su una macchina seriale convenzionale con memoria a banchi intercalati (interleaved memory) e un processore aritmetico veloce a pipeline. La programmazione di questi sistemi viene fatta ricorrendo o a compilatori in grado di vettorizzare automaticamente il codice sorgente, o, come succede nei casi meno evoluti e sofisticati, ad un insieme di sottoprogrammi specializzati di libreria. Nella maggior parte dei casi é fornita una versione estesa del Fortran dove sono disponibili alcune istruzioni vettoriali per un uso diretto delle possibilità offerte dalla macchina. Tali estensioni però non obbediscono a standard riconosciuti e rendono in genere non portabile il programma tra macchine diverse.

Le conoscenze dell'architettura della macchina vettoriale richieste al programmatore riguardano soprattutto:

- il tipo di operazione vettorizzabile o che, per un'assegnata macchina, convenga vettorizzare. Ad esempio non si vettorizzano le operazioni su caratteri oppure in alcune macchine non conviene vettorizzare operazio ni su interi o quelle in virgola mobile in doppia o quadruplice precisione perché poco efficienti;

- la concatenabilità delle operazioni (chaining) legata alla disponibilità di unità funzionali che permettano di eseguire in cascata operazioni senza ricorrere ad accessi alla memoria;

- l'accesso alle memorie, in funzione del numero dei banchi, di come sono intercalati (interleaving), del numero e delle dimensioni dei registri, dell'opportunità di indirizzamento con passo non unitario, della disponibilità di hardware per far riferimento ad elementi di vettori indirizzati in maniera indiretta (operazioni di gather e di scatter);

- presenza di una memoria cache, sua dimensione e tempi di caricamento;

- caratteristiche dell'unità aritmetica con pipeline, quali lunghezza dei registri, durata della inizializzazione della pipeline, modalità di utilizzo dei registri vetto riali in caso di duplice precisione;

- disponibilità di operazioni di riduzione hardware, quali ad esempio sommatoria, produttoria, ricerca del minimo o del massimo di un vettore.

Il programmare un calcolatore vettoriale usando il Fortran richiede la conoscenza delle prestazioni funzionali del compilatore e dell'eventuale precompilatore per vettorizzare il più possibile il programma sorgente. La vettorizzazione può semplicemente definirsi come la generazione di un codice che tratti gli elementi di vettori in maniera sovrapposta nelle unità funzionali pipeline e sfrutti i parallelismi della macchina in modo ottimale. La vettorizzazione riguarda soprattutto i DO loops e il programmatore con appropriate strutturazioni del programma può permettere una significativa vettorizzazione, altrimenti meno efficiente se eseguita unicamente con il compilatore.

Con riferimento al 1100/90 ISP Unisys del CILEA e al relativo compilatore UFTN, i passi principali del processo di vettorizzazione automatica sono i seguenti:

a: Analisi di flusso, suddivisione in blocchi logici e raccolta delle informazioni: questo passo consiste nel riconoscimento dei cicli di DO quali strutture a cui applicare l'analisi di vettorizzazione e nel primo ordinamento dei legami tra le variabili sia scalari sia vettoriali presenti all'interno dei blocchi stessi.

b: Normalizzazione dei cicli di DO: per semplificare l'analisi svolta nei passi seguenti, ogni ciclo di DO viene ridefinito in modo che il nuovo indice del DO abbia sempre lo stesso valore iniziale e lo stesso valore di incremento (tipicamente parta da 1 ed abbia incremento unitario).

c: Estensione dell'uso della variabile del DO: viene attuata una sistematica eliminazione di tutte le variabili che dipendono linearmente dal valore dell'indice del DO in

modo da evidenziare tutte le situazioni in cui si verifichino accessi alla memoria con passo (stride) costante, al fine di ottimizzare l'accesso alla memoria interessata.

d: Analisi di dipendenza: viene presa in considerazione ogni possibile relazione tra i valori assegnati alle variabili sia scalari sia elementi di vettori e l'uso delle variabili stesse quali termini di altre espressioni all'interno del blocco di DO considerato. La vettorizzabilità di una particolare istruzione è infatti possibile solo se il nuovo valore di un dato elemento di un vettore non è una condizione necessaria per il calcolo di un altro elemento del medesimo vettore nel corso dell'attuazione della stessa istruzione in uno dei passi successivi del ciclo di DO.

e: Riconoscimento degli scalari indipendenti: nell'ambito di uno stesso ciclo di DO è frequente l'impiego di scalari utilizzati come deposito provvisorio di valori parziali da includere in più di una espressione. Spesso gli scalari usati per questi scopi ricevono diverse assegnazioni nel corso di uno stesso ciclo di DO e questo viene fatto sostanzialmente per minimizzare l'occupazione di memoria o per ridurre al minimo il numero di variabili utilizzate.
Senza il riconoscimento delle funzioni svolte da due successive assegnazioni di uno stesso scalare usato per questi scopi il blocco di istruzioni comprese tra due successive assegnazioni dovrebbe essere considerato sempre a monte delle istruzioni che seguono la seconda assegnazione mentre in realtà puo esserci del parallelismo se le due assegnazioni non sono tra loro logicamente dipendenti. Il compilatore svolge quindi una ricerca di questi scalari ed eventualmente attua il renaming ossia li considera sdoppiati per evitare che uno stesso scalare svolga funzioni logicamente diverse.

f: Espansione degli scalari: il valore assunto da una grandezza scalare al variare dell'indice del DO potrebbe essere memorizzato non in uno stesso indirizzo di memoria o conservato in uno stesso registro scalare bensì , ad esempio, salvato in un vettore il cui i-esimo elemento conterrebbe il valore assunto dallo scalare all'i-esimo passo del ciclo di DO. A fronte del costo rappresentato dalla necessità di disporre di una maggiore area di memoria sussisterebbe il vantaggio di potere trattare ad alta velocità i dati così memorizzati.

Esistono casi in cui risulterebbe possibile conservare i valori temporanei dello scalare non in memoria bensì direttamente in uno dei registri vettoriali dell'ISP (sono in tutto 16, ciascuno capace di contenere 64 parole in semplice precisione o 32 in duplice). L'efficacia computazionale di questa promozione a vettore, altrimenti detta espansione degli scalari, dipende da molte circostanze tra cui, ad esempio, appunto l'effettiva disponibilità di un registro vettoriale per l'accumulo dei valori temporanei. L'espansione degli scalari, previa valutazione della sua efficacia, viene attuata automaticamente da UFTN in uno specifico passo del processo di vettorizzazione.

g: Suddivisione dei DO: spesso l'eccessiva complicazione dei cicli di DO costituisce un ostacolo alla loro vettorizzazione. La presenza di istruzioni non vettorizzabili causerebbe la non vettorizzazione dell'intero ciclo di DO se il compilatore non fosse in grado di attuare una suddivisione automatica del ciclo in modo da isolare, se possibile, in blocchi separati, le istruzioni non vettorizzabili.

h: Scambio dell'ordine di esecuzione dei DO: si tratta di un intervento per ora soltanto previsto per il futuro UFTN, ma che è già effettuato da altri compilatori vettorizzatori quali il FORTVS2 installato sull'IBM 3090 VF del CILEA. Si possono infatti verificare situazioni in cui è possibile scambiare l'ordine di effettuazione dei cicli di DO annidati facendo in modo che il ciclo di DO esterno diventi il piu interno e quest'ultimo diventi il ciclo esterno. Il riconoscimento della possibilità di effettuare scambi di questo tipo nonchè il giudizio sull'opportunità di effettuarli per migliorare le prestazioni sono compiti che, anche se ardui, possono venire gestiti automaticamente da un compilatore della classe di UFTN.

i: Riconoscimento di costrutti speciali: questa classe di interventi costituisce un punto di forza di UFTN perché comprende azioni la cui effettuazione sarebbe da un lato inevitabile per non rinunciare a soddisfacenti livelli di prestazioni, ma dall'altro costosa dal punto di vista dell'impegno di ristrutturazione del simbolico. Il compilatore, in pratica, è in grado di riconoscere alcuni fra i più frequenti kernel in via di principio non vettorizzabili quali la sommatoria degli elementi di un vettore, la produttoria, il prodotto scalare, la ricerca dell'elemento minimo o massimo, la ricorsione lineare del primo ordine e di applicare automaticamente algoritmi alternativi ai classici comunemente utilizzati e, a volte, tali da consentire anche elevatissimi livelli di prestazioni vettoriali.

l: Ottimizzazioni dipendenti dall'hardware: si tratta di interventi di riordino della sequenza di istruzioni vettorizzabili o di sfruttamento della possibilità della CPU dell'ISP di effettuare in parallelo calcoli scalari e calcoli vettoriali. La gestione di queste trasformazioni è naturalmente un compito del compilatore perchè ove fosse anche parzialmente consentita dal linguaggio (istruzioni Fortran non standard) provocherebbe problemi di portabilità dato che certi "trucchi" potrebbero rivelarsi controproducenti su una macchina diversa dall'ISP.

I passi sopra elencati, anche se riferiti al 1100/90 ISP della Unisys, sono tipici della maggior parte dei compilatori per macchine vettoriali di grande dimensione (CDC Cyber 205, CRAY X-MP, CRAY-2, ETA10, IBM 3090-VF, Amdahl 1200/1400, Hitachi S-810/820, NEC SX/2) e di quelle di medie dimensioni (Alliant FX/8, Convex C, Gould NP1, Intel iPSC-VX, SCS-40, Stellar GS 100).

Quasi per ognuno dei passi che si sono qui elencati esistono situazioni in cui è opportuno l'intervento diretto del programmatore.

Il più delle volte si tratta di integrare le informazioni desumibili automaticamente dal sorgente con altre, di rilievo solo per la vettorizzazione, in mancanza delle quali non è possibile decidere sulla vettorizzabilità di una determinata istruzione.

Ripassando in rassegna i vari passi, il programmatore potrà innanzi tutto cercare di rendere semplice e chiara, per quanto possibile, la gestione di ogni indice di ciclo di DO, evitando l'uso di indici non interi e di espressioni complesse per definirne l'inizio, la fine ed il passo (punto b).

Potrà inoltre cercare di non ricorrere ad interi non facilmente correlabili all'indice del DO in modo che gli interventi previsti al punto c) siano applicabili, se possibile, alla totalità degli interi usati come indici di vettori o nell'ambito di espressioni di assegnazione agli elementi dei vettori.

L'analisi di dipendenza dei dati (punto d) è una fase cruciale per la vettorizzazione e merita qualche esemplificazione; la regola generale è di scrivere cicli di DO in cui nessuno dei vettori cui vengono assegnati nuovi valori venga contemporaneamente usato anche a destra dell'assegnazione.

Il punto e) fornisce lo spunto per sottolineare quanto possa risultare negativa, per la vettorizzazione la cura che il programmatore può porre per contenere lo spazio di memoria necessario, data la disponibilità di grandi capacità di memoria. E' perciò buona pratica non adibire uno stesso scalare a più di una funzione all'interno di uno stesso ciclo di DO e soprattutto cercare attentamente di non utilizzare un elemento di vettore come se fosse un semplice scalare; si evita così di far risolvere al compilatore inutili problemi di dipendenza tra i dati derivanti dalla possibilità che la modifica dei valori di altri elementi del vettore comporti la modifica dell'elemento del vettore usato unicamente come deposito temporaneo di valori intermedi.

Il passo f) ribadisce quanto detto per il passo e) sulla opportunità di non lesinare troppo sull'utilizzo di aree di memoria anche estese come accumulo di valori temporanei. Va ricordato che la nuova possibilità offerta da UFTN di allocazione e rilascio dinamico di aree di memoria consente una estrema flessibilità nell'uso di vettori temporanei. In genere quindi il programmatore è invitato ad abbondare nell'uso di vettori temporanei ottenuti dalla espansione di scalari eventualmente non promossi già automaticamente da UFTN.

L'azione svolta al passo g) è frequentemente ostacolata dalla impossibilità del compilatore di valutare gli effetti collaterali di funzioni e subroutines usate all'interno dei cicli di DO. L'intervento diretto del programmatore riguarda molto spesso situazioni di questo genere.

Gli interventi previsti al punto h) sono mirati a migliorare le prestazioni del programma, ma il non farli non impedisce la vettorizzazione.

Per quanto riguarda l'ultimo passo (punto i) va tenuto presente che l'identificazione automatica di costrutti speciali è una operazione delicata che deve essere agevolata, per quanto possibile, all'atto della stesura del simbolico. Qualora non sussistano problemi di portabilità, è quindi consigliabile l'uso delle funzioni intrinseche non standard offerte da UFTN appunto per gestire, ad esempio, operazioni di riduzione o di ricerca di indici o operazioni su matrici. Le funzioni citate non sono standard rispetto al Fortran 77, ma i vantaggi di un loro utilizzo sono notevoli e vanno confrontati con lo svantaggio di non realizzare un simbolico portabile.

Oltre a questi interventi miranti soprattutto a ottenere un programma più vettorizzabile, il programmatore può operare in modo da migliorare le prestazioni della macchina. Sempre con riferimento al 1100/90 ISP dell'Unisys, ma analoghe considerazioni valgono anche per le altre macchine, l'operatore può:

1) intervenire per eliminare gli eventuali conflitti di accesso ai banchi della memoria intercalata.
Questo problema trae origine dal fatto che il tempo di calcolo di un'unità funzionale è, a regime, nettamente inferiore al tempo di accesso alla memoria. Per ovviare a questo inconveniente la memoria viene suddivisa in banchi (nell'ISP ogni banco è costituito da 512 Kparole) e gli elementi contigui di uno stesso vettore vengono memorizzati in modo da trovarsi distribuiti su banchi diversi. Le operazioni di trasferimento dati tra i registri vettoriali e la memoria viene effettuata in parallelo tra i banchi (otto per l'ISP) e la CPU , in modo che la maggiore lentezza di

accesso alla memoria venga compensata dal parallelismo10 nel trasferimento. In pratica da ogni banco vengono prelevate quattro parole per volta e quindi gli elementi A(1)...A(4) di un vettore A(:) sono memorizzati su un dato banco seguiti dagli elementi A(33)...A(36) a loro volta seguiti dagli elementi A(65)...A(68) e così via. Se, a livello di Fortran, si richiede la disponibilità immediatamente consecutiva degli elementi A(1), A(33), A(65) e così via, il meccanismo di accesso parallelo ai banchi è reso inefficace e la velocità di calcolo è rallentata di un fattore molto prossimo ad otto a causa della non disponibilità tempestiva dei dati su cui effettuare l'operazione.

E'quindi essenziale, al fine di una ottimazione delle prestazioni, ricorrere ad un passo di scansione degli elementi dei vettori (stride) per quanto possibile unitario.

2) usare correttamente le tecniche di unrolling.
La tecnica dell'unrolling consente di ottenere in Fortran prestazioni prossime a quelle ottenibili programmando direttamente in linguaggio assember; è quindi consigliabile nelle situazioni più onerose dal punto di vista computazionale. Essa consiste nello sfruttamento della capacità della CPU di operare in parallelo, almeno parzialmente, sulle varie unità funzionali di cui è costituita. Questa capacità è evidente ad esempio nel seguente ciclo di DO:

```
      DO 10 J = 1, N
         A (J) = A (J)+B (J, K)
10    CONTINUE
```

Il tempo di calcolo, per N sufficientemente grande, è dell'ordine di quello richiesto per effettuare il seguente calcolo:

```
      DO 11 J = 1, N
         A (J) = A (J)+C (K) * B (J, K)
11    CONTINUE
```

dove, come si vede, il numero di operazioni effettuate è il doppio. Questo effetto, detto di "chaining" o concatenamento delle unità funzionali, si verifica pure, ad esempio, nel caso del calcolo del seguente ciclo di DO:

```
      DO 12 J = 1, N
         A (J) = A (J)+C (K) * B (J, K)+E (K) * D (J,K)
12    CONTINUE
```

che richiede un tempo di calcolo inferiore al doppio del tempo richiesto per l'effettuazione del precedente ciclo di DO. Se ora si identifica E(K) con C(K+1) e D(J, K) con B(J, K+1) si vede subito che il precedente sorgente può essere interpretato come il nucleo interno del calcolo del prodotto della matrice B per il vettore C; e da questa osservazione deriva che l'indice K debba essere incrementato di due unità passo dopo passo: si deve cioé effettuare un "unrolling" di ordine due sul DO usato per incrementare K. Il procedimento può essere generalizzato dato che si verifica che, ad esempio, le sei operazioni in virgola mobile richieste dal calcolo di :

```
      DO 13 J = 1, N
         A (J) = A (J)+B (J, K) * C (K)+B (J, K+1) * C (K+1)+B
         (J,K+2) * C (K+2)
13    CONTINUE
```

vengono effettuate in meno dei 3/2 del tempo richiesto dal caso precedente. I vantaggi dovuti alla tecnica dell'unrolling si fanno, tuttavia, sempre meno sensibili al crescere del livello dell'unrolling fino ad essere non solo nulli ma addirittura negativi. Per una data CPU vettoriale esiste, quindi, un livello di unrolling ottimale. Per l'ISP tale livello è risultato essere attorno a 4 per prodotti matrice vettore di piccole dimensioni, mentre è 8 per dimensioni sopra l'ordine 100. Un livello di 16 di unrolling risulta viceversa controproducente.

3) provvedere ad eliminare la ricorsione lineare.
In genere i compilatori e i precompilatori sono provvisti di tecniche per rilevare casi di ricorsione e per modificare il codice per eliminarli. Data la sua importanza, tuttavia, val la pena di esaminarla brevemente per comprendere come il programmatore possa utilmente intervenire in questi casi.

La ricorsione lineare del primo ordine si presenta sotto la seguente semplice forma :

```
      DO 20 J = 1, N
         X (J) = A (J)+B (J) * X (J-i)
20    CONTINUE
```

con X (0) utilizzato solo come costante. Si parlerebbe, viceversa, di ricorsione lineare del secondo ordine se si avesse a che fare con il seguente nucleo:

```
      DO 21 J = 2, N
         X (J) = A (J)+B (J) * X (J-l)+C (J) * X (J-2)
21    CONTINUE
```

dove fungono da costanti sia X (0) che X (1). Poichè esiste un effettivo legame di dipendenza tra i dati, cioè per ogni valore di J il risultato memorizzato in X (J) dipende da quanto è accaduto al passo J-1, la ricorsione non è, a rigore, vettorizzabile e ciò costituisce una grave limitazione operativa, perchè essa si presenta frequentemente nel calcolo matriciale. Per ottenere la vettorizzabilità si hanno a disposizione due vie: quella, evidente ma non facile, di riformulare l'algoritmo in modo da non utilizzare la ricorsione, oppure quella di adottare un algoritmo che

permetta, nonostante tutto, di trattarla in forma vettoriale. Un esempio di algoritmo che consente di vettorizzare la ricorsione lineare del primo ordine è il seguente, detto di riduzione ciclica, che è applicabile se N è una potenza di 2 (ossia N = 2 ** IS):

```
      K = 1
      DO 10 J = 1, IS
         M = K
         K = K * 2
         A (K : N : K) = A (K : N : K)+B (K : N : K)*A (M : N-M : K)
         B (K : N : K) = B (K : N : K)*B (M : N-M : K)
10    CONTINUE
      M = K* 2
      DO 20 J = 0, IS
      K = M
      M = M/2
      X (M : N : K) = A (M : N : K)+B(M : N : K) *X (0 : N-M : K)
20    CONTINUE
```

dove si è utilizzata la concisa sintassi del Fortran vettoriale: ad esempio, X (M : N : K) =.... corrisponde all'effettuazione di un ciclo di DO avente M come primo valore dell'indice, N come estremo superiore e K come incremento di ciclo.
In pratica si procede calcolando, in forma vettorizzabile, dei nuovi coefficienti tratti da quelli originari dei vettori A(:) e B(:); quindi si effettuano IS + 1 passi tutti vettorizzati per ottenere i valori del vettore X(:). Il numero di operazioni complessive risulta essere superiore di un fattore 2.5 a quello richiesto dall'algoritmo originario, ma tali operazioni sono effettuabili con prestazioni ampiamente superiori a 2.5 da parte della C.P.U. vettoriale. Si noti, inoltre, che, nella forma riportata, utile solo per fini espositivi, l'algoritmo comporta un accesso agli elementi di A(:7), B(:) ed X(:) con stride diverso di volta in volta al variare di IS, e comunque pari ad una qualche potenza di 2; quando lo stride risulterà essere un multiplo di 32 si avranno quindi prestazioni degradate a causa del conflitto di accesso ai banchi della memoria, di cui si è già parlato.

E' evidente che un normale programmatore scientifico non possa, non conoscendolo, reinventare un algoritmo della complessità di quello riportato per vettorizzare la ricorsione lineare del primo ordine. Risulterebbero perciò del tutto non vettorizzabili algoritmi classici come, ad esempio, quello per la soluzione di sistemi tridiagonali o l'algoritmo di Gaus-Seidel applicato frequentemente nella soluzione di sistemi di equazioni derivanti dalla discretizzazione di equazioni differenziali con il metodo delle differenze finite. E' quindi da ritenersi una caratteristica di rilievo dei compilatori vettoriali quella di riconoscere e gestire la ricorsione lineare adottando automaticamente un algoritmo capace di consentirne la vettorizzazione.

Naturalmente l'intervento manuale per eliminare la ricorsione, qualora non comporti uno sforzo di ristrutturazione troppo elevato, è comunque consigliabile.

Si consideri il seguente caso

```
      DO 15 J = 1, M
         DO 25 K = 2, N
            A(K, J) = C *A (K-I, J)+B (K, J)
25       CONTINUE
15    CONTINUE
```

Trattandosi, appunto, di un caso di ricorsione lineare vettorizzabile automaticamente il risparmio di tempo di calcolo passando da macchina scalare a macchina vettoriale sarebbe comunque sensibile; ma si potrebbero ottenere prestazioni migliori semplicemente scambiando tra loro i due DO nel seguente modo :

```
      DO 25 X = 2, N
         DO 15 J = 1, N
            A (K, J) = C * A (K-1, J)+B (K, J)
15       CONTINUE
25    CONTINUE
```

Va osservato che il solo difetto di questa versione risiede nel fatto che si accede alle matrici A(:,:) e B(:,:) facendo variare più velocemente il secondo indice e quindi con stride non unitario. Se la prima dimensione di tali matrici fosse un multiplo di 32, probabilmente si scoprirebbe che la versione originaria risulta, nonostante la ricorsione, competitiva o anche migliore di quest'ultima intrinsecamente più vettorizzabile dell'altra. Questa osservazione suggerisce l'importanza della verifica continua dell'efficacia degli interventi manuali fatti per migliorare le prestazioni vettoriali. Interventi apparentemente vantaggiosi, possono risultare in un passo indietro rispetto all'uso della versione originaria.

Analoghe considerazioni potrebbero farsi nel caso della promozione di variabili temporanee a vettore, degli interi ad incremento costante, dei vettori di dimensione limitata. Chi desidera maggiori informazioni, può fare riferimento ai richiami bibliografici [1-13]

Questa esposizione mette in evidenza da un lato la complessità dei problemi che il compilatore deve affrontare, dall'altro lato il lavoro che il programmatore è chiamato a fare per ottenere il miglior sfruttamento della macchina, con un attento lavoro di valutazione del costo della programmazione o della ristrutturazione di un codice scritto per una macchina scalare a fronte delle prestazioni ottenibili su una assegnata macchina vettoriale, della portabilità del sorgente su altre macchine, della leggibilità del sorgente, della sua manutenibilità e riusabilità, fattori questi molto importanti per codici di lunga vita. Questo lavoro, intrinsecamente complesso, è reso ancor più arduo dalla mancanza di standard accettati da tutti i principali costruttori per un'estensione del Fortran al calcolo vettoriale.

4. PROGRAMMAZIONE PARALLELA

Le macchine ad architettura parallela per quanto riguarda quegli aspetti che influenzano il modo di programmare, possono classificarsi in tre grandi famiglie:

a) macchine a memoria condivisa, dotate di una sola grande memoria a cui tutti i processori accedono. Affluiscono a questa famiglia le architetture a bus e quelle a rete di interconnessione ad esempio multistadio.

Nelle macchine a bus, i processori, i moduli di memoria e le unità di ingresso e uscita sono connesse ad uno o più bus di comunicazione ad elevata capacità; tutti i moduli di memoria sono accessibili a tutti i processori. La capacità trasmissiva del bus comune limita il numero di processori che possono essere connessi senza problemi di contesa nell'uso del bus: un massimo tipico è di 12 processori. Si può migliorare questa prestazione aggiungendo ad ogni processore una memoria cache; questa operazione tuttavia, per restare nel modello di macchina a memoria condivisa, richiede un hardware dedicato alla sincronizzazione dei contenuti delle cache.

Una soluzione alternativa è l'utilizzo di una rete di interconnessione, ad esempio multistadio, che permetta la connessione tra processori e memorie. E' una soluzione più complessa circuitalmente, ma permette più connessioni contemporanee.

A questa famiglia appartengono i supercalcolatori CRAY-XMP, IBM 3090, Unisys 1100/90 e successivi modelli, e, tra i mini supercalcolatori l'Alliant FX/8, il Convex C, la NYU Ultracomputer e le macchine della Sequent, della Encore, della Flexible, della Elxsi. Va tuttavia precisato che le due famiglie, a memoria condivisa e a invio di messaggio, rappresentano due estremi di un continuo di soluzioni, soprattutto per i supercalcolatori. La loro afferenza alla famiglia con memoria condivisa, corrisponde in genere ad una semplificazione della loro architettura.

b) macchine con invio di messaggi, dove ogni processore ha una memoria propria edè connessa ad un certo numero di altri processori. Nessuna delle memorie locali è globalmente accessibile. Una connessione possibile è quella a ipercubo, dove il numero di connessioni è detta la dimensione dell'ipercubo. Ad esempio un ipercubo 3 D ha otto processori, ciascuno con tre connessioni. Rientrano in questa famiglia le macchine realizate dalla INTEL, dalla NCube, dalla Ametek, dalla FPS. Sono macchine con un numero di processori compreso tra 16 e 1024.

Un'altra connessione possibile è quella a stella dove il processore al centro è sostanzialmente dedicato a gestire il traffico tra gli altri processori ed è quindi funzionalmente diverso.

c) macchine con architettura ibrida, con memorie locali, ma un sistema operativo che fa si che la macchina le veda come una memoria condivisa. In queste macchine il programmatore opera come su quelle a memoria con divisa, ma le elevate prestazioni si ottengono con metodi analoghi a quelli usati per le macchine con invio di messaggio.

Il programmatore che debba operare con macchine parallele non vede tuttavia l'architettura direttamente, ma le funzioni e il tipo di parallelismo offertigli dagli strumenti software che gli sono messi a disposizione.
Sotto questo aspetto l'ambiente a invio di messaggio mette a disposizione sostanzialmente due funzioni aggiuntive a quelle tipiche per macchine scalari: SEND e RECEIVE.

SEND è usato per mandare un messaggio da un processore ad un altro. Il SEND si presenta in due forme: SEND non bloccato il quale permette la prosecuzione dell'elaborazione appena inviato il messaggio; SEND bloccato che invece fa attendere che il messaggio spedito sia arrivato. Gli argomenti del SEND sono: la destinazione, la lunghezza del messaggio e un vettore contenente il messaggio. Possono esser presenti anche una parola di stato, informazioni per l'indirizzamento e un flag per indicare se attendere oppure no la conferma della ricezione.

RECEIVE è usato per leggere un messaggio mandato da un altro processore. Anche esso può essere non bloccato o bloccato. I suoi argomenti sono: un vettore per contenere il messaggio, la lunghezza di questo vettore, eventualmente l'identificatore del mittente, un flag di stato e l'indicazione se inviare oppure no un messaggio di avvenuta ricezione. Il RECEIVE bloccato è usato quando un algoritmo su un processore richiede informazioni da un altro processore e attende finchè i dati arrivino. Va tenuto presente che tale costrutto può dar luogo ad una condizio-

ne di dead lock, quando ad esempio due processori attendono dati l'uno dall'altro. Il RECEIVE non bloccato è usato quando un processore attende dati su vari ingressi il cui ordine di lettura non è importante, oppure quando legge ripetutamente un ingresso in attesa del dato. Un altro uso di questo costrutto è per gestire un ingresso asincrono. Il programma testa l'ingresso con continuità: in assenza di messaggio viene svolto un assegnato lavoro diverso. Un tale costrutto può esser usato ad esempio per bilanciare il carico tra i processori.
L'ambiente a memoria condivisa richiede in generale un numero di estensioni al linguaggio per una macchina monoprocessore, più elevato dell'ambiente ad invio di messaggio. Anzitutto vanno indicati i dati globali, disponibili per tutti i processori, e i dati privati di ciascun processore. Inoltre, per i dati in comune, è necessario sincronizzarne gli accessi per garantire il loro corretto utilizzo e aggiornamento. In questo ambiente gli stili di programmazione possono classificarsi in due grandi famiglie: lo stile FORK-JOIN dove un processo genera più processi (FORK) e attende che tutti siano terminati (JOIN); lo stile SPMD (single program multi data) dove su ciascun processore funziona lo stesso programma, ma viene eseguito un pezzo diverso di codice a seconda dell'identificatore del processore o dei dati nella memoria comune.

In ambo gli stili è necessario poter isolare sezioni critiche del programma da eseguirsi un processore alla volta; tale possibilità è usata soprattutto per le operazioni di riduzione finora accennate, come la somma degli elementi di un vettore in una variabile globale. E' necessario inoltre poter individuare sezioni seriali, da eseguirsi da un solo processore, utile ad esempio per inizializzare dati globali.

Le operazioni di sincronizzazione, necessarie anche in questo ambiente, sono automatiche nello stile FORK.JOIN (istruzioni di JOIN), richiedono un apposito costrutto nelle operazioni SPMD (una barriera o il costrutto WAIT UNTIL). La barriera è un punto del codice dove tutti i processori aspettano che l'ultimo arrivi, il costrutto WAIT UNTIL inpone a ciascun processore di interrogare la memoria comune per controllare se una certa condizione si è verificata.

Il costrutto più noto è il PARALLEL DO con il quale il programmatore parallelizza il codice a livello di un DO-loop. Questo costrutto può essere presente in due versioni: quello autoschedulato con il quale ad esempio i sottoinsieme indipendenti delle iterazioni del DO-loop sono attribuiti nell'ordine ai processori disponibili e quando un processore ha terminato il proprio compito automaticamente gli viene assegnato un nuovo sottoinsieme. In questo modo si ottiene un bilanciamento automatico del carico, ma si forza una certa sincronizzazione tra i vari processori, non necessaria in generale. Il PARALLEL DO preschedulato permette al programmatore di suddividere il DO-loop a priori e di assegnarne una parte a ciascun processore. E' cura del programmatore bilanciare il carico tra i vari processori. La scelta tra l'uno o l'altro dei due costrutti dipende dalle caratteristiche del programma trattato e dall'hardware disponibile; convenendo il costrutto preschedulato in presenza di un bilanciamento del carico intrinseco al problema e di un costo elevato della sincronizzazione, convenendo invece negli altri casi il costrutto autoschedulato.

Di fronte a questi ambienti, il programmatore è chiamato a mappare il problema sull'architettura parallela disponibile. Le fasi di questo processo possono così schematizzarsi: divisione del problema in segmenti da eseguirsi in parallelo, definizione delle modalità di comunicazione tra i processori e infine delle modalità di sincronizzazione.

Un programma può presentare diversi livelli di parallelismo. Un modo per caratterizzarli é di valutarne la granularità, cioè quanto calcolo può fare un processore indipendentemente dagli altri raffrontato al tempo da spendere per comunicare con gli altri processori. Si è soliti parlare di macrotask e di microtask, ai quali si associano diverse modalità operative e di programmazione. Un esempio può venire dai sistemi per la simulazione delle condizioni metereologiche dove un modello a grana grossa permette che ogni singolo processore tratti volumi atmosferici di centinaia di migliaia di chilometri cubi con una limitata attività di comunicazione tra processori, cioè con un elevato rapporto tra tempi di calcolo e tempi di comunicazione. Un tale modello si presta bene per un sistema costituito da pochi processori potenti interconnessi con canali a limitata capacità. Un modello invece a grana fine può comportare che ciascun processore tratti pochi chilometri cubi, gestendo quindi non molti dati, ma con frequenti scambi di informazione con i processori adiacenti (secondo la topologia del modello). In tal caso è più adatta una architettura con molti processori con limitata memoria ed una elevata capacità di comunicazione. In conclusione, la granularità del programma dipende, cioè, sia dal fenomeno studiato e dal relativo modello di simulazione usato, sia dall'architettura della macchina a disposizione.

Definita la granularità del programma, va scelta la modalità di comunicazione tra processori, che, per quanto visto prima, è legata all'architettura della macchina, se cioè è a memoria condivisa oppure a invio di messaggio. Esistono tuttavia problemi che meglio si adattano ad architetture a memoria condivisa (ad esempio la simulazione di una rete sincrona, dove le attività sono sincronizzate da un segnale di orologio simulato) e altri che si prestano meglio per

architetture ad invio di messaggio (ad esempio la simulazione di una rete asincrona, dove ciascun componente cambia il proprio stato interno a seguito del cambiamento dei suoi segnali di ingresso e non di un segnale di clock comune a tutta la rete).

Infine è necessario valutare le modalità di sincronizzazione tra i processi, tenuto conto della necessità di questa funzione sia nelle architetture a memoria condivisa, dove l'operazione è più complessa, sia in quelle a invio di messaggio.

In conclusione il compito del programmatore è alquanto complesso, dipendendo dalle sue scelte quanto il software scritto si adatta al problema (caratteristiche del modello) e all'architettura hardware e di ambiente della macchina usata; da tale adattamento dipende il tempo necessario per sviluppare il programma, il grado di parallelismo raggiunto, l'onere e la complessità della manutenzione del prodotto.
Per fortuna il costruttore di macchine parallele, come già visto per le macchine vettoriali, offre un ambiente di programmazione che permette al programmatore di concentrare la propria attenzione soprattutto sul problema, anche se l'efficienza del prodotto sarà comunque legata alle scelte sopra ricordate [14-22].
Al fine di dare un esempio di utilizzo di un calcolatore parallelo, nel seguito si espone un caso molto semplice di valore accademico, ma interessante per i molti commenti che permette.
Con riferimento all'ambiente parallelo IBM 3090 e CRAY XMP, si consideri il semplice esempio di sommare in un accumulatore gli n elementi di un vettore (un esempio di operazione di riduzione).
Si consideri innanzi tutto la versione scalare, che può essere scritta nel seguente modo:

```
      PARAMETER (NIN = 5, NOU = 6, ND = 1000000)
      DIMENSION A (ND)
      READ (NIN, *) A
      CALL SOMMAT (RIS, A, ND)
      WRITE (NOU,*) RIS
      END
      SUBROUTINE SOMMAT (RIS,V, N)
      DIMENSION V (N)
      RIS = 0
      DO 10   J = 1, N
          RIS = RIS + V (J)
10    CONTINUE
       END
```

Come si vede si tratta di un programma elementare costituito da un programma principale e da un'unica subroutine. Il programma principale ha il compito di gestire la fase di input, di calcolo e di output. La subroutine effettua il calcolo, ossia accumula su uno scalare i valori di un vettore di lunghezza arbitraria. Il calcolo richiede un impegno linearmente proporzionale al numero di elementi del vettore e la perdita di efficienza potrebbe derivare unicamente dall'overhead causato dalle operazioni di salvataggio dei registri all'atto della chiamata e di recupero dei valori salvati al termine del calcolo. Questo peso potrebbe essere preponderante rispetto al peso del calcolo stesso se il vettore V fosse particolarmente corto.

Una versione per IBM 3090 parallelo è la seguente, dove con i caratteri minuscoli sono evidenziate le estensioni del Fortran previste in ambito IBM per un'opportuna strutturazione del programma e per dar direttive al compilatore per un uso ottimale della macchina:

```
      PARAMETER (NMT = 100, NIN = 5, NOU = 6)
      PARAMETER (ND = 1000000)
      INTEGER MYTASK (NMT), KPUNTA (NMT), NTERMI
      (NMT)
      DIMENSION RIS (NMT)
      COMMON/GLOB1/ A (ND)
      NTASK = MIN (nprocs (), NMT)                    (1)
      READ (NIN ,*) A
      DO 10  I = 1, NTASK
          originate any task MYTASK (I)               (2)
10    CONTINUE
      DO 20 K = 1, NTASK
          KPUNTA (K) = (K-1) (ND/NTASK) + 1           (3)
          NTERMI (K) = K* (ND/NTASK)
          IF (K. EQ. NTASK) NTERMI (K) = ND
          schedule task MYTASK (K),                   (4)
      &       sharing (GLOB1),
      &       calling SOMMAP (KPUNTA (K), NTERMI
              (K),RIS (K))
20    CONTINUE
      wait for all tasks                              (5)
      DO  40 I = 1, NTASK
          terminate task MYTASK (I)                   (6)
40    CONTINUE
      CALL SOMMAT (RRIS, NTASK, RIS)
      WRITE (NOU,*) RRIS
      END
      SUBROUTINE SOMMAP (NP, NF, RIS)
      PARAMETER (ND = 1000000)
      COMMON/GLOB1/ A (ND)
      RIS = 0. 0
      DO 10 J = NP, NF
          RIS = RIS + A (J)
10    CONTINUE
      END
```

Rispetto alla versione scalare le variazioni della routine di calcolo SOMMAP sono ridotte alla semplice introduzione di un COMMON che rappresenta la definizione dei dati globali più importanti. Le variazioni più importanti riguardano il programma principale che si deve far carico della ripartizione delle attività tra i vari processori.

Con l'istruzione (1) viene innanzitutto risolto il problema di comunicare al programma il numero di processori

(virtuali) disponibili. La primitiva "nprocs" accede al valore che l'utente specifica all'atto dell'esecuzione. Può essere specificato un numero di cpu virtuali arbitrario. Il sistema provvede automaticamente ad associare ad ogni cpu virtuale una data cpu reale. A tal riguardo si osserva che i processori virtuali non corrispondono necessariamente a processori fisici disponibili, ma rappresentano un modo per descrivere il parallelismo della macchina virtuale a cui fa riferimento il programmatore. La programmazione riguarda i task che devono essere svolti. Se questi sono tutti di uguale impegno e tanti quanti i processori si ottiene la massima efficienza. L'uso corretto della funzione "nprocs" consente quindi questa ottimizzazione.
Con l'istruzione (2) si definiscono i nomi dei task che si prevede di effettuare. Il vettore MYTASK conserva l'identificatore di ciascun task.
Le quantità che all'istruzione (4) saranno passate in argomento alle subroutine permettono di operare sulle variabili KPUNTA, NTERMI, RIS, che sono globali, come se fossero locali per ciascuna subroutine.
Con l'istruzione (5) si sincronizzano le attività lanciate in parallelo per la corretta prosecuzione del programma principale.
Con l'istruzione (6) si chiude la fase del programma avviata con l'istruzione (2).
Un'altra versione più raffinata dello stesso programma nella quale non è necessario attendere il completamento di tutte le task è la seguente:

```
      PARAMETER (NMT = 100, NIN = 5, NOU = 6)
      PARAMETER (ND = 1000000)
      INTEGER MYTASK (NMT), KPUNTA (NMT), NTERMI
      (NMT)
      DIMENSION RIS (NMT)
      COMMON/GLOB1/ A (ND)
      NTASK = MIN (nprocs (), NMT)
      READ (NIN, *) A
      DO 10 I = 1, NTASK
         originate any task          MYTASK (I)
10    CONTINUE
      DO 20 K = 1, NTASK
         KPUNTA (K) = (K-1) * (ND/NTASK) +1
         NTERMI (K) = K* (ND/NTASK)
         IF (K. EQ. NTASK) NTERMI  (K) = ND
         schedule task  NYTASK (K)
     &   tagging (K),
     &   sharing  (GLOB1),
     &   calling  SOMMAP (KPUNTA (K), NTERMI (K), RIS (K))
20    CONTINUE
      RRIS = 0. 0
      DO 4O I = 1, NTASK
         wait for any task ID, tagging (N)
         RRIS = RRIS + RIS (N)
         terminate task ID
40    CONTINUE
      WRITE (NOU, *) RRIS
      END
      SUBROUTINE SOMMAP (NP, NF, RIS)
      PARAMETER (ND = 1000000)
      COMMON/GLOB1/A (ND)
      RIS = 0.0
      DO 10 J = NP, NF
         RIS = RIS + A (J)
10    CONTINUE
      END
```

In questo caso il valore della somma è calcolato sommando i valori parziali ottenuti man mano che le singole task terminano l'esecuzione. Il parametro tagging è necessario per identificare la somma parziale da aggiungere.
Invece di gestire direttamente la generazione dei task paralleli e la loro sincronizzazione, si può ricorrere alla loro generazione automatica mediante il "parallel loop". L'esempio che segue illustra l'applicazione del "parallel loop":

```
      PARAMETER (NIN = 5, NOU = 6, NMT = 100, ND = 1000000)
      DIMENSION A (ND)
      NTASK = MIN (nprocs(), NMT)
      READ  (NIN, *) A
      RRIS = 0
      parallel loop 20 K = l, NTASK                    (1)
         private (RIS, KPUNTA, NTERMI)                 (2)
         do first                                      (3)
            RIS = 0
         do every                                      (4)
            KPUNTA = (K - 1) * (ND/NTASK) +1
            NTERMI = K * (ND/NTASK)
            IF (K. EQ. NTASK) NTERMI = ND
            DO 10 J = KPUNTA, NTERMI
               RIS = RIS + A (J)
10       CONTINUE
      do final lock                                    (5)
         RRIS = RRIS + RIS
20    CONTINUE
      WRITE (NOU, *) RRIS
      END
```

Osservazioni:

- All'interno di un "parallel loop", di sintassi simile a quella del classico DO Fortran, possono essere dichiarate delle variabili locali alla task tramite l'istruzione private".
  Anche l'indice del "parallel loop" è una variabile privata.

- L'istruzione (2), in cui si dichiarano private ad ogni singolo microtask le variabili RIS, KPUNTA, NTERMI, dev'essere posta subito dopo l'istruzione (1).

- Il " parallel loop" non rappresenta necessariamente l'attivazione di tante task quanti sono i passi della variabile del loop. Il sistema tenderà a definirsi un numero di task ottimale e a far svolgere a ciascuno di essi un numero di iterazioni appropriato. Esiste quindi la necessità di definire cosa fare sia all'inizio che alla fine di ciascuno dei task aperti automaticamente. Le istruzioni "do first" (3) e "do final" (5) hanno questo scopo.

- Mentre l'istruzione "do every" (4) specifica cosa dev'essere fatto ad ogni incremento dell'indice del "parallel loop".

- Il parametro "lock" protegge l'effetuazione del salvataggio dei risultati parziali nella variabile globale RRIS. Si noti che questa organizzazione ottimizza le prestazioni in funzione della disponibilità dei processori. Il sistema può infatti decidere dinamicamente, in funzione del valore di NTASK, la ripartizione dei compiti tra le varie cpu.

Con riferimento all'ambiente parallelo CRAY, il programma viene scritto ricorrendo a primitive apposite:

```
      PARAMETER (NMT = 4, NIN = 5, NOU = 6)
      PARAMETER (ND = 1000000)
      INTEGER MYTASK (2, NMT), KPUNTA (NMT), NTERMI
      (NMT)
      DIMENSION RIS (NMT)
      EXTERNAL SOMMAP
      COMMON/GLOB1/A (ND)
      NTASK = NMT
      READ (NIN, *) A
      DO 10 I = 1, NTASK
         MYTASK (1, I) = 2                           (1)
10    CONTINUE
      DO 20 K = 1, NTASK
         KPUNTA (K) = (K-1) * (ND/NTASK) + 1
         NTERMI (K) = K * (ND/NTASK)
         IF (K. EQ. NTASK) NTERMI (K) = ND
      CALL tskstart (MYTASK(1, K), SOMMAP,
      &  KPUNTA (K), NTERMI  (K), RIS (K))           (2)
20    CONTINUE
      RRIS = 0. 0
      DO 40 I = 1, NTASK
      CALL taskwait (MYTASK (1, I))                  (3)
      RRIS = RRIS + RRIS (I)
40    CONTINUE
      WRITE (NOU, *) RRIS
      END
      SUBROUTINE SOMMAP (NP, NF, RIS)
      PARAMETER (ND = 1000000)
      COMMON/GLOB1/A (AND)
      RIS = 0 0
      DO 10 J = NP, NF
         RIS = RIS+A(J)
10    CONTINUE
      END
```

Osservazioni:
(1) Prima di attivare le task occorre realizzare una matrice contenente dati utilizzati dal sistema. In pratica MYTASK serve per conservare l'identificatore della task generato dal sistema ed opzionalmente l'identificatore che l'utente vuole associare alla task. Per consentire questa scelta occorre specificare il numero di parole dedicate ad ogni task.

(2) E' una primitiva di attivazione dei vari task. Per ciascun task dev'essere specificato il vettore di riconoscimento del task, il nome della routine che il task utilizza, la lista degli argomenti che devono essere passati alla routine stessa.

(3) E' la primitiva di sincronizzazione. Essa causa la sospensione dell'esecuzione fino al momento in cui il task specificato ha completato la sua attività.

Va osservato che le variabili locali per il task sono quelle locali nella subroutine. Le variabili in COMMON o in argomento sono considerate globali. La routine chiamata può a sua volta chiamare al tre routine che risultano appartenenti allo stesso task. Per risolvere il problema della condivisione di variabili locali tra routine diverse, ma appartenenti allo stesso task viene introdotto il costrutto di task common. Il "task common" rappresenta un'estensione del linguaggio e consente di delinire aree globali, ma solo nell'ambito di uno stesso task. Nel loop 40 il programma attende che le task terminino secondo lo stesso ordine seguito per attivarle. Questo vuol dire che se la prima task è la più lunga il tempo dell'utilizzo dei risultati si aggiunge a quello della prima task. Una versione del DO 40 in cui il tempo totale corrisponde al tempo della task più lunga più il tempo per il trattamento unicamente dei risultati prodotti dalla task più lunga, è la seguente:

```
      NTASKA = NTASK
39    NORA = 0
      DO 40 K = l, NTASKA
         IF  (tsktest (MITASK (1, NTA (K))) THEN
            NORA = NORA + 1
            NTA (NORA) = NTA (K)
         ELSE
            RRIS = RRIS + RIS (NTA (K))
         ENDIF
40    CONTINUE
      NTASKA = NORA
      IF (NORA. GT. 0) GOTO 39
```

Il programma ritorna periodicamente alla label 39 dove viene azzerato il contatore del numero di task che non sono ancora terminati. Nel DO 40 il programma controlla quali dei task ancora in corso sono nel frattempo terminati. Per far ciò utilizza la primitiva "tsktest". In caso di responso positivo viene incrementato il contatore dei task ancora attivi, mentre se il responso è negativo il risultato prodotto dai task può essere utilizzato per completare la sommatoria.
In pratica quindi vengono fatte molte più operazioni nel

programma principale, ma si ha il vantaggio che il tempo globale è pari al tempo del task più lungo più il solo tempo dei calcoli da farsi sui risultati del task più lungo.Una versione che illustra lo stile SPMD può essere realizzata nell'ambiente IBM ricorrendo all'artificio di adibire il programma principale alla sola funzione di attivazione delle task e di controllo della loro conclusione. In pratica il programma principale dovrebbe essere visto come una specie di appendice del sistema operativo stesso. Nel caso del presente esempio esso potrebbe essere realizzato come segue :

```
      PROGRAM LANCIO
      INTEGER NTX (100), NYTASK (100)
      PARAMETER (ND = 1000000)
      COMMON/GLOB1/A (ND)
      COMMON/GLOB2/JSYNC, JAMEN, JEVEN, NPROC RRIS
      NPROC = nprocs()
      DO 1 NTA  = 1, NPROC
          NTX (NTA) = NTA
          originate any task MYTASK (NTA)                    (1)
1     CONTINUE
      CALL plorig (JSYNC)                                    (2)
      CALL peorig (JEVEN)                                    (3)
      CALL peorig (JAMEN)                                    (4)
      CALL peinit (JAMEN, NPROC, 1, 1)                       (5)
      DO 2 NTA = 1, NPROC
          schedule task MYTASK (NTA)                         (6)
              sharing (GLOB1,GLOB2),
              calling PMAIN (NTX (NTA))
2     CONTINUE
      wait for all task
      END                                                    (7)
```

In sostanza con l'istruzione (1) e con la (7) il programma attiva e conclude le NPROC task mentre con la (6) specifica il lavoro che queste task dovranno svolgere. Il lavoro è per tutte lo stesso ovvero la routine PMAIN (Pseudo MAIN) che, dal punto di vista della filosofia SPMD, rappresenta il singolo ed unico programma realizzato in modo da svolgere funzioni distinte (di controllo e di esecuzione) in base al proprio nome. In questo caso il nome è l'argomento di PMAIN e la convenzione adottata è che se il valore di tale argomento è uno, la routine PMAIN si debba far carico dei compiti aggiuntivi che, nell'approccio FORK-JOIN erano svolti dal solo programma principale.
Le istruzioni (2) (5) servono per inizializzare le variabili usate per la protezione delle fasi sequenziali e per lo scambio dei messaggi tra le task attivate da LANCIO. In particolare la variabile JSYNC svolge la funzione di chiave protettiva dell'operazione di accumulo dei risultati parziali della sommatoria. La variabile JEVEN viene utilizzata da PMAIN per dichiarare conclusa la fase di lettura dei dati e la variabile JAMEN serve per dichiarare conclusa la fase di calcolo delle somme parziali.
La variabile JAMEN richiede una inizializzazione più complessa ossia le istruzioni (4) e (5) perchè si vuole che tutte le NPROC task inviino il messaggio prima di dichiarare finita la fase di wait. Nel caso della variabile JEVEN basta viceversa che una sola delle task invii il messaggio per considerare finita la fase di wait che la riguarda.
La routine PMAIN è realizzata nel seguente modo:

```
      SUBROUTNE PMAIN (KISONO)
      PARAMETER (ND = 1000000)
      COMMON/ GLOB1/A (ND)
      COMMON/ GLOB2/JSYNC, JAMEN, JEVEN, NPROC, RRIS
      IF  (KISONO. EQ. 1) THEN
          READ (*,*) A
          CALL pepost (JEVEN)                                (1)
      ENDIF
      KPUNTA = (KISONO - 1) * (ND/NPROC) +1
      NTERMI = KISONO * (ND/NPROC)
      IF (KISONO EQ. NPROC) NTERMI = ND
      CALL pewait (JEVEN)                                    (2)
      CALL SOMMAP (KPUTA, NTERNI, RIS)
      CALL pllock (JSYNC)                                    (3)
      RRIS = RRIS+RIS
      CALL plfree (JSYNC)                                    (4)
      CALL pepost (JAMEN)                                    (5)
      CALL pewait (JAMEN)                                    (6)
      IF (KISONO. EQ. 1) WRITE (*,*) RRIS
      RETURN
      END
```

Come è tipico dell'approccio SPMD la PMAIN capisce di dover svolgere compiti particolari dalla variabile KISONO. Mentre la task che attua la lettura dei termini della sommatoria è impegnata a completare questo compito le altre possono dedicarsi all'effettuazione di calcoli preliminari conclusi i quali (istruzione (2)) attendono l'arrivo del messaggio di lettura terminata. Tale messaggio è trasmesso all'istruzione (1) dalla sola task che legge. I risultati delle sommatorie parziali confluiscono nel risultato globale RRIS. Le istruzioni (3) e (4) proteggono l'operazione di accumulo del contributo parziale mentre l'istruzione (5) serve per comunicare la conclusione del calcolo effettuato da ciascuna task. Al termine dell'attesa imposta dall'istruzione (6) la sola task che svolge anche operazioni di input/output effettua la stampa del risultato.

5. CONCLUSIONI

Le considerazioni svolte nei precedenti due paragrafi mettono in evidenza come gli ambienti offerti dai costruttori richiedano al programmatore un notevole lavoro e una non comune esperienza per ottenere elevate prestazioni utilizzando il Fortran e le sue estensioni per il calcolo vettoriale e parallelo. Mostrano inoltre che il risultato è spesso un programma non facilmente portabile e di difficile manutenzione. Rispetto alla programmazione delle macchine scalari, gli ambienti vettoriali e parallelo sono più complessi, non obbediscono a precisi standard e

possono piu facilmente modificarsi nel tempo con l'apparire di nuove famiglie di macchine. La letteratura scientifica riporta notevoli tentativi di risoluzione dei problemi citati:

- Scegliere lo stile di programmazione più adatto per sviluppare programmi corretti e di facile manutenzione e progettare l'architettura delle macchine in modo che essa sia la più aderente allo stile scelto. E' un vecchio sogno, che contrasta con la realtà della evoluzione delle architetture che ha finora tenuto in poco conto le necessità delle applicazioni. L'interesse per l'ambiente Fortran discende dalla grande quantità di programmi esistenti per il calcolo tecnico scientifico e dalla diffusa esperienza di programmazione, mentre lo sviluppo di programmi corretti e facilmente mantenibili richiederebbe un ambiente per linguaggi funzionali più che per linguaggi procedurali.
- Affidare il compito al compilatore di interfacciare l'architettura e il programmatore, essendoci una varietà di ambienti applicativi e una varietà di architetture che seguono loro logiche indipendenti e difficilmente controllabili. Il problema quindi di realizzare codici efficienti per un'assegnata architettura è lasciato al compilatore. In questa categoria rientrano i sistemi che eseguono automaticamente la vettorizzazione e la parallelizzazione dei codici. E' una tecnologia ancora non matura, con risultati interessanti, ma anche con problemi insoluti di grande rilevanza.
- Offrire al programmatore rappresentazioni esplicite della concorrenza con primitive per creare e terminare processi e per gestire la comunicazione tra processi. Queste primitive non devono essere specifiche della macchina, ma legate a concetti astratti quali FORK, JOIN ecc. Finora non si è trovato il modo di offrire una soluzione accettabile, in quanto queste primitive risul tano poco efficienti per alcune classi di macchine. Ad esempio costrutti paralleli per sistemi MIMD sono a volte non appropriati per sistemi SIMD.
- Permettere l'accesso alle primitive specifiche di ciascuna macchina. Data l'attuale grande varietà delle architetture, sembra essere questo il modo per ottenere elevate prestazioni. Esso tuttavia contrasta con l'obiettivo della portabilità, della facile manutenzione e della riusabilità e richiede una specifica specializzazione del programmatore, come lo richiedeva l'assembler. La soluzione ideale dovrebbe essere un compromesso tra le esigenze sopra espresse, ma è ancora lontana, soprattutto se si fa riferimento agli ambienti offerti dalle singole case costruttrici.

Per ora la soluzione va cercata con buon senso nell'ambito delle regole collaudate per un'ordinata programmazione. Nel caso si debbano trasformare codici scritti per macchine scalari si possono dare i seguenti suggerimenti :

- mantenere un'elevata modularità
- mantenere strutturato il codice
- usare commenti per rendere leggibile il codice e facilitarne la trasportabilità e la portabilità
- isolare le sezioni non trasportabili
- valutare attentamente dove viene svolta la parte più onerosa del calcolo attraverso molteplici test per una campionatura significativa del possibile utilizzo
- valutare quante risorse richiede il codice trasformato, in termini di memoria, numero di istruzioni, over heads, etc.
- cercare di conoscere cosa fa il programma. Spesso infatti la conversione è fatta da persone che non h anno questa conoscenza e ciò impedisce una efficace ottimizzazione.
- fare interventi graduali con controllo della loro correttezza passo-passo.

Nel caso invece che si intenda scrivere nuovi codici :

- scegliere il modello del problema più adatto al tipo di architettura disponibile
- procedere con metodi top-down verso una elevata modularità del programma, un'elevata chiarezza del sorgente e con un valido uso dei commenti
- isolare i moduli con i costrutti particolari che impediscono la portabilità del programma
- limitare la vettorizzazione e la parallelizzazione alle sezioni computazionalmente più onerose
- usare librerie di programmi ottimizzate
- limitare l'ottimizzazione dove veramente è necessaria.

Sono regole elementari ma costituiscono un primo passo in attesa della definizione del Fortran 8X e della fornitura di adeguati ambienti di sviluppo.

6. BIBLIOGRAFIA

[1] G.P. Bottoni, TECNICHE DI VETTORIZZAZIONE PER PROGRAMMAZIONE IN FORTRAN SULL'UNISYS ISP DEL CILEA, Bollettino CILEA n. 10, 1987, pagg. 16-23 e n. 12, 1987 pagg. 28-40

[2] B.L. Buzbee, A STRATEGY FOR VECTORIZATION, Parallel Computing, vol.3, 1986, pag.186

[3] J. Dongarra, S.C. Eisenstat, SQUEEZING THE MOST OUT OF AN ALGORITHM IN CRAY FORTRAN, ACM Transaction on Math. Software, VoI. 10 n. 3,1984, pag. 219

[4] B. Axelsson, V. Eijkout, A NOTE ON THE VECTORIZATION OF SCALAR RECURSIONS, Parallel Computing, Vol. 3, n.1 1986, pag. 735

[5] G. Erbacci, U. Fabbri, VETTORIZZAZIONE ED OTTIMAZIONE DEI PROGRAMMI FORTRAN PER

IL CRAY X-MPI/12 CINECA, Monografie CRAY, n. 4

[6] J.J. Dongarra, F.G. Gustavsom, A. Karp, IMPLEMENTING ALGORITHMS FOR DENSE MATRICES ON A VECTOR PIPELINE MACHINE, Siam Revie'v, VoI. 26 n. 1, 1984, pagg. 91-112

[7] G.R. Gao, A STABILITY CLASSIFICATION METHOD AND LTS APPLICATION TO PIPELINED SOLUTION OF LINEAR RECURRENCES, Parallel Computing, Vol. 4 n. 3, 305-321

[8] W. Gentzsch, G. Schaferj, SOLUTION OF LARGE LINEAR SYSTEMS OF VECTOR COMPUTERS, Proc. Int. Conf. Parallel Computing 83, 1984, pagg. 159-166

[9]W.D. Hillis, C.L. Steele, DATA PARALLEL ALGORITHMS, Commun. ACM, Vol. 29 n. 12, 1986, pagg. 1170-1183

[10] D.J. Kuck, HIGH-SPEED MACHINES AND THEIR COMERS, Parallel Processing Systems, Cambridge University Press,1982., pagg. 193-214

[11] D.A. Padua, M. J. Wolfe, ADVANCED COMPILER OPTIM' OR SUPERCORNPUTERS, Commun. ACM, Voe. 29, n.12,1986 pagg. 1184 - 1201

[12] D. Callhan, J. Dongarra, D. Levine, VECTORIZATION COMPILERS: A TEST SUITE AND RESULTS, Argonne Naturial Laboratory, Technical Memorandum n. 109, 1988

[13] A. Cantore, I. De Lotto, G. Meloni, APPLICAZIONI SCIENTIFICHE E TECNICHE DEI SUPER CALCOLATORI, Alta Frequenza LV, 2, 1986, pagg. 83-93

[14] A. H. Karp, PROGRAMMING FOR PARALLELISM, Computer, Vol. 20, n. 5,187, pagg. 43-57

[15] C.D. Howe, B. Moxon, HOW TO PROGRAM PARALLEL PROCESSORS, IEEE Spectrum, Settembre,1987 pagg. 36-41

[16] R.W. Hockney, C.R. Jessope, PARALLEL COMPUTERS, Adam Hilger Ltd, Bristol, 1981, capitolo IV

[17] K. Hwang, COMPUTER AND PARALLEL PROCESSING McGraw-Hill, New York, 1985, pagg. 533-551

[18] A.V. Aho, R. Sethi, J.D. Ullman, COMPILERS: PRINCIPLES, TECHNIQUES AND TOOLS, Addison Wesley, New York, 1986

[19] J.R. Allen, K. Kennedy, AUTOMATIC TRANSLATION OF FORTRAN PROGRAMS TO VECTOR FORM, Rice COMP TR 84-9, 1984

[20] J.R. Allen, DEPENDENCE ANALYSIS FOR-SUBSCRLQ S AND ITS APPLICATION TO PROGAM TRANSFORMATION, Rice Univer. Phd Thesis, 1983

[21] K.M. Chandy, ARCHITECTURE INDEPENDENT PROGRAMMING Proc. III Int. Conf. on Supercomputer Voe. 3, 1988, pagg. 345-351

[22] PARALLEL FORTRAN: LANGUAGE AND LIBRARY REFERENCE, IBM SC23-0431

[23] E.Ciementi ed altri, SUPERCOMPUTING: FOR SCIENCE AND ENGINEERING IN GENERAL AND FOR CHEMISTRY AND BIOSCIENCES IN PARTICULAR, Biological and Artificial Intelligence Systems, ESCOM, Leiden, 1988, pag. 319 - 424.

STRUMENTI SOFTWARE PER LA SIMULAZIONE DELLE RETI NEURONALI

Nicolò Cesa-Bianchi, Claudio Ferretti
Dipartimento di Scienze dell'Informazione
Università di Milano

RIASSUNTO

Nello studio dei modelli connessionisti, la simulazione può svolgere un ruolo importante per lo sviluppo di una teoria che ancora si trova nel suo stadio iniziale. In questo articolo, vengono presentati due strumenti software per la simulazione di reti neuronali. Il primo, sviluppato in C-language su VAX, è un sistema interattivo per la simulazione di modelli connessionisti definiti tramite un apposito linguaggio di specifica del quale viene descritta la sintassi. Il secondo è una libreria di funzioni C per l'implementazione efficiente dell'algoritmo di back-propagation su architettura sequenziale; l'ulteriore sviluppo di questo prototipo sarà finalizzato all'ottenimento di un sistema sufficientemente flessibile ed efficiente per essere applicato a problemi reali.

ABSTRACT

In the study of connectionist models, simulations can play a major role in the development of a theory which is still in its infancy. In this work, two software tools for neural network simulation are presented. The first one, written in C-language on a VAX computer, is an interactive system for the simulation of connectionist models. These models are defined by a proper specification language whose syntax is described in the paper. The second tool is a C-language library which tries to implement efficiently the back-propagation algorithm on a sequential computer; further work on this prototype will be addressed to the goal of obtaining a system flexible and efficient enough to be applied on real-world problems.

1. INTRODUZIONE

Lo studio delle reti neuronali è in quella fase iniziale del suo sviluppo in cui l'attività di ricerca puramente sperimentale, spesso solamente guidata da semplici intuizioni, può ancora fornire spunti interessanti di riflessione e spesso aprire la strada verso nuovi risultati. Per questo motivo, unitamente alla necessità di fornire mezzi adeguati per la verifica di ipotesi teoriche, è giustificabile ogni sforzo rivolto allo sviluppo di strumenti per la simulazione flessibile ed efficiente delle architetture connessioniste.

Il presente lavoro intende descrivere due strumenti per la simulazione delle reti neuronali in ambiente C-Unix. Si tratta di un simulatore software "general purpose", provvisto di strutture che permettono la definizione di un'ampia classe di architetture connessioniste, e di una libreria "special purpose" per l'implementazione dell'algoritmo di back-propagation, uno dei più diffusi per le applicazioni del Connessionismo ai problemi di pattern recognition. Questi due prodotti si propongono di soddisfare esigenze sia di carattere teorico-didattico, dove è privilegiata la flessibilità d'uso, sia di carattere esclusivamente applicativo dove entrano in gioco anche problemi legati all'efficienza implementativa. Lo sviluppo di software per simulazione neuronale, all'interno del Dipartimento di Scienze dell'Informazione dell'Università di Milano, si inserisce in un più ampio contesto di ricerca che fa capo al Laboratorio di Reti Neuronali recentemente costituito. Questa struttura, tramite l'acquisizione di tecnologia specifica e la raccolta di materiale bibliografico, intende offrirsi come ambiente rivolto a tutti coloro che intendono svolgere attività di ricerca nel settore delle reti neuronali.

2. METODI PER IL CALCOLO NEURONALE

Da un punto di vista puramente computazionale, una rete neuronale è un sistema di calcolo costituito da una collezione di semplici unità di elaborazione chiamate neuroni formali. La sequenza di computazione di una rete neuronale è rappresentabile dalla sequenza di stati assunti dalle unità di elaborazione che la compongono. In generale, i neuroni possono operare in parallelo comunicandosi reciprocamente il proprio stato corrente attraverso una densa rete di interconnessioni. Lo stato di alcune unità può essere vincolato dall'esterno.

Formalmente, una rete neuronale è costituita da un insieme di N neuroni formali che indichiamo con {1,2,..., N}. Ogni neurone possiede un proprio stato interno $x_i \in D$ dove D è un insieme arbitrario. Indicheremo con $\underline{x}$ il vettore N-dimensionale avente come i-esima componente lo stato x_i dell'i-esimo neurone. Questo vettore rappresenta lo stato globale della rete.
Una rete neuronale può essere rappresentata come un grafo orientato e pesato con i neuroni come vertici e le loro interconnessioni come archi. In particolare, ad ogni interconnessione da un neurone origine i ad un neurone destinazione j è associato un numero reale ω_{ij} detto peso. Se un'interconnessione non esiste allora il peso corrispondente è nullo. Rappresentiamo l'insieme delle interconnessioni con la matrice W di dimensioni $N \times N$ definita dai suoi elementi:

$$\omega_{ij} = \begin{cases} \text{peso fra } i \text{ e } j & \text{se la connessione esiste,} \\ 0 & \text{altrimenti;} \end{cases}$$

per $i, j \in \{1, 2, ..., N\}$.

Lo stato interno di ciascun neurone i è calcolato tramite una "funzione di attivazione" f_i il cui valore dipende, in generale, dalla matrice dei pesi W, dal vettore di stato dei neuroni $\underline{x}$ e, possibilmente, da un parametro reale aggiuntivo:

$$f_i : M(\mathrm{IR}, N \times N) \times D^N \times \mathrm{IR} \rightarrow D$$

Dove $M(\mathrm{IR}, N \times N)$ indica la classe delle matrici reali quadrate di ordine N.
Assumendo un tempo discreto t, possiamo così descrivere la dinamica della rete:

- ad ogni istante $t. \in \{0, 1, ...\}$ lo stato globale della rete è dato da $\underline{x}(t) = (x_1(t), ..., x_N(t))$;
- lo stato iniziale $\underline{x}(0)$ o viene introdotto dall'esterno come input al sistema o è casuale;
- lo stato futuro $\underline{x}(t + 1)$ è ottenuto tramite il ricalcolo dello stato di un sottoinsieme dei neuroni della rete, la regola che definisce di volta in volta tale sottoinsieme è detta la "modalità di attivazione" del modello considerato.

Come output della computazione viene generalmente considerato lo stato di equilibrio eventualmente raggiunto durante la computazione o, nel caso dei modelli probabilistici, la distribuzione stazionaria sullo spazio degli stati.
Come abbiamo visto il comportamento della rete è determinato da:

1) stato iniziale
2) modalità di attivazione della rete
3) funzione di attivazione
4) pesi delle connessioni

Il punto 1 è proprio di ogni singola computazione, come avviene per i dati di input di un normale programma. I punti 2 e 3 sono definiti per ciascun modello e quindi non rappresentano variabili dinamiche. L'alterazione dei pesi sulle connessioni (punto 4), invece, è un'operazione paragonabile alla programmazione nei computer tradizionali. Ma, a differenza di ciò che avviene per la programmazione convenzionale, gli elaboratori neuronali sono in grado di modificare autonomamente i propri parametri fino a soddisfare un dato insieme di specifiche definite, ad esempio, come coppie input/output. Questo processo di auto-adattamento è governato dalla cosiddetta "regola di apprendimento", caratteristica di ogni modello.

2.1 Alcuni modelli

Perceptrone [5]: uno dei primi modelli studiati; considera un neurone binario di output con stato $x \in \{0,1\}$ che riceve in ingresso lo stato di N altri neuroni binari di input. La funzione di attivazione è definita come:

$$f(\underline{x}) = \begin{cases} 1 \text{ se } \Sigma_j \omega_j x_j > r, \\ 0 \text{ altrimenti;} \end{cases} \qquad (2.1)$$

dove i ω_j sono i pesi sulle connessioni incidenti sul nodo di output. Una regola di apprendimento per questo modello corrisponde a rafforzare i pesi sulle connessioni fra neuroni attivi (regola di Hebb). In [MIN69] sono evidenziate le limitazioni computazionali del perceptrone.

A strati (error backpropagation) [6]: è un'estensione del precedente modello: i neuroni sono connessi in modo tale che $i > j \Rightarrow \omega_{ij} = 0$ strutturando la rete in una sequenza di strati dove ciascun neurone manda connessioni solo

verso gli strati superiori. Di questo modello, di cui parleremo più diffusamente nel seguito, ci limiteremo a dire che è in grado di superare le limitazioni computazionali del caso precedente.

Hopfield [2]: in questo caso la topologia della rete è arbitraria ma la matrice dei pesi W è vincolata ad essere simmetrica: ($\omega_{ij} = \omega_{ji}$). I neuroni sono binari e la funzione di attivazione è la (2.1). I neuroni vengono aggiornati uno alla volta in una sequenza casuale realizzando di fatto un processo stocastico. L'apprendimento consiste nell'organizzazione della rete in modo che alcuni stati siano stabili rispetto alla dinamica sopra esposta. In questo modo, se la rete è avviata in uno stato simile ad uno di quelli appresi, essa evolverà fino a stabilizzarsi in quest'ultimo. Grazie a questa proprietà la rete è in grado di funzionare come memoria auto-associativa.

Boltzmann machine [1]: il modello è simile al precedente tranne per la funzione di attivazione che è di tipo probabilistico:

$$f_i(\underline{x}) = \begin{cases} 1 \text{ con probabilità} & \dfrac{1}{1+\exp(-\Sigma_{j>i}\,\omega_{ji}\,x_i)/r} \\ 0 \text{ altrimenti} \end{cases} \qquad (2.2)$$

Il parametro r controlla l'instabilità dell'evoluzione della rete. La stocasticità delle f_i rende proficuo, nell'analisi di questi sistemi, l'uso di strumenti propri della meccanica statistica. L'apprendimento avviene secondo una regola locale che raccoglie statistiche alla Hebb in due fasi dinamiche distinte.

2.2 Simulazione di Reti Neuronali

Esistono già diversi sistemi software per la simulazione di reti neuronali, ciascuno in grado di operare su di una specifica classe di macchine, a volte come estensione di ambienti di programmazione tradizionali.
P3 [9], ad esempio, opera su LISP-machine Simbolycs ed offre, per la definizione delle reti, una estensione del linguaggio residente, di cui utilizza, durante la simulazione, sofisticati strumenti di input/output.
Anche CNS [8] è un'estensione del linguaggio LISP, ma richiede l'utilizzo di un normale personal computer PC-compatibile: in fase di simulazione offre più limitate capacità di visualizzazione.
Noi abbiamo sviluppato un sistema di simulazione il cui software è stato scritto in linguaggio "C ". Tuttavia lo sperimentatore può anche utilizzarlo senza dover conoscere la programmazione, disponendo, per le operazioni più importanti di un ambiente operativo proprio e autonomo.

Attualmente opera in ambiente UNIX, su hardware VAX Digital. I particolari che definiscono i diversi modelli, quali le funzioni d'attivazione e le regole di apprendimento, sono, per ragioni di efficienza, descritti da funzioni "C" compilate assieme al simulatore vero e proprio. Molte funzioni accompagnano già il programma, permettendo di eseguire simulazioni dei principali modelli di reti neuronali. Il sistema è stato organizzato in modo che la modifica o la creazione di queste funzioni possa avvenire non conoscendo che poche linee del resto del programma. Il simulatore, una volta attivato, comunica interattivamente con l'utente attraverso il terminale. Lo schermo è diviso in due aree: una rende disponibili una griglia di 10 x 10 posizioni in cui visualizzare lo stato di altrettanti neuroni, la seconda è utilizzata per digitare comandi di attivazione e controllo della simulazione.

La rete deve però essere descritta, in tutte le parti relative ai particolari di modello e di strttura, in un file che il sistema richiederà all'atto della effettiva simulazione. Neuroni e rete vi sono specificati con la seguente sintassi (Backus Naur Form modificata):

```
<rete> :: = NET
<numero_di_neuroni>
{<neurone>}
[LRN: <id_regola_apprendimento> [<buffer>]]

<neurone>::=<id_numerico_del_neurone>
[<numero_connessioni_entranti>
({<neurone origine> * <peso_associato>} )
[, <stato iniziale>
[, <attività>
[, <id_funzione_di_attivazione> ]]]]
```

La sintassi rispecchia le restrizioni da noi poste sulla generalità dei modelli da simulare, coerenti, comunque, con la descrizione del paradigma di calcolo neuronale esposta più sopra: la rete è descritta dai suoi neuroni e dalla regola di apprendimento, che può richiedere l'uso di buffer per statistiche aggiuntive; i neuroni sono specificati singolarmente, ciascuno con la propria funzione d'attivazione, indicata, come la regola d'apprendimento, da un identificatore scelto tra quelli disponibili e noti al simulatore: lo stato è un numero reale; l'attivazione del neurone avviene ad intervalli di durata casuale di cui noi specifichiamo, con <attività>, la media; i neuroni attivi in un dato istante sono aggiornati in modo sincrono.

L'attività del simulatore è controllata da un linguaggio a comandi. Questo è completo di strutture analoghe a quelle disponibili in ambiente shell/UNIX, quali la sequenza e il ciclo "FOR", ma anche di strutture proprie dell'ambito simulativo neuronale, come il ciclo srlx (<comandi>), che

termina quando la rete presenta in due iterazioni successive lo stesso stato globale. I principali comandi sono:
nnet <nome_file>: legge dal file la definizione della rete da attivare e simulare; il file porta anche altri dati
du <nome_unità> cerca nel file attivato un elenco di neuroni da visualizzare; si. puo' scegliere di mostrarne lo stato in forma numerica o analogica, a barra
iu <nome_unità>: cerca nel file un elenco di neuroni da considerare "di input" e rendere inattivi
nex: legge dal file una configurazione di stato da assegnare ai neuroni di input; posso anche descriverne molte e leggerne una alla volta
set <neurone> <stato>: assegna al neurone lo stato indicato nel comando o uno casuale
stp <numero_passi>: attiva la rete per un certo numero di istanti, o passi di computazione
wlrn : applica la regola di apprendimento prescelta, aggiornando i pesi di tutte le connessioni, eventualmente anche in base a dati raccolti durante la computazione nel buffer richiesto in fase di definizione della rete
save <nome file>: salva, nel file specificato e in formato leggibile dal comando "nnet" lo stato corrente di rete e connessioni; così si può rendere utilizzabile in altre sessioni una rete sintetizzata tramite apprendimento.
Supponiamo ora di voler costruire e utilizzare la rete di Fig. 2.2. Il modello utilizzato permette l'apprendimento di uno stato globale, o immagine, caricandolo nella rete e calcolando i pesi con una formula. Noi comunichiamo quest'ultima al simulatore in forma di funzione "C" per l'apprendimento, con un suo identificatore. La funzione di attivazione del modello è già nota al programma e ne daremo solo l'identificatore. Definiamo così una rete di 3x5=15 neuroni, tutti uguali nelle specifiche (attività pari a 1/3) e completamente connessi. Specifichiamo la nostra regola di apprendimento senza richiedere buffer dati.
Disponendo del "file" attiviamo il simulatore e digitiamo:
nnet file; iu unit; nex; wlrn; save sint.

Con questa sequenza abbiamo attivato la rete, letto da file che tutti i neuroni erano di input, caricata l'immagine nei neuroni, calcolati i pesi in base agli stati presenti e salvata la rete ottenuta.
Ora digitiamo:
nnet sint; set nx 0; set ny 1 ; set ...; srlx(stp 5),
attivando la nuova rete, assegnando una configurazione arbitraria e facendo operare la rete fino a che lo stato

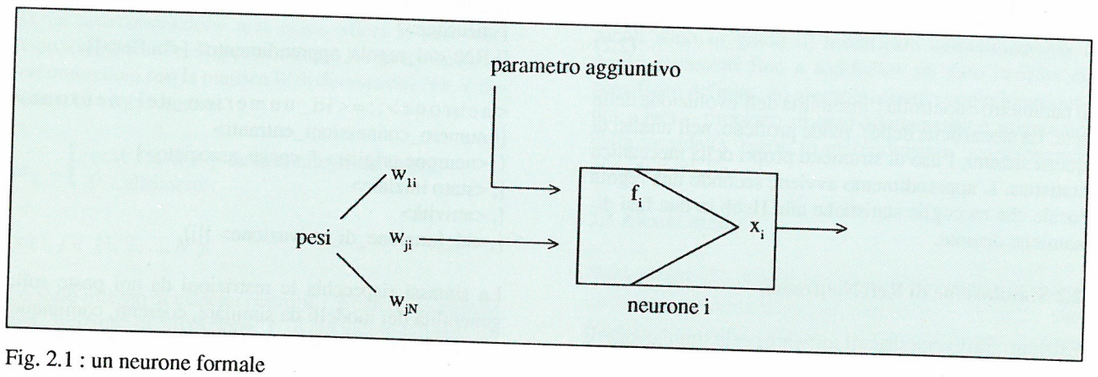

Fig. 2.1 : un neurone formale

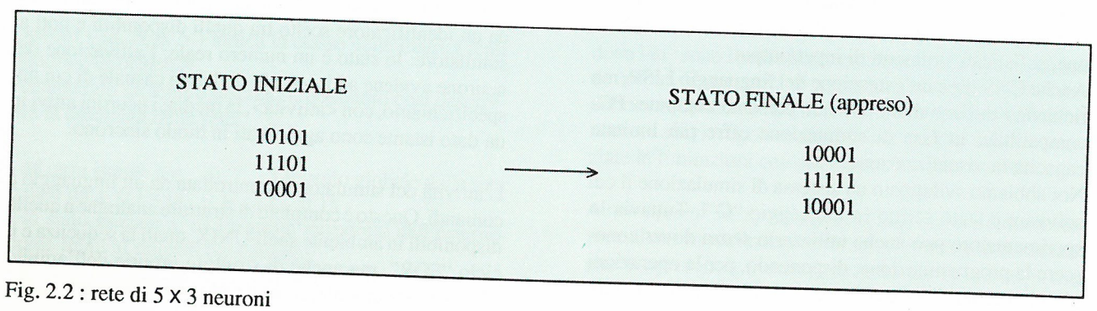

Fig. 2.2 : rete di 5 x 3 neuroni

rimane invariato per 5 passi di computazione.

3. UNA LIBRERIA IN C-LANGUAGE PER L'ALGORITMO DI BACK-PROPAGATION

L'algoritmo di error back-propagation (BP), introdotto nel 1985 da Rumelhart et al. [6] e [7], è una tecnica di apprendimento tramite esempi mediante la quale è possibile costruire applicazioni arbitrarie fra insiemi di stringhe binarie di lunghezza prefissata. La BP costituisce una generalizzazione dell'algoritmo di apprendimento per il Perceptrone sviluppato da Rosenblatt nei primi anni '60 [5]. Mediante questa tecnica era possibile calcolare soltanto applicazioni caratterizzabili come funzioni booleane linearmente separabili [4]. Attualmente, la BP rappresenta un algoritmo in rapida via di diffusione in vari campi applicativi. Fra i settori maggiormente interessati possiamo ricordare:

- riconoscimento di caratteri,
- riconoscimento del parlato,
- identificazione di tracce radar e sonar,
- analisi di immagini in medicina,
- robotica.

Ricordiamo anche che un interessante ramo di ricerca è rivolto allo sviluppo di chip analogici in grado di realizzare l'algoritmo di back-propagation direttamente a livello hardware.

3.1 Descrizione dell'algoritmo

In questo contesto, caratterizziamo un problema di applicazione fra stringhe binarie tramite una funzione booleana parziale definita da un insieme di coppie input/output:

$$T = \{(\underline{\alpha}^1, \underline{\beta}^1), \ldots (\underline{\alpha}^N, \underline{\beta}^N)\} \qquad (3.1)$$

dove $\underline{\alpha}^k \in \{0,1\}^m$ e $\underline{\beta}^k \in \{0,1\}^n$ per $k = 1,2,\ldots N$. L'insieme T costituisce il training set, ovvero l'insieme degli esempi ripetutamente forniti in ingresso all'algoritmo di apprendimento finché ogni coppia input/output di T non verrà riprodotta correttamente.
Il processo di apprendimento si svolge modificando progressivamente due matrici reali A e B - rispettivamente di dimensione (h x m) e (n x h) con h opportuno - dette matrici dei pesi, e due vettori reali $\underline{\tau}^A$ $\underline{\tau}^B$- rispettivamente di dimensione h e n detti vettori delle soglie. Tali matrici e tali vettori definiscono una funzione parziale Φ da $\{0,1\}^m$ a $\{0,1\}^n$ che associa ad ogni $\underline{\alpha}^k$ un vettore $\underline{z} \in \{0,1\}^n$ o un valore indefinito. La funzione Φ è calcolata nel modo seguente:

STEP 1. Dato $\underline{u} \in \{0,1\}^m$ tale che $\underline{u} = \underline{\alpha}^k$ per un certo k fra 1e N, si calcola:

$$\underline{v}' = (f(v_1,\ldots, f(v_h)) \qquad (3.2)$$

dove:

$$\underline{v} = A\underline{u} - \underline{\tau}^A \qquad (3.3)$$

con $A\underline{u}$ che indica l'usuale prodotto righe per colonne

$$f : R \rightarrow (0,1),\ x \rightarrow f(x) := \frac{1}{1+e^{-cx}} \qquad (3.4)$$

con $c > 0$

STEP 2. Si calcola :

$$\underline{w}' = (f(w_1,\ldots, f(w_n)) \qquad (3.5)$$

dove:

$$\underline{w} = B\underline{v}' - \underline{\tau}^B \qquad (3.6)$$

STEP 3 Le componenti del vettore risultante $\underline{w}$, che in generale appartengono all'intervallo (0,1), vengono normalizzate nel vettore di output $\underline{z}$ le cui componenti (z_1, z_2,...z_n) sono definite come:

$$z_i = \begin{cases} 0 \text{ se } w_i' < \theta \\ 1 \text{ se } 1 - w_i' < \theta \\ \lrcorner \text{ altrimenti} \end{cases} \qquad (3.7)$$

dove θ è un coefficiente positivo e minore di 0.5 e $\lrcorner$ rappresenta un valore indefinito fra 0.5 - θ e 0.5 + θ. Se esiste un i tale che $z_i = \lrcorner$ allora $\underline{u}$ non appartiene al dominio di Φ, altrimenti $\Phi(\underline{u}) = \underline{z}$.

Si osservi che il valore di h - che concorre a determinare le dimensioni di A, B e τ^A non è immediatamente determinabile dalla specifica T del problema. Al contrario, il problema della decisione a priori di h è tuttora aperto. Ci limitiamo ad osservare che h deve essere comunque maggiore di un certo valor minimo, dipendente da T, al di sotto del quale il sistema non possiede un numero di parametri sufficiente per il calcolo di T. D'altra parte, valori troppo grandi di h rispetto al problema considerato conducono ad uno spreco evidente di risorse computazionali.

In ogni caso, la questione se una scelta opportuna di h basti a garantire la possibilità di realizzare un'applicazione arbitraria fra $\{0,1\}^m$ e $\{0,1\}^n$ è stata affrontata e risolta in [3] per una classe di funzioni che comprende la (3.4).
Il processo di apprendimento realizzato dalla procedura di backpropagation consiste nell'applicazione del metodo del gradiente per minimizzare una funzione E che misura lo scarto fra la funzione parziale Φ e la funzione obiettivo

definita da T. La funzione E, per ogni coppia $(\underline{\alpha}^\kappa, \underline{\beta}^\kappa) \in T$, è definita come:

$$E_k = \frac{1}{2} \sum_{i=1}^{n} (z_i - \beta^\kappa_i)^2 \qquad (3.8)$$

dove ogni z_i è ottenuto a partire da $\underline{\alpha}^\kappa$ tramite (3.2)-(3.7). Applicando il metodo del gradiente, otteniamo che le variazioni delle componenti di A e B sono espresse da:

$$\Delta_K a_{ji} = -\eta \frac{\partial E_k}{\partial a_{ji}} \qquad \Delta_K b_{ji} = -\eta \frac{\partial E_k}{\partial b_{ji}} \qquad (3.9)$$

con η coefficiente reale positivo detto learning rate. Come è dimostrato in [RUM85], è possibile derivare analiticamente le seguenti espressioni per il calcolo delle derivate parziali per ogni componente di A e B:

$$\frac{\partial E_k}{\partial b_{ji}} = v'_i \delta_j^B, \qquad \frac{\partial E_k}{\partial a_{ji}} = u_i \delta_j^A, \qquad (3.10)$$

dove:

$$\delta_j^B = c\,(w'_j - \alpha^\kappa_j)\, w'_j\, (1 - w'_j); \qquad (3.11)$$

$$\delta_j^A = c v'_i\, (1 - v'_i) \sum_{i=1}^{n} \delta_j^B b_{ji} \qquad (3.12)$$

Perciò alla k-esima iterazione del metodo del gradiente, modificheremo le matrici dei pesi $A^{(k)}$ e $B^{(k)}$ secondo lo schema:

$$a_{ji}^{(k+1)} = a_{ji}^{(k)} - \eta u_i \delta_j^A \qquad b_{ji}^{(k+1)} = b_{ji}^{(k)} - \eta v'_i \delta_j^B \qquad (3.13)$$

Osserviamo che, in generale, lo spazio parametrico individuato da A, B, $\mathfrak{I}^A$ e $\mathfrak{I}^B$ rispetto alla funzione E non è convesso. Questo significa che, in generale, il metodo del gradiente (e di conseguenza l'algoritmo di BP) può non convergere al minimo di E per particolari stati iniziali e per particolari istanze di T.

3.2 Implementazione

L'algoritmo di BP è stato realizzato in ambiente C-UNIX come una libreria di funzioni richiamabili all'interno di programmi C. L'architettura del sistema intende riferirsi al modello concettuale di "tipo di dato astratto", in cui la definizione di un oggetto è basata sulla descrizione del suo comportamento astratta dalla struttura particolare che lo realizza. In particolare, un dato risulta caratterizzato dalle operazioni che sono effettuabili su di esso (l'interfaccia funzionale) piuttosto che dal tipo particolare di implementazione scelto. Questa trasparenza rende la parte implementativa suscettibile di cambiamenti che non si ripercuotono sulle parti che utilizzano il dato servendosi dell'astrazione offerta dall'interfaccia funzionale.

Attenendoci alla terminologia utilizzata in letteratura per la BP, conveniamo di chiamare vettore di stato per lo strato di input (input layer) il vettore $\underline{u}$ in (3.3), vettore di stato per lo strato nascosto (hidden layer) il vettore $\underline{v}'$ in (3.2) e vettore di stato per lo strato di output (output layer) il vettore $\underline{w}'$ in (3.5). Nel gergo connessionista, il modello qui considerato è chiamato "perceptrone generalizzato a tre strati".
Un prototipo del programma, che conteneva la caratterizzazione della BP come tipo di dato astratto, era stato già sviluppato da Franco Sicoli nell'ambito della propria tesi di laurea. Il presente lavoro costituisce perciò uno sviluppo di quel prototipo di cui sono state mantenute alcune fondamentali caratteristiche.
Diamo ora un elenco delle principali funzioni messe a disposizione dalla libreria.

Funzioni per la definizione dei parametri della rete:

. netsetdim (layer, size): fissa le dimensioni dei tre layer (input, hidden e output).
. netsetlr (value): fissa il valore del learning rate η
. netsetslope (value): fissa il valore del parametro c relativo all'andamento della funzione (3.4).
. netinit (w1, w2): inizializza le componenti di A, B, $\mathfrak{I}^A$ e $\mathfrak{I}^B$ assegnando a ciascuna un valore random fra w1 e w2.
. netsetw (parameter, position, vector): inizializza una riga della matrice A o B (o equivalentemente uno dei vettori $\mathfrak{I}^A$ e $\mathfrak{I}^B$) coi valori contenuti in vector.
. netseti (vector): carica l'input layer coi valori contenuti in vector.
. netsett (vector): carica il target array coi valori contenuti in vector.

Funzioni per l'utilizzo della rete:

. netrun(): calcola la funzione computata dalla rete rispetto al valore corrente dell'input layer.
. netout (vector): restituisce in vector il valore dell'output layer.
. nettrain(): calcola le componenti del gradiente rispetto ai valori attuali dei layer input, hidden, output e del target array.
. netchange(): aggiorna i valori delle matrici A e D e dei vettori $\mathfrak{I}^A$ e $\mathfrak{I}^B$ rispetto al gradiente precedentemente calcolato con nettrain().
. neterror(): calcola il valore della funzione di errore E rispetto al valore corrente dell'output layer e del target array.

Oltre a queste, esistono altre funzioni (che qui non menzioniamo) per il caricamento e scaricamento su file di una rete precedentemente istanziata. L'allocazione in memoria di una rete deve essere dichiarata esplicitamente tramite la funzione netalloc().

Il tentativo di applicare efficientemente l'algoritmo di BP a problemi reali ha avuto come effetto collaterale lo sviluppo di un insieme di tecniche e accorgimenti che rappresentano un primo esempio di *engineering* nel campo del neural computing. Sulla base di questo, stiamo ampliando il prototipo qui presentato per permettere una più vasta libertà di scelta nell'architettura della rete e nella specifica parametrica dell'algoritmo. E' anche possibile migliorare la caratterizzazione del sistema come tipo di dato astratto utilizzando, per esempio, linguaggi C-like object oriented come il C++.

4. BIBLIOGRAFIA

[1] G.E. Hinton, T.J. Sejnowski, D.H. Ackley, BOLTZMANN MACHINES: CONSTRAINT SATISFACTION MACHINES THAT LEARN, tech. rep. CMU-CS-84-119, CMU, Pittsburgh, 1984.

[2] J.J. Hopfield, NEURAL NETWORKS AND PHYSICAL SYSTEMS WITH EMERGENT COLLECTIVE COMPUTATIONAL ABILITIES, proc. Natl. Acad. Sci. USA, 79, (2554-2558), 1982.

[3] B. Irie, S. Miyake, CAPABILITIES OF THREE-LAYERED PERCEPTRONS, proc. of IEEE International Conference on Neural Networks, 1 (641-648), IEEE Press, New York, 1988.

[4] M. Minsky, S. Papert, PERCEPTRONS, MIT Press, Cambridge MA, 1969.

[5] F. Rosenblatt, PRINCIPLES OF NEURODYNAMICS, Spartan, New York, 1962.

[6] J.E. Rumelhart, G.E. Hinton, R.I. Williams, LEARNING INTERNAL REPRESENTATIONS BY ERROR PROPAGATION, Institute for Cognitive Science technical report 8506, UCSD, La Jolla, CA 92093, 1985.

[7] D.E. Rumelhart J.L. McClelland (eds.), PARALLEL DISTRIBUTED PROCESSING, MIT Press, Cambridge, MA, 1986.

[8] Z. Schreter, CNS, Genetic AI and Epistemics Laboratory, University of Geneva, 1987.

[9] D. Zipser, D. Rabin, P3: A PARALLEL NETWORK SIMULATING SYSTEM, draft: Inst. for Cognitive Sc., University of California, 1984.

NOTE DI SOFTWARE 42/43

OBJECT ORIENTED PROGRAMMING E PROLOG OVVERO L' IMPLEMENTAZIONE DI OGGETTI IN PROLOG

Roberto Grande
Dipartimento di Scienze dell'Informazione
Università di Milano

RIASSUNTO

Il paradigma di programmazione Object Oriented si é rivelato promettente in due direzioni. Come strumento di rappresentazione della conoscenza più aderente al modo con cui noi concettualizziamo la realtà. Come metodologia di sviluppo del software improntata alla gestione modulare e alla condivisione delle risorse. In questo articolo si analizza la possibilità di sfruttare entrambi questi aspetti nell'ambito della tecnologia della Programmazione Logica.
L'analisi é centrata sulla corrispondenza tra le caratteristiche componenti dei due paradigmi della Programmazione Object Oriented e della Programmazione Logica, con particolare riguardo all'implementazione delle tecniche di astrazione , della modularizzazione e dello stato di un oggetto.

ABSTRACT

The Object Oriented Programming paradigm has revealed promising for two main reasons. As a tool for Knowledge Representation more adherent to the way we capture the real world. As a software developement methodology based on modularity and resource sharing techniques. In this paper we analyse the possibility of exploiting the above features within the Logic Programming technology. The analysis is centered on the correspondence between components of the two paradigms. Particular attention is devoted to the implementation of abstraction techniques, of modularization and of object's state.

INTRODUZIONE

L'approccio Object Oriented (OOA) all'ingegneria del software presenta dei vantaggi nella fase di analisi del problema, nella fase di definizione delle specifiche, nella fase di implementazione (sintesi) e nella fase di manutenzione.

In ognuna di queste fasi i vantaggi derivano per lo più da una gestione modulare delle risorse. Ciò é reso possibile da determinate caratteristiche, comuni agli ambienti Object Oriented Programming (OOP), che sono "ad alto livello" rispetto al livello sintattico dei linguaggi di programmazione; queste caratteristiche possono quindi essere "sovraimposte" a quelle dei singoli linguaggi di base, dando luogo a diversi ambienti object oriented. Ne sono un esempio C++ per il C, Object Pascal per il Pascal, LOOPS per il LISP, SPOOL e Vulcan per il PROLOG.

In questo articolo si intende analizzare la possibilità di importare caratteristiche tipiche degli OOPS nell'ambito della Programmazione Logica; come elemento rappresentativo della classe dei linguaggi di Programmazione Logica é stato scelto PROLOG perchè, tuttora, é quello più usato.
Si sono esclusi dall'analisi tutti quei tentativi di "fondere" i due paradigmi di programmazione (in questo articolo si intende il termine "paradigma" come equivalente di "stile di programmazione") mediante la creazione di ambienti in cui entrambi coesistono sotto forma di due sottoambienti distinti che comunicano attraverso una interfaccia (ed eventualmente un data base condiviso) e a cui il

programmatore può accedere, di volta in volta, secondo l'una o l'altra modalità.
Il punto di vista che si é scelto é quello di considerare PROLOG come il linguaggio di programmazione di base, e il paradigmia OOP come un insieme di metodologie di analisi e di tecniche di programmazione (cliché) da sovraimporre a PROLOG; l'ambiente risultante dovrà comprendere delle estensioni al linguaggio PROLOG in modo da fornire un opportuno supporto sintattico alle caratteristiche object oriented che si vogliono introdurre.

2. MOTIVAZIONI

Una prima considerazione di carattere generale a favore dell'opportunità di ambienti per la Programmazione Logica improntati all'OOP é costituita dalla parziale sovrapposizione delle aree di applicazione in cui sono stati separatamente impiegati i due paradigmi (e.g.: Knowledge-based systems, expert sistems, model building, database research, interfaces modelling, etc...).
I vantaggi apportati dai due singoli paradigmi sono complementari e sinergici.

Dal punto di vista della rappresentazione della conoscenza sono fondamentali la potenza espressiva, la leggibilità e la possibilità di strutturare ed organizzare la conoscenza offerte da un formalismo. L'OOP permette di organizzare la base di conoscenza come una collezione di *oggetti* che interagiscono attraverso un meccanismo uniforme (*message passing*); favorendo in questo modo la modularizzazione della conoscenza. Un' altra caratteristica vantaggiosa é quella dell'*ereditarietà* tra *classi*. Essa permette la creazione di nuovi oggetti specializzando oggetti simili esistenti; questa caratteristica contribuisce alla fattorizzazione della conoscenza in una gerarchia di classi. D'altra parte la potenza espressiva e la leggibilità di un linguaggio OOP, considerate ad un livello di granularità basso (a livello di definizione di *metodi* o di classi), non sono migliori di quelle offerte dai tradizionali linguaggi procedurali. A ciò, crediamo possa ovviare la potenza espressiva e la leggibilità (possibilità di una lettura dichiarativa) offerte da PROLOG. In altre parole riteniamo che sia vantaggioso mantenere una semantica dichiarativa per ciò che riguarda la rappresentazione della conoscenza a basso livello di granularità e, nel contempo, sfruttare la semantica procedurale offerta dall'OOP per una migliore strutturazione della conoscenza ad un livello di granularità più alto (unsiemi di classi, interazione tra oggetti, definizione dell'ereditarietà) ed una organizzazione più flessibile del controllo sull'uso di questa conoscenza (che non il controllo offerto dal semplice backtracking automatico di PROLOG).

Dal punto di vista dell'ingegneria del software non é il caso di ribadire qui i vantaggi che comporta l'approccio OOP (rimandiamo a Meyer, [11]) mentre vale la pena sottolineare che proprio uno di quei vantaggi, la facile modificabilità dei programmi, acquista ulteriore importanza in applicazioni di A.I. Molto spesso in queste applicazioni il programmatore non é sicuro degli algoritmi che usa, egli piuttosto testa gli algoritmi attraverso continue modifiche (programmazione esplorativa) ed é quindi fortemente avvantaggiato da uno stile di programmazione che favorisca la modificabilità dei programmi.
Notiamo infine che si sono spesso vantate le doti della Programmazione Logica come strumento per il rapid prototyping e per la stesura di specifiche eseguibili, queste doti però vengono vanificate quando si trattano sistemi di vaste dimensioni (programming in the large) dalla mancanza di una medodologia (standardizzata) di sviluppo di programmi e di opportuni tools di supporto al programmatore per lo sviluppo di software affidabile, modulare e riusabile.

3. ANALISI DELLE CORRISPONDENZE

Esistono diversi approcci al problema della fusione del paradigma della Programmazione Logica ed il paradigma OOP.
Questi approcci possono essere grossolanamente classificati a seconda di come le caratteristiche (semantiche) componenti il paradigma OOP sono mappate in quelle (sintattiche) della Programmazione Logica. Componenti della Programmazione Logica sono termini, clausole, goals, variabili logiche, unificazione, etc. Componenti dell'OOP sono oggetti, classi, metodi, messaggi, ereditarietà, variabili di istanza, etc.

4. CLASSI DI OGGETTI COME TERMINI

Poichè gli Object Oriented Languages (OOL) sono definiti dall'esigenza fondamentale che gli oggetti siano supportati come una primitiva del linguaggio [17] il primo problema da risolvere diventa: cosa é l'equivalente di un oggetto in PROLOG?
Il modo canonico con cui gli OOL supportano ed organizzano gli oggetti, é tramite il concetto di classe (esistono delle eccezioni come ad esempio gli Actors di Hewitt).
Una classe é solitamente costituita da una parte di *specificazione* , che specifica le caratteristiche dichiarative (attributi) e procedurali (operazioni) di un insieme di oggetti; ed una parte di *realizzazione* che provvede all'implementazione della suddetta specificazione.

Se ci limitiamo a considerare l'aspetto dichiarativo di una classe di oggetti, come qualcosa che permette di rappresentare gli attributi di entità strutturate, un possibile candidato come controparte in PROLOG é il "termine

strutturato". Da questo punto di vista, le variabili eventualmente presenti in un termine strutturato permettono di specificare gli attributi caratteristici di una classe di oggetti con una funzione analoga agli *slots* (variabili di istanza) degli OOL: una loro instanziazione equivale alla specificazione di un particolare oggetto nella classe.

Ad esempio per rappresentare un conto in banca come oggetto, possiamo scrivere il termine PROLOG:

account('Roberto', 12377/1, 50.000)

e usarlo come argomento di predicati PROLOG in query del tipo

?- deposit(account('Roberto', 12377/1, 50.000), 1.137.000, Newaccount).

ove "deposit" é definito dalla clausola:

```
deposit(account(X,Y,Balance), Amount,
account(X,Y,NewBalance):-
        NewBalance is Balance + Amount.
```

Il termine *account(Owner,Number,Balance)* rappresenta l'aspetto dichiarativo di una classe di oggetti ma anche limitandoci a questo aspetto ci sono alcuni inconvenienti strettamente legati all'approccio precedente:

1) gli attributi degli oggetti/termini non sono accessibili per nome ma solo in base alla posizione che occupano: per utilizzarli é quindi necessario conoscere la struttura esatta del termine. Per oggetti caratterizzati da molti attributi o da una struttura complessa l'accesso per nome rende più facile al programmatore il compito di scrivere i corrispondenti metodi.

2) gli oggetti rappresentati devono avere un numero fisso di attributi o proprietà poichè i termini hanno un numero fisso di argomenti; non é possibile aggiungere informazioni addizionali senza modificare in maniera drastica la struttura del termine che li rappresenta.

3) assenza di un identificatore unico per gli oggetti: non possiamo, ad esempio, dichiarare:

account_1 = account('Roberto', 12377/1, 50.00)

e unificare la query:

?- deposit(account_1, 1.137.000, Newaccount).

con la testa della clausola che definisce *deposit* ; in pratica non é possibile distinguere oggetti che non sono completamente caratterizzati dall'insieme dei loro attributi (come due rettangoli della stessa dimensione). E' sempre possibile includere l'identificatore tra gli attributi di un oggetto, ma non potremo usarlo per referenziare l'oggetto stesso.

4.1. Implementazione di ADT : tecniche di astrazione

Gli inconvenienti evidenziati nei punti 1) e 2) ci portano a considerare il problema di come fornire delle tecniche di "astrazione" con cui maneggiare gli oggetti. Il concetto di astrazione riveste un ruolo molto importante nella filosofia OOP. Esso permette di "usare" gli oggetti senza dover dipendere dai loro particolari implementativi; quando si adottano gli oggetti come base per la decomposizione di un problema l'uso di tecniche di astrazione favorisce una maggiore chiarezza concettuale che permette di progettare e verificare una applicazione ad un livello di astrazione più alto, e porta ad una maggiore immunità dei programmi alle modifiche locali (da una indagine di Lientz e Swanson [10] risulta che il 17.5% del costo di manutenzione del software dipende da modifiche dei programmi dovute al cambiamento delle strutture di dati).

Aggiungere possibilità di astrazione sui dati ad un linguaggio richiede che sia considerato il delicato bilancio tra struttura e disciplina da un lato e flessibilità ed efficenza dall'altro. Le tecniche di astrazione sono effettivamente convenienti quando é possibile scegliere una particolare astrazione gia nelle fasi iniziali del processo di progettazione, ma possono risultare in una inutile limitazione quando non si é sicuri della precisa astrazione adatta al problema, e si intendono esplorare varie possibilità nel corso del processo di progettazione e prototyping. Questo, spesso, é il caso delle applicazioni di intelligenza artificiale o comunque delle applicazioni che richiedono la comprensione dei concetti sottostanti ad una classe di problemi piuttosto che la soluzione di uno specifico problema. Per questa ragione alcuni sistemi Object Oriented basati sul Lisp, e progettati per questo genere di applicazioni, non forniscono possibilità di astrazione allo scopo di migliorare la flessibilita concettuale del problem solving. Nel caso di PROLOG riteniamo che la sua semantica dichiarativa e la possibilità di stendere direttamente delle specifiche eseguibili, neutralizzano in parte gli svantaggi dell'astrazione. Implementare una nuova astrazione non é più costoso e, anzi, può coincidere con l'esplorare un nuovo punto di vista.

L'esigenza di adottare tecniche di astrazione richiede che l'accesso agli attributi di un oggetto sia mediato da una opportuna interfaccia verso l'esterno che permetta di astrarre dalla reale implementazione dell'oggetto stesso. Un meccanismo di questo tipo può essere realizzato usando degli opportuni predicati come selettori degli

attributi [1].
Nell'esempio precedente potremmo definire un predicato

fetch_balance(account(_,_,Balance),Balance).

per accedere all'attributo "balance".
Una volta definiti i selettori possiamo astrarre da modo in cui é rappresentato un oggetto. Per manipolare il valore dei singoli attributi dobbiamo solo conoscere il nome dei selettori corrispondenti ed usare questi nel resto del programma. L'uso di selettori é comodo quando si adottano delle rappresentazioni complesse e che possono essere soggette a modifiche: in quest'ultimo caso tutto ciò che bisogna fare é modificare la definizione dei selettori mentre il resto del programma rimarrà immutato.

Questa esigenza diventa ancora più forte quando si considera anche l'aspetto procedurale: l'insieme di operazioni che possono essere eseguite su/da un oggetto. Ancora, ogni operazione lecita deve essere mediata da una interfaccia in grado di filtrare gli aspetti rilevanti per un utilizzatore dell'oggetto ("cliente"), cioè gli aspetti comportamentali e funzionali, e astrarre da quelli irrilevanti, cioè dall'implementazione delle operazioni e delle strutture di dati locali su cui agiscono.

Un meccanismo per implementare la suddetta astrazione, che estende l'idea dei selettori, é stato proposto da C.Zaniolo nel contesto dell'approccio oggetti/termini [18].
L'operatore infisso predefinito "with" viene usato per specificare sia la parte dichiarativa che quella procedurale di un oggetto:

object *with* method_list. ove

object : termine prolog;
method_list : lista di termini interpretati come clausole PROLOG.

L'esempio precedente in questa implementazione diventa:

```
account(Owner, Number, Balance) with
      [( deposit(Amount, account(Owner, Number,
NewBalance)):-
        NewBalance is Balance + Amount ),
        ( fetch_owner(Owner) ),
        ( fetch_number(Number) ),
        ( fetch_balance(Balance) ),.....].
```

Una volta specificate in questo modo le proprietà degli oggetti (in realtà si tratta di classi di oggetti), si può interagire con un determinato oggetto (istanza) inviando-gli dei messaggi rappresentati dai termini che costituiscono la testa dei rispettivi metodi contenuti nella method_list.
A tale scopo ci si avvale dell'operatore infisso predefinito ":" ; un esempio di interazione può essere:

```
?- account('Roberto', 12377/1, 50.000) : deposit(1137000,X).
   X = account('Roberto', 12377/1, 1187000)

?- account('Roberto', 12377/1, 1187000) : fetch_balance(Y)
   Y = 1187000
```

L'implementazione del meccanismo di astrazione é affidata ad un preprocessor che "compila" le specificazioni degli oggetti in insiemi di clausole, una per ogni metodo nella method_list, in cui il termine che costituisce la testa di un metodo é stato "aumentato" aggiungendo ai suoi argomenti il termine che descrive la parte dichiarativa della classe di oggetti; analogamente le query contenenti l'operatore ":" vengono trasformate in query unificabili con le teste delle clausole così ottenute, "aumentando" in modo simile il termine che rappresenta il messaggio (interfaccia esterna).
Le clausole con cui lavora il sottostante interprete PROLOG e le rispettive query, sono alla fine del tipo:

```
deposit(account(Owner,Number,Balance),Amount,
                account(Owner,Number,NewBalance)):-
NewBalance is BalAmount.

?- deposit(account('Roberto', 12377/1, 50.000),
1137000,X).

fetch_balance(account(Owner,Number,Balance),Balance).

?-fetch_balance(account('Roberto',12377/1,1187000),
Y).
```

4.2. Modularizzazione ed information hiding

Nell'approccio OOP il concetto di astrazione si accompagna spesso in modo sinergico ai concetti di *modularità* ed *information hiding* . Col primo si intende il racchiudere in moduli, protetti da uno o più livelli di interfaccia astratta, le parti di codice che definiscono una o più classi correlate al fine di favorire lo sviluppo ed il debugging di programmi di vaste dimensioni (programming in the large). Col secondo si pone l'accento sulla proibizione di accedere alle informazioni contenute in un oggeto con modalità diverse da quelle consentite dalle suddette tecniche di astrazione.

Nell'approccio di Zaniolo entrambe queste caratteristiche non sono rispettate. Non c'é modularità poiche i metodi vengono "compilati" in un insieme di clausole raggruppate nel data base in procedure, cioè sulla base del nome del predicato/messaggio e non sulla base della classe di oggetti a cui i metodi si riferiscono. In presenza di *polimorfismo* per *overloading* si avranno all'interno della stessa procedura clausole che implementano metodi associati a classi differenti. I metodi/clausole sono infatti associati alle classi tramite il primo argomento del predicato che definiscono, la selezione della clausola da utilizzare in risposta ad un dato messaggio avviene tramite unificazione ed é affidata ad un dispatching sul primo argomento. Per quanto riguarda l'information hiding si osservi che é permesso un accesso diretto ai vari metodi senza usare il meccanismo del message passing ma semplicemente tramite una query alle clausole che li implementano; inoltre é necessario conoscere la struttura dichiarativa di un oggetto/termine per referenziarlo all'atto dell'invio di un messaggio.

Un diverso schema che realizza l'astrazione con un maggior grado di modularità e protezione (information hiding), é stato proposto da Furukawa et al. [2] nel linguaggio Himiko (non é un ambiente implementato "on top of PROLOG", ma un'estensione di PROLOG che supporta i meccanismi di modularizzazione e data abstraction come costrutti sintattici).
Himiko é un linguaggio tipizzato che si basa su una logica a molte sorte. Sono disponibili due generi di tipi di dato: i *types* edi *patterns* .
Un type é un termine strutturato (la cui struttura é specificata in maniera esplicita) incapsulato in un modulo e a cui é possibile accedere solo attraverso un determinato insieme di predicati la cui definizione é contenuta nello stesso modulo. Un type implementa un ADT.
Un pattern é un termine la cui struttura é visibile dall'esterno dei moduli.
Un programma é costituito da una gerarchia di moduli. Un modulo dal punto di vista semantico costituisce una porzione di una teoria; dal punto di vista sintattico é del tutto analogo alla definizione di una classe: consiste di una parte di specificazione (interfaccia) che specifica il nome dei tipi di dati ed il nome e il tipo di argomenti dei predicati accessibili dall'esterno e una parte di realizzazione che realizza l'implementazione degli ADT fornendo la struttura concreta dei termini che li rappresentano e l'implementazione delle loro operazioni sotto forma di clausole di Horn. (Per una analisi della differenza tra i concetti di tipo, ADT e classe rimandiamo a [17]).
Un modulo in Himiko può definire un gruppo di ADT simultaneamente (a differenza di una classe negli OOL) e le loro mutue relazioni.
La struttura tipica di un modulo é:

```
module <module name>
        interface
                type <n1>, <n2>, <n3>
                rel
                r1(<n1>, <n2>, <n3>)
                r2(<n1>, <integer>, <n2>)
                :
                :
        realization
                repr
                n1 = ...term structure...
                n2 = ...term structure...
                n3 = ...term structure...
        clause
                r1(...).
                r1(...) :- s1(...), s2(...).
                r2(...).
                r2(...) :- s3(...), s4(...).
                r2(...) :- s5(...), r2(...).
                :
                :
        end-of-module
```

Soltanto i predicati il cui nome é specificato nella parte **interface** (come r1 e r2) sono accessibili dall'esterno. Gli altri predicati definiti nel modulo (s1, s2, etc...) trovano un uso puramente locale al modulo stesso. Il type checking supporta un ulteriore livello di protezione: se ad esempio il tipo n1 é specificato solamente per argomenti di r1 e r2, qualsiasi tentativo di accedere direttamente da un altro modulo ad un termine di tipo n1, senza utilizzare r1 o r2, provoca un condizione di errore.
Di seguito é riportato un esempio di modulo che definisce l'ADT *queue* :

```
module queue
        interface
                type <queue>
                rel

                create_q(< queue>)

                en_q(<item>, <queue>, <queue>)
                de_q( <item>, <queue>, <queue>)

realization
        repr
        queue = <list>
        clause

                create_q([]).

                en_q(X, Q1, Q2) :- append(Q1, X, Q2).
                de_q( _, [], []).
```



```
                de_q( X, [X\Q], Q).
end-of-module
```

nell'esempio *item* é un tipo di base o predefinito, la definizione di *queue* può essere parametrica rispetto al tipo *item* ; *list* é un tipo di base e fornisce la struttura concreta di *queue* .

Altri ambienti PROLOG, quali Multilog, PrologII e Prolog2, supportano dei meccanismi di modularizzazione e di information hiding. Essi si basano sulla possibilità di definire porzioni di data base isolate (chiamate a seconda dei casi mondi o teorie) e caratterizzate da un insieme di predicati locali ed un insiema di predicati esportabili. Le tecniche di astrazione non sono supportate direttamente in questi ambienti ma vanno implementate dal programmatore. Vedremo che questi meccanismi di modularizzazione ed information hiding, specialmente se non sono abbinati ad un sisteme di tipi per i termini, favoriscono uno stile di decomposizione dei problemi e, quindi, dei programmi, basato sui moduli anzichè sui termini, portando ad associare i concetti di classe e oggetto a quello di porzione isolata di data base.

4.3. Unificazione ed uguaglianza

La potenza delle tecniche di astrazione nel manipolare oggetti definiti dal programmatore come termini strutturati, può essere ottenuta anche operando delle modifiche al meccanismo di Unificazione di PROLOG. E' questa una caratteristica comune a linguaggi come LM-Prolog [7], Tao [14], Uniform [5], Intermission [6].
L'Unificazione di termini di tipo primitivo quali i simboli , le liste, etc... é trattata in modo puramente sintattico come in PROLOG. L'Unificazione di oggetti definiti dall'utente provoca l'attivazione di particolari "metodi di Unificazione" in base ai quali gli oggetti possono unificare in accordo alle loro proprietà astratte. In pratica si ha l'implementazione di una Unificazione Semantica che utilizza le proprietà e le operazioni specificate per i vari oggetti: se, ad esempio, abbiamo definito le proprietà degli interi come ADT, possiamo unificare 5+X con 8 ottenendo l'instanziazione X=3; il messaggio di Unificazione implicitamente inviato ai due termini attiva il metodo che definisce l'addizione tra interi. In questo modo é possibile supportare a livello di linguaggio le potenzialità fornite dalle tecniche di astrazione degli OOL.

Un approccio simile seguito da Kornfeld [8] generalizza l'Unificazione attraverso dichiarazioni esplicite di Uguaglianza. Quando l'interprete PROLOG fallisce l'unificazione sintattica di due oggetti/termini A e B, tenta di soddisfare la goal:

equals(A, B).

utilizzando le dichiarazioni di Uguaglianza definite dal predicato *equals* . In caso di successo gli oggetti A e B unificano riportando l'opportuna instanziazione di vincoli.
In questo approccio le clausole che definiscono *equals* corrispondono a dei "metodi di Unificazione" ed i funtori dei termini corrispondono alle classi.
L'introduzione delle dichiarazioni di Uguaglianza permette anche di supportare direttamente l'ereditarietà tra classi di oggetti. Si consideri ad esempio l'insieme di regole :

*perimeter(reg_poly(Nsides,Length), P) :- P is Nsides * Length.*

*area(square(L), A) :- A is L * L.*

equals(reg_poly(4,L), square(L)).

la query:

?- perimeter(square(10), P).
richiede l'Unificazione di *square(10)* con *reg_poly(Nsides,Length)* ed ha successo in base alla dichiarazione *equals* .
In questo modo l'oggetto *square* ha ereditato il predicato *perimeter* dall'oggetto *reg_poly* .

Si osservi che l'ereditarietà supportata come abbiamo visto, o in modo simile, da meccanismi di Unificazione Semantica, ha una applicabilità piuttosto ristretta a causa della natura simmetrica dell'Uguaglianza e dell'Unificazione. Essa é appropriata con oggetti matematici per i quali si possono dare definizioni basate su condizioni necessarie e sufficenti usando l'Uguaglianza.Non può essere però usata nelle situazioni comuni dove la relazione di ereditarietà é asimmetrica.

4.4. Message passing e chiamata di procedura

A differenza dell'approccio di Zaniolo, in questi linguaggi, come anche in Himiko non é fornita una notazione esplicita per il message passing che é realizzato tramite l'Unificazione (sintattica in Himiko, semantica negli altri) e viene quindi a coincidere con l'usuale concetto di chiamata di procedura in PROLOG.

Noi riteniamo che il message passing, pur non essendo una proprietà necessaria, rinforzi l'idea di astrazione e più in generale l'approccio Object Oriented.
Dal punto di vista implementativo il message passing e la

chiamata di procedura sono equivalenti, nel senso che l'uno può essere implementato nell'altra [9] (lo abbiamo visto nell'approccio di Zaniolo). Dal punto di vista psicologico, Rentsch [15] ha sottolineato una sottile distinzione tra i due concetti: la chiamata di procedura denota un' azione, il message passing denota una richiesta; nel primo caso, il chiamante "controlla" il chiamato (attraverso un insieme di interfacce procedurali), nel secondo é lasciata al ricevente completa libertà sul modo di interpretare un messaggio. In PROLOG questa differenza é, forse, più sottile ancora, in quanto l'equivalente della chiamata di procedura, la query, può essere interpretata come una richiesta.
In effetti, riteniamo che la differenza fondamentale si possa cogliere analizzando la semantica associata al diverso ordine di valutazione. Si consideri ad esempio le due query:

?- account('Roberto', 12377/1, 50.000) : deposit(1137000, X). (a) e

?- deposit(account('Roberto', 12377/1, 50.000), 1137000, X). (b);

in generale l'ordine con cui le informazioi sono presentate (in una sequenza left to rigth) determina il modo in cui le entità nel dominio del discorso sono percepite, organizzate e classificate (che si tratti di un interprete umano o meccanico) [16]. La notazione (a) incoraggia una classificazione degli oggetti in base alle operazioni a loro comuni, focalizzando l'attenzione sugli oggetti; la notazione (b), invece, incoraggia una classificazione dello stesso dominio del discorso in base alle proprietà delle procedure o funzioni (tipo degli argomenti, ect...) ponendo queste al centro dell'universo.

5. OGGETTO = ADT + STATO

La classificazione object-oriented é più naturale quando i programmi sono definiti dalle loro strutture di dati e le operazioni applicabili ai dati possono variare durante il ciclo di vita dei dati stessi o addirittura nel corso di una computazione; al fine di supportare questa classificazione sono importanti le caratteristiche di "stato interno" e di "persistenza" degli oggetti: senza queste caratteristiche il message passing si riduce ad una "metafora della chiamata di procedura" [3].
In effetti gli esempi che abbiamo visto costituiscono diversi modi per implementare un ADT. Negli ADT non c'é il concetto di stato (nell'usuale approccio algebrico al I° ordine). L'implementazione di un ADT non permette di memorizzare dati (astratti) ma solo di manipolarli. L'utente passa i dati (astratti) ad una interfaccia ADT e, quindi, salva il risultato.

5.1. La persistenza degli oggetti

La persistenza é un concetto relativo piuttosto che assoluto. Un oggetto é persistente se la durata della sua "vita" é lunga relativamente alle operazioni che agiscono su di esso. Dal punto di vista della persistenza é possibile fare una classificazione sulla base della durata relativa delle operazioni e dei dati: nelle funzioni la persistenza delle operazioni é maggiore della persistenza dei dati, negli oggetti la persistenza dei dati é almeno uguale a quella delle operazioni. Se la vita degli oggetti é limitata é difficile realizzare una interazione flessibile tra gli oggetti e l'utente del sistema quale é, ad esempio, richiesta dall'implementazione di una interfaccia utente.

Negli OOL la persistenza di un oggetto é supportata da due proprietà: quella di possedere un identificatore e quella di possedere uno stato. Ciò garantisce che un oggetto possa al contempo "persistere" ed "evolversi" nel tempo. Abbiamo già visto come risulti innaturale dotare gli oggetti di un identificatore se si decide di rappresentare oggetti e classi di oggetti con termini PROLOG. Ancora più problematica é la rappresentazione dello stato di un oggetto secondo questo approccio.

5.1.1 .Le variabili logiche

Si é detto che le variabili logiche di un termine strutturato (nello schema oggetti/termini) possono essere considerate analoghe alle variabili di istanza; ciò é vero solo parzialmente. Negli OOL l'insieme di valori attribuiti alle variabili di istanza definiscono lo stato di un oggetto; questo non é possibile con le variabili logiche di PROLOG.
La differenza fondamentale con le variabili di istanza negli OOL, consiste nel fatto che le variabili logiche di PROLOG sono "write-once": una volta instanziate ad un valore all'atto della creazione di un oggetto o nel corso di una interazione (che abbia avuto successo) con un oggetto, non possono più essere instanziate ad un'altro valore (il backtracking automatico permette di disinstaziare delle variabili solo in caso di insuccesso), ne é possibile utilizzare il valore a cui é stata precedentemente instanziata una variabile , nel corso di un'altra interazione (i vincoli creati nel tentativo di soddisfare una goal vengono rilasciati a seguito del successo di questa). Ad esempio, con l'approccio oggetti/termini, un termine che rappresenta una data finestra sullo schermo non può rappresentare la stessa finestra che cambia di posizione o dimensioni (a meno che non venga ogni volta distrutto e ricreato).

La vita di un oggetto inteso come istanza di un termine dura il tempo di una singola chiamata di procedura poichè non é possibile salvare storie di computazioni o stati

intermedi su cui eseguire ulteriori operazioni.

5.2 Il problema dello stato

Senza l'idea di stato la stessa distinzione tra istanze e classe perde molto del suo significato (e quindi anche l'esigenza di un identificatore per gli oggetti): l'istanza si riduce ad un dispositivo di I/O per la classe.
Ciò che differenzia un oggetto da un altro, ovvero il diverso modo con cui rispondono ai medesimi messaggi, é determinato dal loro stato.

Per trattare lo stato nell'ambito di un approccio puramente logico oggetto/termine é stato suggerito il seguente stratagemma .
Un termine corrisponde allo stato di un oggetto ad un dato momento nel tempo. Un oggetto consiste della lista di questi stati, che rappresenta la storia dell'oggetto (history list). L'oggetto é referenziato tramite un puntatore alla history list. Lo stato corrente dell'oggetto é lo stato che compare immediatamente prima della coda correntemente non instanziata nella lista. Il suo reperimento avviene con tecniche di ricerca su lista ed ogni cambiamento dello stato é ottenuto instanziando ulteriormente la coda della history list. Da un punto di vista logico, più che un cambiamento dello stato di un oggetto si ha una progressiva scoperta della sua storia computazionale. La semantica di questo approccio é sospetta anche perchè per simulare lo stato si deve usare il predicato metalogico *var* al fine di rilevare ciò che altrimenti sarebbe un side effect nascosto nell'Unificazione. Infine la persistenza di un oggetto in questo schema non é maggiore della persistenza di una chiamata di procedura.

Riteniamo che in PROLOG "puro" (senza *assert* , *retract* ed altre caratteristiche extra logiche) non sia possibile implementare lo stato in maniera soddisfacente; ciò in generale é vero per una logica del I° ordine qualora non si usino operatori modali o riflessivi (per una discussione più approfondita si veda [3]).

5.2.1 Oggetti come porzioni di Data Base

Agli inconvenienti che derivano dalla scelta di usare gli oggetti strutturati come base per la programmazione ad oggetti ed in particolare all'impossibilità di rappresentare lo stato di un oggetto, si può ovviare adottando un diverso schema nel mappare le componenti caratteristiche dell'OOP nelle componeneti dalla Programmazione Logica.

Un modo diretto per realizzare questo mapping consiste nel rappresentare la sua struttura dichiarativa (attributi) con un insieme di fatti nel data base, e la parte procedurale (metodi) con un insieme di clausole che possono accedere a tali fatti chiamandoli come sottogoal nei loro bodies.
In questa rappresentazione l'esempio dell'account diventa:

```
isa(account_1, account).

owner(account_1, 'Roberto').

number(account_1, 12377/1).

balance(account_1, 50.000).


fetch_balance(Id, Balance) :-
        isa(Id,account),
        balance(Id, Balance).

deposit(Id, Amount) :-
        fetch_balance(Id, Balance),
        NewBalance is Balance + Amount,
        retract(balance(Id Balance)),
        assert(balance(Id, NewBalance)).
```

La costante *account_1* che collega gli attributi di un singolo oggetto costituisce l'identificatore unico con cui é possibile referenziarlo.
E' possibile un semplice accesso per nome agli attributi utilizzando il nome dei predicati che li rappresentano: *balance, owner, number*.
E' inoltre possibile, in ogni momento, fornire degli attributi addizionali asserendo dei nuovi fatti nel data base:

```
assert(facilities(account_1, bancomat)).
assert(history(account_1, History_list)).
```

o raffinare attributi primitivi in oggetti strutturati:

```
isa('Roberto', human).
surname('Roberto', 'Grande').
address('Roberto', 'Via_Assietta_9').
employ('Roberto', free_lance).
```

L'insieme dei fatti relativi ad un determinato identificatore, presenti nel data base fornisce lo stato attuale dell'oggetto referenziato dall'identificatore.
Non tutti i valori degli attributi devono essere asseriti con dei fatti, alcuni possono essere calcolati (inferiti) a partire da altri attributi; ciò che conta é che l'insieme dei fatti sia sufficente a definire uno stato. Una classe é definita dall'insieme di attributi/predicati e dai metodi/clausole che li utilizzano; tra questi dovrà essere presente un metodo per la creazione di nuove istanze, ad esempio:

```
create_instance(Id, [(Attribbute,Value)/ List_Attr_Val])
```

```
                    X=..[Attribute, Id, Value],
                    assert(X),
                    create_istance(Id,
List_Attr_Val).

create_istance(Id, []).
```

Il maggiore svantaggio di questo approccio é da imputarsi all'uso pesante che viene fatto dei built in extralogici *assert* e *retract* per cambiare il valore degli attributi e per creare nuove istanze.
Un secondo aspetto negativo é dovuto alla mancanza di possibilità di encapsulation e di information hiding. Poichè tutti gli oggetti condividono lo stesso data base non protetto, ogni programma può accedere o modificare lo stato di un oggetto senza usare il meccanismo del message passing. (Un approccio simile a quello descritto é stato proposto da [4]).
In termini puramente logici, potremmo dire che una teoria (intesa come porzione di data base) rappresenta la fotografia di un oggetto in un determinato stato (trascurando esplicitamente la variabile tempo); il cambiamento di stato corrisponde all'azione di un opportuno operatore modale che fa passare da una teoria ad un 'altra.
Affinché la persistenza dell'oggetto sia garantita anche a livello semantico (e non solamente dalla presenza di uno stesso identificatore), bisogna risolvere problemi di consistenza e completezza che possono sorgere dall'applicazione di operatori modali ad una base di conoscenza. In pratica si tratta di disciplinare l'uso di *assert* e *retract* incapsulandoli in opportune procedure (col ruolo di interfacce astratte) atte a garantire che la modifica di attributi o la creazione di nuovi oggetti sia immune da side effect indesiderati.

Per ottenere un ulteriore livello di protezione é necessario che sia supportato a livello sintattico un meccanismo di encapsulation e modularizzation che consenta di avere data base separati per ogni distinta classe di oggetti. Abbiamo già citato alcuni sistemi PROLOG che in diversa misura forniscono un tale supporto. In questa direzione va anche considerato l'approccio di Nakashima in Prolog/KR [13]. Egli propone una estensione di PROLOG che supporta direttamente possibilità di modularizzazione per rappresentare gerarchie di oggetti. In Prolog/KR i moduli sono detti "multiple worlds" o "contesti". Un contesto é un insieme di clausole di Horn. Ogni oggetto, nella rappresentazione di Nakashima , é definito dal proprio contesto. Ad esempio per rappresentare l'account visto negli esempi precedenti si deve creare un contesto di nome *account_1* e asserire i fatti (Prolog/KR usa una sintassi Lisp like):

```
(with account_1
        (assert (owner 'Roberto'))
        (assert (number 12377/1))
        (assert (balance 50.000)))
```

Mentre prima rappresentavamo le proprietà di molti oggetti in un singolo data base per mezzo di tuple Attributo-Identificatore di Oggetto-Valore, Prolog/KR riserva un data base separato per ogni oggetto e rappresenta le proprietà dell'oggetto con tuple Attributo-Valore.
Il message passing é ottenuto per mezzo del predicato *with*:

```
(with Receiver Message)
```

che richiede la valutazione della goal *Message* nell'ambito del contesto indicato da *Receiver*.

Tramite il nesting dei contesti é anche possibile supportare direttamente l'ereditarietà. Ad esempio con

```
(with A (with B p))
```

viene eseguito p usando le definizioni di A e B. Più precisamente se non é stata trovata la definizione di p in B essa viene cercata in A ed é aggiunta in coda a B formando così il data base corrente con cui valutare p. Si noti che, nell'esempio, il nesting dei contesti é determinato dinamicamente a run time. La possibilità di costruire gerarchie di contesti in modo dinamico e controllato da programma costituisce un meccanismo molto potente per controllare l'ereditarietà, che solitamente non é disponibile negli OOL.
Qualora la gerarchia degli oggetti é statica, é possibile specificarla in fase di definizione dei contesti per non doverla riportare ad ogni invocazione di goal. Sono inoltre disponibili dei meccanismi di "overriding" e di inibizione dell'ereditarietà

L'inconveniente principale nell'approccio di Nakashima é dovuto alla mancanza di una chiara distinzione tra oggetti e classi. Le "foglie" in una gerarchia di contesti sono considerate degli oggetti individuali. Questi oggetti individuali possono però essere usati come "contesti schema" per creare nuovi individui ad un livello più basso nella gerarchia. Quando ciò accade diventa problematico stabilire se i contesti usati come schema non devono più essere considerati oggetti individuali (cosa che dovrebbe essere una proprietà intrinseca dell'oggetto).
In Prolog/KR sono forniti anche dei costrutti per supportare varie tecniche di astrazione in base ai quali é possibile implementare tutte le caratteristiche di una interfaccia ADT (per una trattazione più dettagliata rimandiamo a [12]).

6. CENNI ALL'EREDITARIETÀ

Per quanto riguarda l'ereditarietà tra classi (una caratteristica fondamentale in molti linguaggi OOL) ci siamo limitati a trattarla in quei casi in cui viene supportata direttamente a livello dei costrutti sintattici. E' il caso dei meccanismi di Unificazione Semantica e delle dichiarazioni di Uguaglianza in Uniform e in Kornfeld o del nesting di contesti multipli in Prolog/KR. Aggiungiamo qui che, generalmente , l'ereditarietà presenta problemi più dal punto di vista implementativo e della compatibilità semantica con un linguaggio che non da quello della compatibilità sintattica (se può essere supportato dai costrutti del linguaggio); essa infatti viene solitamente implementata al livello superiore di ambiente. Ci riserviamo di discutere questi problemi in un articolo successivo.

7. CONCLUSIONI

Abbiamo considerato secondo varie dimensioni i pro ed i contro di alcune possibili scelte implementative di uno stile di programmazione ad oggetti in PROLOG. In questa analisi abbiamo spesso mantenuto artificiosamente separati i vari aspetti che concorrono a costituire un linguaggio di programmazione. L'analisi esaustiva delle sinergie e delle interrelazioni tra questi aspetti e soprattutto la valutazione dei benefici e degli inconvenienti, porterebbe al compito della progettazione di un nuovo ambiente per la Programmazione Logica Object Oriented; un tale intento va di la dagli scopi del presente articolo e soprattutto andrebbe supportato da un lavoro di sperimentazione e realizzazione.

8. BIBLIOGRAFIA

[1] Bratko, PROLOG PROGRAMMING FOR A.I., Addison Wesley, 1983.

[2] Furukawa et al., MODULARIZZATION AND ABSTRACTION IN LOGIC PROGRAMMING, New Generation Computing 1, 1983.

[3] Goguen, UNIFYING FUNCTIONAL, OBJECT-ORIENTED AND RELATIONAL PROGRAMMING WITH LOGICAL SEMANTICS, SRI-CSL-87-7 - July 1987.

[4] Gullichsen, BIGGERTALK: OBJECT ORIENTED PROLOG, STP-125-85, MCC-STP, Austin, TX, Nov 1985.

[5] Kahn, UNIFORM - A LANGUAGE BASED UPON UNIFICATION WHICH UNIFIES (MUCH OF) LISP, PROLOG AND ACT1, IJCAI-7 1981.

[6] Kahn, INTERMISSION: ACTORS IN PROLOG, "Logic Programming", (K.L.Clark e S.-A.Tarnlund eds.), Academic Press 1982.

[7] Kahn, HOW TO IMPLEMENT PROLOG ON A LISP MACHINE, "Implementation of PROLOG" (J.A Campbell ed.), Ellis Horwood 1984.

[8] Kornfeld, EQUALITY FOR PROLOG, IJCAI-8 1983.

[9] Lauer, Needham, ON THE DUALITY OF OPERATING SYSTEM STRUCTURE, 2nd International Symposium on Operating Systems, IRIA Oct. 1978.

[10] Lienz, Swanson, SOFTWARE MAINTENANCE: A USER/MANAGEMENT TUG OF WAR, Data Management, Apr. 1979.

[11] Meyer, OBJECT ORIENTED SOFTWARE CONSTRUCTION, Printice Hall, 1988.

[12] Nakashima, DATA ABSTRACTION IN PROLOG/KR, New Generation Computing 1, 1983.

[13] Nakashima, KNOWLEDGE REPRESENTATION IN PROLOG/KR, International Symposium on Logic Programming, 1984.

[14] Okuno, A FAST INTERPRETER-CENTERED ON LISP MACHINE ELIS, Proceedings of 1984 Lisp and Functional ProgrammingConference, 1984.

[15] Rentsch, OBJECT ORIENTED PROGRAMMING, Sigplan Notices, Sept. 1982.

[16] Wegner, PERSPECTIVE IN OBJECT ORIENTED PROGRAMMING, CRAI, Spring International Seminar Jun. 1988.

[17] Wegner, OBJECT-ORIENTED CONCEPT HIERARCHIES, CRAI, Spring International Seminar, Jun. 1988.

[18] Zaniolo, OBIECT ORIENTED PROGRAMMING IN PROLOG, International Symposium on Logic Programming, 1984.

TRANSLATING PROGRAMS INTO DELAY-INSENSITIVE CIRCUITS*

Jo C. Ebergen
Department of Computer Science and Mathematics
Eindhoven University of Technology

RIASSUNTO

Questo lavoro presenta i circuiti delay - insensitive, ovvero componenti il cui comportamento funzionale é indipendente dai ritardi nella comunicazione o negli elementi.
Dopo aver rivisto la storia e la terminologia, l'articolo dà le motivazioni fondamentali per lavorare con questo tipo di circuiti e introduce brevemente un metodo per la loro progettazione.

ABSTRACT

This paper presents delay - insensitive circuits, i.e. components whose functional operations are insensitive to delays in wires or elements. After revieming history and terminology, the paper gives motivations to work with delay insensitive circuits and briefly sketches a method to design these circuits.

1. INTRODUCTION

In 1938 Claude E. Shannon wrote his seminal paper [23] entitled "A Symbolic Analysis of Relay and Switching Circuits". He demonstrated that Boolean algebra could be used elegantly in the design of switching circuits. The idea was to specify a circuit by a set of Boolean equations, to manipulate these equations by means of a calculus, and to realize this specification by a connection of basic elements. The result was that only a few basic elements, or even one element such as the 2-input NAND gate, suffice to synthesize any switching function specified by a set of Boolean equations. Shannon's idea proved to be very fertile and out of it grew a complete theory, called switching theory, which is used by most circuit designers nowadays.

In the thesis [5] a method is presented for designing *delay-insensitive circuits*. (Operationally speaking, a delay-insensitive circuit is a connection of basic elements whose functional operation is insensitive to delays in wires or elements). The principal idea of this method is similar to that in Shannon's paper: to design a circuit as a connection of basic elements and to construct this connection with the aid of a formalism. The method of constructing such a circuit, as described in [5], is by translating programs satisfying a certain syntax. The result of such a translation is a delay-insensitive connection of elements chosen from a finite set of basic elements. Moreover, this translation has the property that the number of basic elements in the connection is proportional to the length of the program. Furthermore, in [5] a rigorous formalization is given of what it means for such a connection to be delay-insensitive.

In this paper[1] we briefly describe some of the history of designin delay-insensitive circuits and some of the reasons why we would like to design delay-insensitive circuits. By means of an example we convey the idea of designing delay-insensitive circuits. We conclude with an outline of the method described in [5].

* Reprinted from
CWI Newsletter, Issue N. 15, June 1987

1. The research reported in this paper was carried out while the author was working at CWI in Amsterdam.

2. SOME HISTORY

Delay-insensitive circuits are a special type of circuits. We briefly describe their origins and how they are related to other types of circuits and design techniques. The most common distinction usually made between types of circuits is the distinction between *synchronous circuits and asynchronous circuits*. Synchronous circuits are circuits that perform their (sequential) computations based on the successive pulses of the clock. From the time of the first computer designs many designers have chosen to build a computer with synchronous circuits.
In [25] Alan Turing, one of the first computer designers, has motivated this choice as follows:

We might say that the clock enables us to introduce a discreteness into time, so that time for some purposes can be regarded as a succession of instants instead of a continuous flow. A digital machine must essentially deal with discrete objects, and in the case of the ACE (Automatic Computing Engine) this is made possible by the use of a clock. All other digital computing machines except for human and other brains that I know of do the same. One can think up ways of avoiding it, but they are very awkward.

In the past fifty years many techniques for the design of synchronous circuits have been developed and are described by means of switching theory [11, 15]. The correctness of synchronous systems relies on the bounds of delays in elements and wires. The satisfaction of these delay requirements cannot be guaranteed under all circumstances, and for this reason problems can crop up in the design of synchronous systems. (Some of these problems are described in the next section.) In order to avoid these problems interest arose in the design of circuits without a clock. Such circuits have generally been called *asynchronous* circuits.

The design of asynchronous circuits has always been and still is a difficult subject. Several techniques for the design of such circuits have been developed and are discussed in, for example, [11, 15, 28]. For special types of such circuits formalizations and other design techniques have been proposed and discussed. David E. Muller has given a formalization of a type of circuits which he coined by the name of *speed-independent* circuits. An account of this formalization is given in [16]. From a design discipline that was applied in the Macro-modules project [3, 4] at Washington University in St. Louis, the concept of a special type of circuit evolved which was given tile name *delay-insensitive* circuit. It was realized that a proper formalization of this concept was needed in order to specify and design such circuits in a well-defined manner. A formalization of the concept of a delay-insensitive circuit was later given in [26]. For the design and specification of delay-insensitive circuits several methods have been developed based on, for example, Petri Nets and techniques derived from switching theory [17].

Another name that is frequently used in the design of asynchronous circuits is *self-timed systems*. This name was introduced by C. L. Seitz in [22] in order to describe a method of system design without making any reference to timing except in the design of the self-timed elements. Recently, Alain Martin has proposed some interesting and promising design techniques for circuits of which the functional operation is unaffected by delays in elements or wires [12, 13]. His techniques are based on the compilation of CSP-like programs into connections of basic elements. The techniques presented in [5] exhibit a similarity with the techniques applied by Alain Martin.

3. WHY DELAY-INSENSITIVE CIRCUITS?

The reasons for designing delay-insensitive systems are manifold. One reason why there has always been an interest in asynchronous systems is that synchronous systems tend to reflect a *worst-case* behavior, while asynchronous systems tend to reflect an *average-case* behavior. A synchronous system is divided into several parts, each of which performs a specific computation. At a certain clock pulse, input data are sent to each of these parts and at the next clock pulse the output data, i.e. the results of the computations, are sampled and sent to the next parts. The correct operation of such an organization is established by making the clock period larger than the worst-case delay for any subcomputation. Accordingly, this worst-case behavior may be disadvantageous in comparison with the average-case behavior of asynchronous systems.

Another more important reason for designing delay-insensitive systems is the so-called *glitch phenomenon*. A glitch is the occurrence of metastable behavior in circuits. Any computer circuit that has a number of stable states also has metastable states. When such a circuit gets into a metastable state, it can remain there for an indefinite period of time before it resolves into a stable state. For example, it may stay in the metastable state for a period larger than the clock period. Consequently, when a glitch

occurs in a synchronous system, erroneous data may be sampled at the time of the clock pulses. In a delay-insensitive system it does not matter whether a glitch occurs: the computation is delayed until the metastable behavior has disappeared and the element has resolved into a stable state. One frequent cause for glitches are, for example, the asynchronous communications between independently clocked parts of a system.

The first mention of the glitch problem appears to date back to 1952 (cf. [1]). The first publication of experimental results of the glitch problem and a broad recognition of the fundamental nature of the problem came only after 1973 [2, 8] due to the pioneering work on this phenomenon at the Washington University in St. Louis.

A third reason is due to the effects of *scaling*. This phenomenon became prominent with the advent of integrated circuit technology. Because of the improvements of this technology, circuits could be made smaller and smaller. It turned out, however, that if all characteristic dimensions of a circuit are scaled down by a certain factor, including the clock period, delays in long wires do not scale down proportional to the clock period [13, 21]. As a consequence, some VLSI designs when scaled down may no longer work properly anymore, because delays for some computations have become larger than the clock period. Delay-insensitive systems do not have to suffer from this phenomenon if the basic elements are chosen small enough so that the effects of scaling are negligible with respect to the functional behavior of these elements [24].

A fourth reason is the clear separation between functional and physical correctness concerns that can be applied in the design of delay-insensitive systems. The correctness of the behavior of basic elements is proved by means of physical principles only. The correctness of the behavior of connections of basic elements is proved by mathematical principles only. Thus, it is in the design of the basic elements only that considerations with respect to delays in wires play a role. In the design of a connection of basic elements no reference to delays in wires or elements is made. This does not hold for synchronous systems where the functional correctness of a circuit also depends on timing considerations. For example, for a synchronous system one has to calculate the worst-case delay for each part of the system and for any computation in order to satisfy the requirement that this delay must be smaller than the clock period.

As a last reason, we believe that the translation of parallel programs into delay-insensitive circuits offers a number of advantages compared to the translation of parallel programs into synchronous systems. In [5] a method is presented with which the synchronization and communication between parallel parts of a system can be programmed and realized in a natural way.

4. AN EXAMPLE

In order to get an idea of designing delay-insensitive circuits we describe in an informal way a small example. Consider the modulo-3 counter specified by the following communication behavior. The modulo-3 counter has three communication actions: one input, denoted by $a?$, and two outputs, denoted by $p!$ and $q!$. The communication behavior is an alternation of inputs and outputs, starting with an input. The outputs depend on the inputs as follows. After the n-th input, where $n > 0$, if n **mod** $3 \neq 0$, then output $q!$ is produced, else output $p!$ is produced. This behavior is expressed in the following program, or so-called command,

$$E0 = \textbf{pref}\,[a?;q!;a?;q!;a?;p!].$$

Here,$[E]$ denotes repetition of the enclosed behavior E and $E1$; $E2$ denotes concatenation of $E1$ and $E2$. The notation **pref**E denotes the prefix-closure of the behavior E, i.e. if the string of symbols (also called *trace*) $a?\ q!\ a?\ q!\ a?\ p!$ is a possible behavior of E, then also each prefix of this trace is a possible behavior of **pref**E.
The following physical interpretation may be associated with the symbols. With each symbol corresponds a terminal of the circuit and with each occurrence of that symbol in a trace corresponds a voltage transition (either a high-going or a low-going transition) in that terminal. Voltage transitions corresponding to inputs are caused by the environment of the circuit; voltage transitions corresponding to outputs are caused by the circuit itself.
In the same way the basic TOGGLE and XOR component can be specified as given in Figure 1.

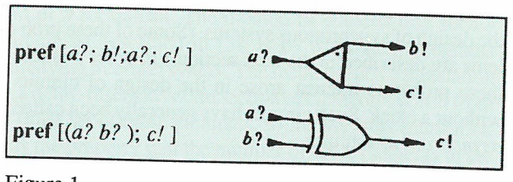

Figure 1

The first component is the TOGGLE component and can be considered as a modulo -2 counter. The second component is an XOR component and has the following repetitive behavior. First, the environment provides either an input $a?$ or an input $b?$ (the | separates the alternatives), and then the component produces an output $c!$. After the environment has received an output $c!$ it may produce a

new input again, and so on. (Notice that the behaviors of components are specified as orderings of events instead of as logical functions.) We emphasize that all specifications must be understood as prescriptions for the behavior of the component *and* environment. Consequently, in constructing a decomposition for the modulo-3 counter E0 we assume that the environment satisfies the prescribed behavior in E0, i.e. the environment provides new inputs $a?$ only when an output has been received. Under this assumption the modulo-3 counter can be decomposed as depicted in Figure 2.

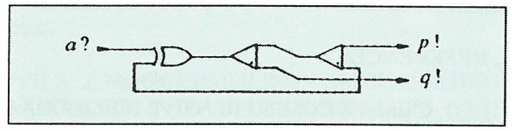

Figure 2

Notice that in the decomposition the prescription for the environment of every basic component is not violated. Without much difficulty we can convince ourselves that the functional behavior of this decomposition is unaffected by delays in connection wires or in basic elements. In other words, we could say that the modulo-3 counter is realized by a delay-insensitive connection of basic elements. Knowing how to construct a modulo-3 counter the reader may try, as an exercise, to construct a modulo-17 counter from TOGGLE and XOR components. (There exist several solutions, some more efficient than others.)

5. OUTLINE OF THE METHOD

The method presented in [5] for designing delay-insensitive circuits is briefly described as follows. An abstraction of a circuit is called a component; *components* are specified by programs written in a notation based *on trace theory*. Trace theory was inspired by Hoare's CSP [6, 7] and developed by a number of people at the University of Technology in Eindhoven. It has proven to be a good tool in reasoning about parallel computations [18, 19, 24, 9] and, in particular, about delay-insensitive circuits [10, 20, 21, 26, 27].

The programs are called *commands* and can be considered as an extension of the notation for regular expressions. Any component represented by a command can also be represented by a regular expression, i.e. it is also a *regular* component. The notation for commands, however, allows for a more concise representation of a component due to the additional programming primitives in this notation. These extra programming primitives include operations to express parallelism, tail recursion (for representing finite state machines), and projection (for introducing internal symbols).

Based on trace theory the concepts of *decomposition* of a component and of *delay-insensitivity* are formalized. The decomposition of a component is intended to represent the realization of that component by means of a connection of circuits. Several theorems are presented that are helpful in finding decompositions of a component. Delay-insensitivity is formalized by the definition of DI *decomposition*. A DI decomposition represents a realization of a component by means of a *delay-insensitive* connection of circuits. In order to link decomposition and DI decomposition, the definition of a DI component is introduced. Operationally speaking a DI component represents a circuit for which the communication between circuit and environment takes place in a delay-insensitive way. (It turns out that the definition of a DI component is equivalent with Udding's formalization of a delay-insensitive circuit.) By means of the definition of a DI component one of the fundamental theorems in the thesis can be formulated as follows: DI decomposition and decomposition are equivalent if all components involved are DI components.

This theorem is applied as follows to the example described in the previous section. We showed, informally, that the modulo-3 counter can be decomposed into TOGGLE and XOR components. Furthermore, we have that the TOGGLE component, XOR component, and modulo-3 counter $E0$ are DI components. Consequently, it follows by the above mentioned theorem that the decomposition of Figure 2 forms a DI decomposition of the modulo-3 counter.
Because of the above mentioned theorem, it is important to have techniques to recognize DI components. For this purpose a number of so-called *DI grammars* are developed, i.e. grammars for which any command generated by these grammars represents a (regular) DI component.
Based on these grammars syntax-directed translations of commands into DI decompositions of components represented by these commands are developed. With these grammars the language L_4 of commands is defined. It is shown that any regular DI component represented by a command in the language L_4 can be decomposed in a syntax-directed way into the finite set B of basic DI components and so-called *CAL components*. CAL components are also DI components. Consequently, since all components involved are DI components, the decomposition into these components is, by the above theorem, also a DI decomposition.

The set of all CAL components is, however, not finite. In order to show that a decomposition into a finite basis of components exists, two decompositions of CAL compo-

nents are discussed: one decomposition into the finite basis B0 and one decomposition into the finite basis B1 The decomposition of CAL components into the basis B1 is in general *not* a DI decomposition, since not every component in B1 is a DI component. This decomposition, however, is in general simpler than the decomposition into B0 and can be realized in a simple way if so-called *isochronic forks* are used in the realization. The decomposition of CAL components into the basis B0 is an interesting but difficult subject. Since every component in B0 is a DI component, every decomposition into B0 is therefore also a DI decomposition. In [5] a general procedure for the decomposition of CAL components into the basis B0 is described, which is conjectured to be correct.

The complete decomposition method can be described as a syntax-directed translation of commands in L_4 into commands of the basic components in B0 or B1. Consequently, the decomposition method is a constructive method and can be completely automated: as soon as we have a specification of a component expressed as a command in L_4we can find mechanically a decomposition of this component into B0 or B1. Moreover, it is shown that the result of the complete decomposition of any component expressed in L_4 can be linear in the length of the command, i.e. the number of basic elements in the resulting connection is proportional to the length of the command.

Although many regular DI components can be expressed in the language L_4, which is the starting point of the translation method, probably not every regular DI component can be expressed in this way. Nevertheless, it is also shown that for any regular DI component there exists a decomposition into components expressed in L_4, which can then be translated by the method presented.

6. CONCLUDING REMARKS

The research described in [5] has been fascinating and many-sided. It includes, for example, aspects of
- Language design: which programming primitives do we include in the language in order to be able to present a clear and concise program for a component?
- Programming methodology: do there exist techniques to design programs from specifications for components in the language of commands?
- Translation techniques: how do we translate programs into connections of basic elements?
- Syntax and semantics: how we can satisfy semantic properties (like a DI component) by imposing syntactic requirements on programs?
- VLSI design: what physical constraints must be met in order to realize the circuit designs obtained in the VLSI medium?

In the thesis the aim of delay-insensitive design has been pursued as far as possible, i.e. correctness arguments based on delay-assumptions have been postponed as far as possible, in order to see what sort of designs such a pursuit would lead to. In this approach our first concern has been the correctiiess of the designs and only in the second place have we addressed their efficiency. Accordingly, although the number of basic components is already proportional to the length of the program, still many, optimizations are possible in translating programs into delay-insensitive circuits.

7. REFERENCES

[1] T. J. Chaney, A COMPREHENSIVE BIBLIOGRAPHY ON SYNCHRONIZERS AND ARBITERS, Technical Memorandum No. 306C, Institute for Biomedical Computing, Washington University, St. Louis, 1980

[2] T. J. Chaney, C.E. Molnar, ANOMALOUS BEHAVIOR OF SYNCHRONIZER AND ARBITER CIRCUITS. IEEE Transactions on Computers C-22, 421-422, 1973

[3]W.A. Clark, MACROMODULAR COMPUTER SYSTEMS, Proceedings of the Spring Joint Computer Conference, AFIPS, 1967

[4]W.A. Clark, C.E. Molnar, MACROMODULAR COMPUTER SYSTEMS, R. Stacy, B. Waxman (eds.), Computers in Biomedical Research, Vol. IV, Academic Press, New York, 1974

[5]J. C. Ebergen. TRANSLATING PROGRAMS INTO DELAY-INSENSITIVE CIRCUITS, Ph. D. Thesis, Eindhoven University of Technology, 1987

[6] C.A.R. Hoare, COMMUNICATING SEQUENTIAL PROCESSES, Communications of the ACM 21, 666-677 1978

[7] C.A.R. Hoare, COMMUNICATING SEQUENTIAL PROCESSES, Prentice-Hall, 1985

[8] M. Hurtado, DYNAMIC STRUCTURE AND PERFORMANCE OF ASYMPTOTICALLY BISTABLE SYSTEMS, D. Sc. Dissertation, Washington University, St. Louis, 1975

[9] Kaldewaij, A FORMALISM FOR CONCURRENT PROCESSES, Ph. D. Thesis, Eindhoven University of Technology, 1986

[10] Kaldewaij, THE TRANSLATION OF PROCESSES INTO CIRCUITS, J.W. De Bakker, A.J. Nijman, P.C. Treleaven (eds.). Proceedings PARLE, Parallel Architectures and Languages Europe, Vol. 1, Springer LNCS, 195- 213, 1987

[11] Zvi Kohavi, SWITCHING AND FINITE AUTOMATA THEORY, McGraw-Hill, 1970

[12] A. J. Martin. THE DESIGN OF A SELF-TIMED CIRCUIT FOR DISTRIBUTED MUTUAL EXCLUSION. H. Fuchs (ed.). Proceedings 1985 Chapel Hill Conference on VLSI, Computer Science Press, 247-260, 1985

[13] A. J. Martin, COMPILING COMMUNICATING PROCESSES INTO DELAY-INSENSITIVE VLSI CIRCUITS, Distributed Computing 1, 226-234, 1986

[14] C. Mead, M. Rem, MINIMUM PROPAGATION DELAYS IN VLSI. IEEE Journal of Solid-State Circuits SC-17, 773-775, 1982

[15] R.E. Miller, SWITCHING THEORY, Wiley, 1965

[16] R.E. Miller, Chapter 10 in: [15]

[17] C.E. Molnar, T.P. Fang, F.U. Rosenberger, SYNTHESIS OF DELAY-INSENSITIVE MODULES. H. Fuchs (ed.), Proceedings 1985 Chapel Hill Conference on VLSI, Computer Science Press, 67-86, 1985

[18] M. Rem, CONCURRENT COMPUTATIONS AND VLSI CIRCUITS. M. Broy (ed.), Control Flow and Data Flow: Concepts of Distributed Computing, Springer-Verlag, 399-437, 1985

[19] Martin Rem. TRACE THEORY AND SYSTOLIC COMPUTATIONS. J.W. De Bakker, A. J. Nijman, P.C. Treleaven (eds.). Proceedings PARLE, Parallel Architectures and Languages Europe, Vol. I, Springer LNCS, 14- 34, 1987

[20] H. M.J.L. Schols, A FORMALISATION OF THE FOAM RUBBER WRAPPER PRINCIPLE, Master's Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1985

[21] H. M.J.L. Schols, T. Verhoeff, DELAY-INSENSITIVE DIRECTED TRACE STRUCTURES SATISFY THE FOAM RUBBER WRAPPER POSTULATE, Computing Science Notes 85/04, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1985

[22] C.L. Seitz, SYSTEM TIMING. CARVER MEAD, LYNN CONWAY, Introduction to VLSI systems, Addison-Wesley, 218-262, 1980

[23] C. E. Shannon, A SYMBOLIC ANLYSIS OF RELAY AND SWITCHING CIRCUITS, Trans. AIEE, 57, 713-723, 1938

[24] J. L.A. Van De Snepscheut. TRACE THEORY AND VLSI DESIGN, LNCS 200, Springer-Verlag, 1985

[25] Alan M. Turing, LECTURE TO THE LONDON MATHEMATICAL SOCIETY on 20 February 1947. B.E. Carpentar, R.W. Doran (eds.), Charles Babbage Institute Reprint Series for the History of Conputing, Vol. 10, MIT Press, 1986

[26] J. T. Udding, CLASSIFICATION AND COMPOSITION OF DELAY- INSENSITIVE CIRCUITS, Ph.D. Thesis, Eindhoven University of Technology, 1984

[27] J. T. Udding, A FORMAL MODEL FOR DEFINING AND CLASSIFYING DELAY-INSENSITIVE CIRCUITS AND SYSTEMS, Distributed Computing 1, 197-204, 1986

[28] S.H. Unger, ASYNCHRONOUS SEQUENTIAL SWITCHING CIRCUITS, Wiley Interscience, 1969

STAGING OF LUNG CANCER - AN EXAMPLE OF A MICROCOMPUTER-PHYSICIAN INTERFACE FOR SURGICAL DECISIONS

Richard M. Peters
Department of Surgery
University of California Medical Center

RIASSUNTO

Questo lavoro riporta esperienze con un sistema per l'aiuto alle decisioni del chirurgo. Pone in evidenza i requisiti di interfaccia uomo - macchina: una buona interfaccia medico - calcolatore dovrebbe contribuire a risolvere almeno alcuni dei problemi, sia tecnici sia psico - logici, che emergono quando sistemi software sono introdotti nell'ambiente medico. Tali problemi sono descritti nel lavoro, mettendo in evidenza il punto di vista del chirurgo.

ABSTRACT

This paper reports experiences with a system for surgical decision. It focuses on requirements on man - machine interface:a good physician - computer inteface shonld helps in solving at least some of the problems, both technical and psycological, arising when computer systrms are introduced into hospitals. These problems are described from the physician's vien point.

1. INTRODUCTION

Up to the present, computer systems for hospitals and clinics have largely concentrated on carrying out administrative functions and none, in this author's view, are designed to deal with the major issue that the physician faces, a solution for the patient's problems.[1,2] One approach to assisting the physician has been to develop "artificial intelligence." To date, this approach also has been unrewarding for two reasons: (1) we have not first exploited the major function of the computer at this time; (2) because artificial intelligence is a misnomer. We should understand that the physician uses his brain for parallel processing of very complex things - he thinks.[3,4] Computers in their present format are excellent in recalling information, reformatting it so that it is useful to the physician in problem solving. We should concentrate on developing the synergism between these two systems - the computer's recall function and the brain reasoning power - rather than thinking that one can substitute for the other.

The need for this synergistic development is heightened at this time by both sociologic and financial influences.[5] There is a great pressure to limit the rise in costs of health care. The methods used in the United States are to do preadmission workups, to try to shorten hospital stay, to limit the number of laboratory tests, limit use of imaging techniques, and, very important, to prevent duplication. These forces which limit health care costs are in conflict with social pressures: the inexorable increases in paper for documentation to satisfy the many growing bureaucracies for controlling payment and to provide a record that permits assessment for quality control. Also opposing reduction of health care costs is the necessary limitation in the number of hours that physicians in training should work. It is no longer acceptable to keep these people on call for 36 hours and only offer 12 hours free time every

48 hours. The complexity of decisions and the social pressures for a more reasonable life style will inevitably force change from this exploitation. In the United States, we are facing a shortage of a quarter of a million nurses and are closing ICUs because we do not have adequate nurses to care for the patients. These social needs must be met by diminishing the burden of paper work and documentation and making information more readily available to both physicians and nurses.

2. PRESENT MEDICAL COMPUTER SYSTEMS

Computers designed to run hospitals, as process controllers for laboratories, to manage pharmacy inventories, cannot meet the needs of physicians and nurses. We must analyze the physicians' as well as their nurse partners' methods of working. The physicians must define the patient's problem, and to do that they must have an easy and effective way of recording history and physical examination. When they have done that, they must decide on what other studies are needed to define the patient's problem. The choices and cost of studies and treatments increase with each advance in medicine. This is an element of the present health care crisis.

Table I lists the multiple nodes of decisions that are required to solve a patient's problems. In each subgroup the choices are increasing, becoming more difficult, and mistakes and duplication more expensive.

Methods of Imaging	**Clinical Laboratori Test**
Classic x-rays	Blood chemistries
CAT scans	Drug levels
Ultrasound	Toxicology
Magnetic resonance	Immunology
Imaging	Microbiology
Radionucleotide	Hematology
Angiography	Coagulation
Diagnostic Procedures	
Cardiac evaluation	
Pulmonary function	
Exercise tolerance	
Endoscopy	
Needle biopsies	
Non-invasive vascular	

Table 1 history and phisical exmacillary information (new and old)

Physicians no longer have just classic x-rays; they must decide between computed axial tomography (CAT) scans, ultrasound, magnetic resonance imaging, radionucleotide scanning, or angiography. Does the patient need cardiac evaluation, pulmonary function evaluation, should exercise tolerance be included, is endosocopy indicated, should the invasive radiologist be called for needle biopsies, and should they have non-invasive vascular evaluation? The increase in these choices inevitably means that the physicians need more information and more help in arranging for the appropriate workup for their patients.

When they have completed the analysis of the problem, then they must enter into choice of treatment - for the surgeon, the primary choice of correct treatment is between medical and invasive treatment. If the decision is made that resolution of the patient's problem requires invasive treatment, it is no longer a question of whether he should have operation but whether he should have an interventional radiologist drain the abscess, a cardiologist dilate the coronary, an endoscopist dilate the restricted common duct, a lithotriptor break the stones, or an open surgical procedure. If the choice is made for medical therapy, the increasing potency and toxicity of available drugs make decisions no less complex. The drug choice involves at least knowledge of the information listed in Table II. The decisions about drug treatments can be more precise if the information listed here is easily recalled by the physician and conflicts pointed out.

Drug efficacy
Drug allergies
Drug-drug interactions
Drug cost
Drug dosing
Drug metabolism
Drug laboratory test interactions

Table 2 drug choice

As this information from all the sources is accumulated, it is filed in a chart in sequential order as reports are received or in different sections of the chart for each source of the information. The physicians are expected to extract pertinent information from this chart, the organization of which totally disregards the purpose of clinical data collection, which is to solve the patient's problem. Patient records are not formatted for problem solving and are inefficient for archiving because they duplicate data without clarifying their meaning.

Much of the laboratory processing is automated and the laboratory directors believe the physician should be satisfied with their reports. Data filed by the agent collecting and in the format useful to the collector ignores the needs of the physicians. The automation of the laboratories with their process controllers has no direct relation to the needs of the physician. For example, drugs may be done in the toxicology physician while BUN, creatinine and potassium are done in the chemistry laboratory; yet the level of the creatine and BUN are critical determinants of drug level. The information from these two sources must be combined to avoid drug toxicity. Why should physicians have to look in different areas of the chart to find out these two types of information? They obviously should be brought together.

At present, I believe only the intensive care unit record is properly formatted, designed for problem solving by charting of all the information needed for decision making in a time-based format so that the physicians can correlate the information.[6] The ICU chart in almost all instances requires the nurses to copy values from other reports and, therefore, becomes a duplicate record and a time consuming chore for the nurses. There are in the offing some attempts at computer generation of this record in a screen display format.[7] We have yet to see whether these will successfully replace paper. At least one of these now being used in the United States is illegible unless the physician is within 18 inches of the screen. The problem of how to get the screen where the nurses and physicians can see it and how to display information has not been solved.

Let us divert from these specific shortcomings to look a little bit at the problems we face. There is a popular term for collecting data: "data acquisition". I would suggest that there are two steps in data acquisition - one is input to the computer, and the other is output to the provider. Entering information into a computer serves no purpose if it is not enhanced when it is presented to the user. The purpose is to improve acquisition by the end user - the physician or nurse. There are a number of input systems for laboratory work, and some procedures, but for the physician's and nurse's own functions, we have not worked out either a good input or output system.

3. A PHYSICIAN-ORIENTED SYSTEM

Our proposal is a new orientation for medical computer systems, a physician-oriented system. We are in the process of developing a physician-oriented patient data acquisition and reporting system. It is based on the Macintosh computer and uses as input interface software, Hypercard, and the data base Fourth Dimension for storage of information and for report generation. We anticipate this system will be networked to hospital information systems. The data of a subgroup of patients will be downloaded to the Macintosh network to speed data recall and to permit reformatting of reports to serve the physician's needs. This distributed processing will accomodate fast response with different report formats for each type of physicians - one way for renal transplant group, another for the cardiac group, each suited to the needs of a group of physicians.

Data acquisition by this system requires only familiarity with the basic Macintosh conventions. It uses a tree structure for entry of data using the mouse, thus minimizing the use of the keyboard. There will be liberal use of graphics. The collection of data is structured and data driven. This is an important concept to emphasize. One of the problems of the tree form of information entry or list entry is that the physicians are expected to enter in great detail all of the information, whether or not it is pertinent.

Our system is designed so that as the physician feels greater detail is appropriate, added entries expand the opportunities for the detail. The physician can elect to stop the entry at any node. As the physicians make choices from the tree, the text for the archived record is displayed for them as they are making the choices. They can change choices or edit the text. Our original concept of a physician-oriented system arose from the need to enter a history and physical examination in a structured manner that would allow the physician to be talking to the patient while he was collecting some of the data. If the history is entered into the computer immediately in a structured form, it will have to be collected only once and can be easily accessed by all those who need it, unlike the present chart that can be removed by another group, or lost.

We can put in prompts for important information. For example, in surgical patients we can make sure that the inquiry is made as to whether the patient is on aspirin, has he had history of previous bleeding, or any other pertinent, critical information. Equally important to the physicians as they are taking the history is to identify the patients' problems. Our system will allow them to list the problems as they are collecting and analyzing data. This problem list can then be updated on each subsequent visit. The interactive capabilities of the Macintosh computer are enhanced by the use of Hypercard and its buttons and text windows, and the entry of clinical data is facilitated. We have capitalized on Hypercard by writing a number of subroutines to fit the real needs of physicians and aiding them in entering data. The software is designed to permit immediate exit from the system, if the physician is called

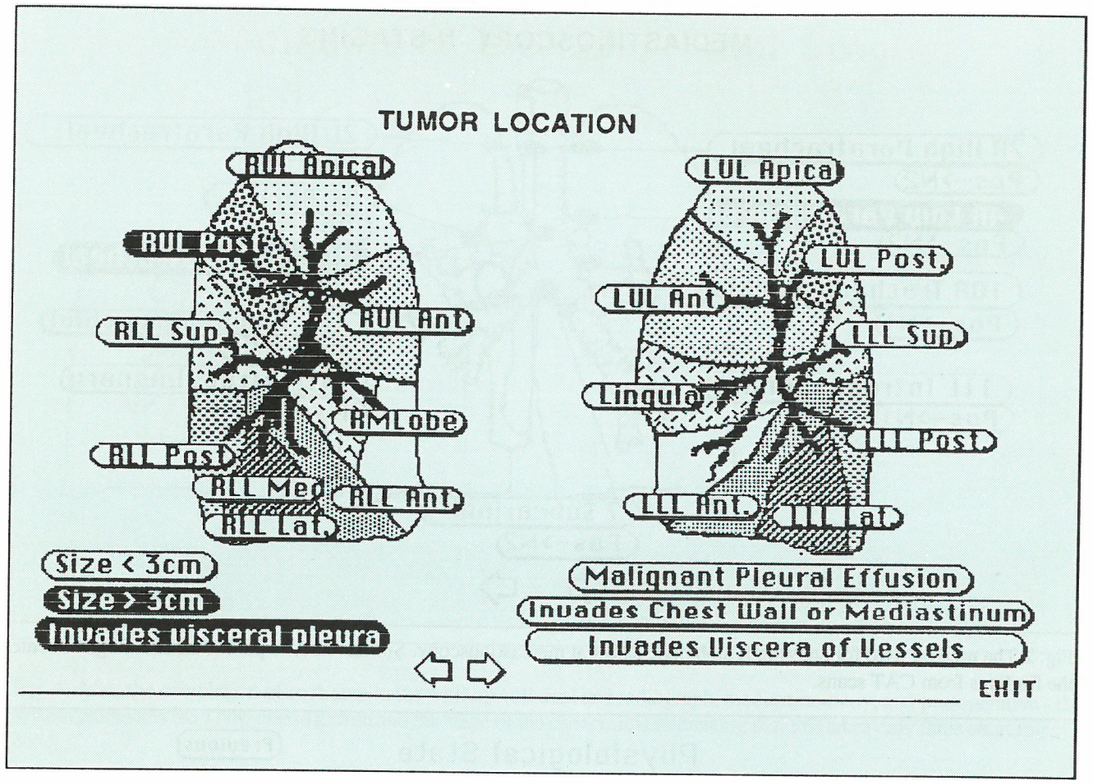

Fig. 1 The screen display when the user enters the carcinoma of the lung tree. By using the mouse, they click on the appropriate "buttons" and enter information about the tumor size and location. A subsequent screen not illustrated permits entry of the extent of endobronchial involvement.

away, without loss of information. He can come back to right where he was. It is easy to change screens or text displayed. In the jargon of the Hypercard language, the Home Card will provide the physicians access to all the services that they need for solving their patients' problems. The system is viewed as a module for interface by the physician which can be networked to hospital computer systems, retrieving laboratory work or other information, scheduling procedures, accessing drug order entry system from the pharmacy system, or using a hospital entry's order system with the Macintosh comput eracting as the interface to that system.

4. LUNG CANCER STAGING

For this surgical audience, I have chosen to illustrate the design concept with a series of entry panels for a patient with carcinoma of the lung. We use it to stage the carcinoma and the physiologic state of the patient with each step of the workup. The orderly collection defines the decision nodes unobtrusively and assists the physicians in making appropriate decisions. In the first screen to come up (Fig. 1), the tumor location and size are defined.

T staging is completed with a screen for detailing bron choscopic findings. The next screen looks at the preoperative physical examination and imaging analysis (CAT scan) of the lymph node status to give the indication for a mediastinoscopy. If mediastinoscopy is indicated, the nodes found from the CAT scan are displayed and the user defines (1) whether they were biopsied, and (2) if they were histologically positive (Fig. 2).

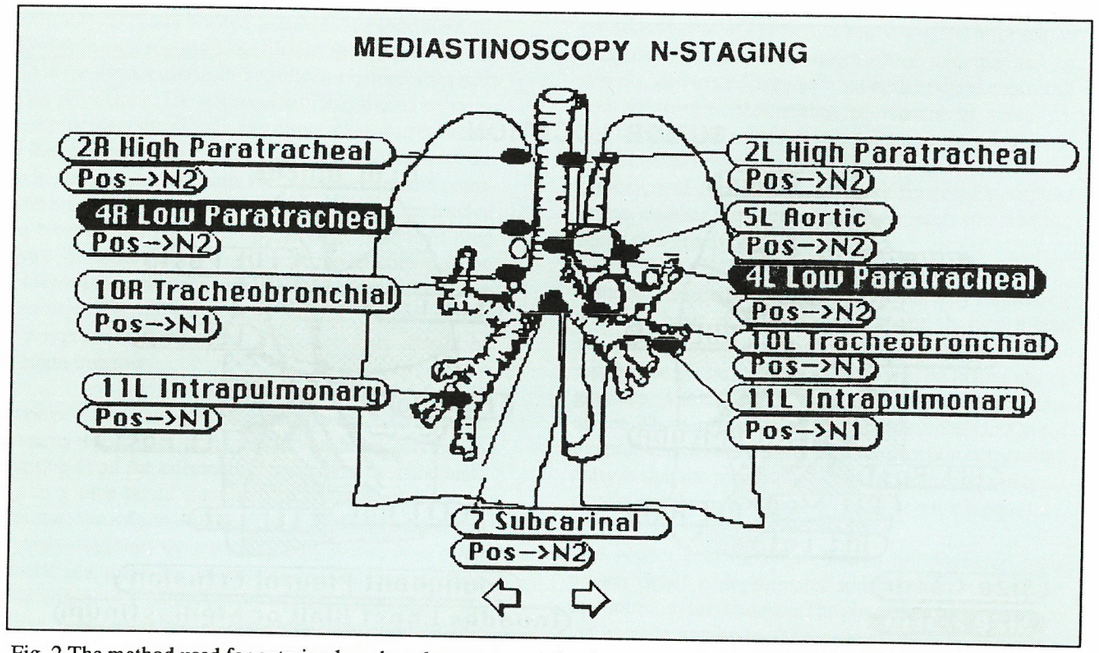

Fig. 2 The method used for entering lymph node status at mediastinoscopy. Similar screens permit the radiologist to enter the findings from CAT scans.

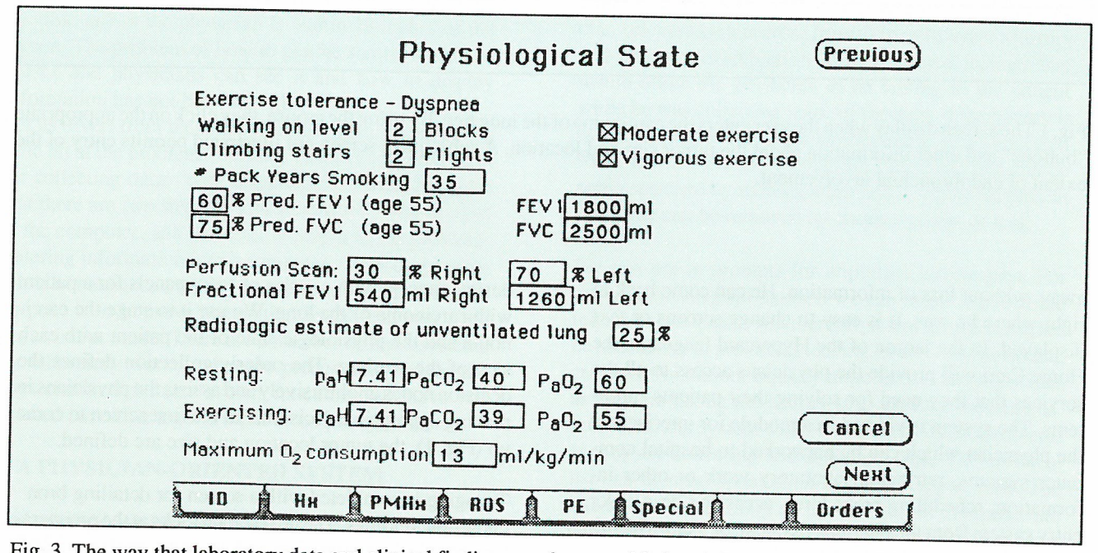

Fig. 3 The way that laboratory data and clinical findings can be assembled to define the physiologic state of the patient. A screen for cardiac status is included in the lung cancer system.

The preoperative TNM classification and stage are then calculated for the physician. [8]In parallel, the physiologic state of the patient is evaluated. These screens (Fig.3) assemble the available data from pulmonary function and cardiac studies and pertinent laboratories or provide an easy entry system if a laboratory is not interfaced to the system.

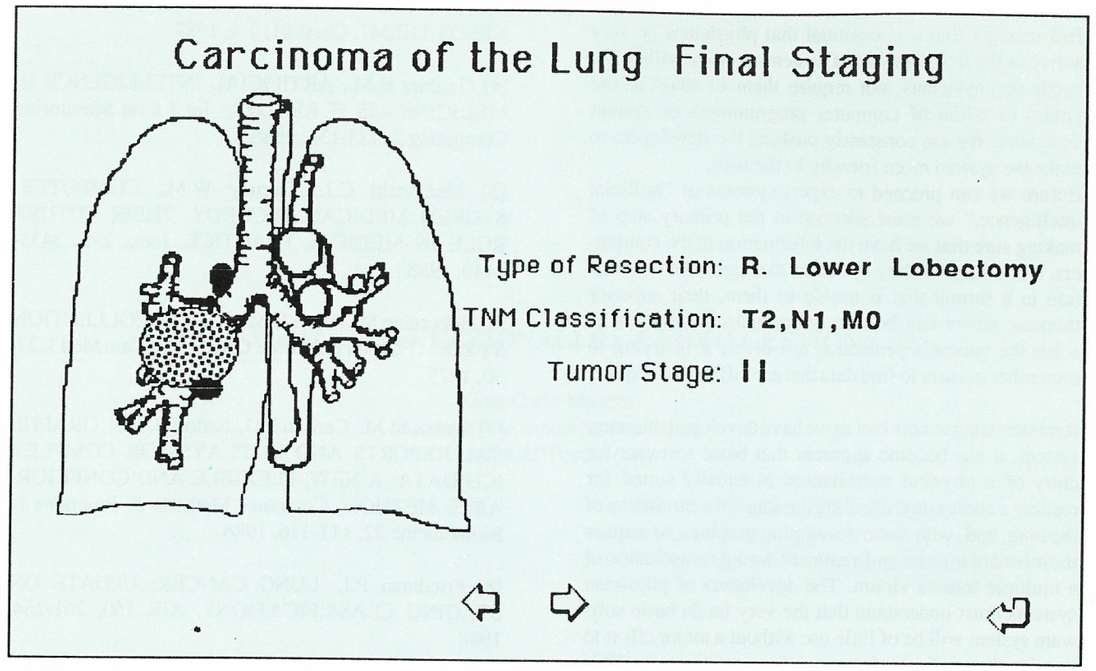

Fig. 4. After the completion of each step in the workup - clinical and radiograph, mediastinoscopy, and postoperative - the system calculates the TNM classification and the stage of the cancer. It is interesting that this takes only three short lines of code.

In at most four screens, all of the information is assembled for decision about the indications for resection of the patient's cancer. The anesthesiologist, consulting pulmonologist, and surgeon need not sort and collate information from many sources. If resection is appropriate, the type of resection and the nodal status after resection are entered on the final screen and stage recalculated. This screen then assures accurate recording of stage for all patients. Statistical analysis is easy and prognostic advice to patients accurate.

5. DISCUSSION AND CONCLUSIONS

This type of organization of data means that those people taking care of the patient at all levels can recall the data. We have good anesthesiologist to surgeon information exchange and vice versa, and the same information is completely available to consultants or the nursing staff. Retrospective analyses of the results of resections are meaningful because we can be assured that the data collection is s complete as possible. The TNM stage and analysis of data will be easily done (Fig. 4).

I am sure it is apparent to the reader that the same kind of screens can be set up for colon cancer, breast cancer, or any other cancer to provide preoperative staging and orderly collection of the information. Criteria for other procedures can be developed.

The example shown of method of staging and working up a carcinoma of the lung gives us the diagnostic side of the issue. The system must also provide the other information that is so essential to physicians. From the input side of the system, one can exit to query Medline, call on the drug information immediately available with suggested doses, etc., from information stored on CD disk and easily return to the patient chart. Also we will display interpretations of other information and protocols for patient workup or treatment that may be active in the institution. All of these are easily accessed from the menu bars.

We believe modularity is essential. With the type of architecture being developed, this system can be used in a hospital, in a physician's office or clinic, with a configuration appropriate to each one of these environments. We

feel strongly that it is essential that physicians be very active in the development of systems so they will fit the needs of physicians, not require them to adapt to the jargon or whim of computer programmers or system designers. We are constantly pushing the developers to make the system more friendly to the user.
Before we can proceed to expert systems or "artificial intelligence," we must succeed in the primary step of making sure that we have the information in the computers. If it is easy for the physicians to acquire the information in a format that is usable to them, their superior thinking power can be free to correlate information to solve the patient's problems, not divert it in trying to remember trivia or to find data that are difficult to acquire.

It is interesting to note that as we have developed this entry system, it has become apparent that basic software for entry of a physical examination is equally suited for reading a chest x-ray, deciding the stage of a carcinoma of the lung, and, with some developing graphics, to acquire the record of injuries and treatment during resuscitation of a multiple trauma victim. The developers of physician systems must understand that the very facile basic software system will be of little use without a major effort to provide the medical information for tree structures. This need for detailed medical information requires a strong alliance between a medical center and a computer development group to provide the best software tools for facile clinical input and output of the medical information.[9]

To date, we have found a great resource for this in bright medical students with computer knowledge who are developing their skills in medical practice.

The important message I want to leave is that a cooperative effort between software system developers and clinicians (removing the institutional bureaucracies of the software industry and health care industry) will result in a synergism leading to orders of magnitude increase in computer effectiveness in medicine. Those of us interested in clinical computer development must listen to the active practicing physicians. The system must serve them.

6. REFERENCES

[1] Friedman B.A., Martin J.BHOSPITAL INFORMATION SYSTEMS. THE PHYSICIAN'S ROLE, Jama 257, 1792, 1987

[2] Bleich H.L, Beckley RF, Horowitz G.L. et al, CLINICAL COMPUTING IN A TEACHING HOSPITAL, N Eng J Med 312, 756-764, 1985

[3] Kinney E.L., MEDICAL EXPERT SYSTEMS. WHO NEEDS THEM?, Chest 91, 3-4, 1987

[4] Gardner R.M., ARTIFICIAL INTELLIGENCE IN MEDICINE - IS IT READY?, Int J Clin Monitoring Computing 2, 133-134, 1986

[5] McDonald C.J., Tierney W.M., COMPUTER-STORED MEDICAL RECORDS. THEIR FUTURE ROLE IN MEDICAL PRACTICE, Jama 259, 3433-3440, 1988

[6] Hilberman M., Peters R.M., A DATA COLLECTION SYSTEM FOR INTENSIVE CARE, Crit Care Med 3, 27-30, 1975

[7] Shabot M.M., Carlton P.D., Sadoff S. et al, GRAPHICAL REPORTS AND DISPLAYS FOR COMPLEX ICU DATA: A NEW, FLEXIBLE AND CONFIGURABLE METHOD, Computer Methods & Programs in Biomedicine 22, 111-116, 1986

[8] Friedman P.J., LUNG CANCER: UPDATE ON STAGING CLASSIFICATIONS, AIR 150, 261-264, 1988

[9] Shortliffe E.H., COMPUTER PROGRAMS TO SUPPORT CLINICAL DECISION MAKING, Jama 258, 61-66, 1987

LE CONDIZIONI ECCEZIONALI E LA LORO GESTIONE (*)

Gian Carlo Macchi
Siemens Telecomunicazioni S.p.A.
Cassina de' Pecchi (MI)

RIASSUNTO

Un sistema di elaborazione non può essere più affidabile e sicuro del software che su di esso risiede. La comunità scientifica non ha tuttavia ancora ben definito la giusta miscela di fault-avoidance e di fault-tolerance che il software dovrebbe possedere per far fronte a malfunzionamenti dovuti ad errori di progetto o di debugging.
Questo scritto tratta delle eccezioni, un argomento che partecipa di entrambi gli aspetti di cui sopra. Delle eccezioni vengono considerati la definizione, i modelli realizzativi e gli schemi di intercettazione e di gestione: il tutto con particolare riguardo alla terminologia e ai metodi più largamente adottati o più promettenti per il futuro.

ABSTRACT

A computing system cannot be more reliable and safer than the software residing on it. The scientific community have not yet well defined the right mixture of fault-avoidance and fault-tolerance the software should have to front bad working because of design or debugging errors.
This paper deals with the exceptions, a topic comprehensive of above both arguments. Of the exceptions, the definition, models of implementation, and mechanisms for catching and managing are considered: the whole with detailed interest into the terminology and the methods more widely adopted or more promising for the future.

1. DEFINIZIONE DI ECCEZIONE

La necessità di definire che cos'è un'eccezione prima di inserire in un linguaggio meccanismi che la riguardano può sembrare ovvia. Nella realtà, come spesso accade, le cose sono andate diversamente. Infatti i primi tentativi di dotare i linguaggi di strumenti per l'intercettazione e il trattamento delle eccezioni sono stati effettuati senza che i progettisti disponessero di una definizione esplicita a cui fare riferimento.

Questo modo di procedere, che può aver avuto inizialmente qualche giustificazione, non può essere ulteriormente mantenuto a causa dei risultati fortemente discutibili che ha prodotto.

1.1 Che cos'é un'eccezione

Un'eccezione (o condizione eccezionale) è la manifestazione che qualcosa di imprevisto è avvenuto durante l'esecuzione di un programma.
Sua caratteristica peculiare è l'imprevedibilità, nel senso che nessuna persona ragionevole di solito scrive programmi contenenti eccezioni, cioè che, ad esempio, danneggiano i puntatori di una lista o tentano di calcolare il logaritmo di zero.

Il termine "eccezione" pone l'accento sul fatto che la condizione che si verifica è qualcosa di non usuale che impedisce la normale prosecuzione del programma; di proposito non viene utilizzato il termine "errore" poiché il verificarsi di un'eccezione non significa necessariamente

(*)Apparso anche su: Newsletter AICA "Affidabilità nei Sistemi di Elaborazione", n° 2, Scuola Superiore Reiss Romoli - STET, maggio 1987.

che qualcosa sia sbagliato nel programma stesso.

L'eccezione è quindi una condizione che richiede un'azione, senza la quale il programma potrebbe terminare prematuramente con un errore, oppure ciclare indefinitamente, oppure anche raggiungere un punto d'uscita normale, ma non l'obiettivo ad esso associato.

In base alla definizione data è chiaro che, in generale, di un'eccezione non sono noti né la causa, né dove tale causa risiede: il tentativo di calcolare il logaritmo di zero può essere dovuto ad un evento ignoto e irrilevato verificatosi in una parte del programma anche molto "lontana" da quella deputata ad eseguire il calcolo.

Si noti infine che la definizione di eccezione, nel significato qui adottato, è stata anche formalizzata. Sono inoltre in corso tentativi di definizione formale dei meccanismi d'intercettazione e di trattamento. Per una loro trattazione si può fare riferimento a [14], [8], [9], [10].

1.2 Che cosa un'eccezione non è

In base alla definizione data si può escludere che un'eccezione possa essere considerata una struttura di controllo dei linguaggi di programmazione.
Pertanto, ad esempio, la possibilità di trattare la fine di un file non dovrebbe essere realizzata, come in PL/I o in alcune versioni preliminari di Ada, mediante la generazione di un'eccezione qualora si tenti di leggere oltre la fine del file stesso, ma piuttosto dotando il linguaggio di una funzione intrinseca (come la function EOF del Pascal) che consenta di verificare, prima di effettuare la lettura, la liceità dell'operazione.

Un' eccezione infatti non è un evento pianificato, ma una condizione che impedisce il completamento di un'azione pianificata. Un' eccezione non deve avere scopi di monitoring (1) e quindi non può essere la conclusione sistematica di un'operazione.
Ad esempio un programma che debba contare quante volte una parola è presente in un testo non dovrebbe generare un'eccezione ogni volta che trova la parola cercata.
Il problema può invece essere risolto con strumenti più adatti, come i monitor (2) o le coroutine (3).

Un'eccezione non va considerata come un errore, come è già stato detto (v. § 1.1). Infine un'eccezione non è necessariamente legata alle capacità di interrupt della macchina cui il programma è destinato, anche se tali capacità, peraltro spesso limitate alla rilevazione e alla segnalazione di una condizione, ma non alla sua gestione, possono essere convenientemente sfruttate.
La possibilità di intercettare e trattare le eccezioni deve essere infatti un meccanismo linguistico intrinseco del linguaggio di programmazione ad alto livello e della sua capacità di descrivere i problemi in modo il più possibile astratto e indipendente dalla macchina ospite.

Per finire va notato come la non comprensione di che cosa sia un'eccezione e la sua confusione con altri concetti ha portato, nei casi più fortunati, a meccanismi inutilmente pesanti e posticci, in altri casi addirittura a meccanismi inutilizzabili [28].

1.3 Una breve digressione sul PL/I

Nel PL/I, il primo linguaggio ad essere stato dotato di un meccanismo per la rilevazione e il controllo delle condizioni eccezionali, chiamate "on condition", il concetto di eccezione è assimilato a quello di interrupt.
Ciò può essere derivato sia dalla mancanza, all'epoca del progetto, di una definizione di eccezione, sia dallo scopo principale che i progettisti si erano prefissati: creare un linguaggio che consentisse di programmare tutte le applicazioni, quindi utilizzabile non solo dai programmatori scientifici e commerciali, ma anche dai programmatori di sistema (4).

I linguaggi ad alto livello fino a quel momento disponibili avevano poche possibilità di definire le azioni da intraprendere al verificarsi di condizioni come overflow, divisioni per zero e così via. Spesso tali condizioni potevano essere intercettate solo ricorrendo a patch in Assembly. Più raramente il linguaggio consentiva l'intercettazione, ma non disponeva di costrutti adatti alla gestione della situazione. Il problema fu così risolto dotando il PL/I dell'istruzione eseguibile: ON <condizione> <azione>.

2. MODELLO REALIZZATIVO

La definizione di eccezione dovrebbe essere accompagnata da un modello di realizzazione. E' opportuno che tale modello sia definito in modo esplicito: infatti l'esperienza nel campo dei linguaggi di programmazione ha mostrato che i progettisti tendono comunque a costruirsi un proprio modello mentale che può facilmente risultare incoerente e inadeguato; in tal caso il minimo che può accadere è un'esagerazione delle difficoltà realizzative (5).

Naturalmente occorre tenere presente che potrebbe essere
la definizione del linguaggio o addirittura la definizione stessa di eccezione a precludere la possibilità di trovare un modello realizzativo concettualmente semplice (6).

Un esempio di linguaggio recente che crea difficoltà nella definizione di un modello semplice di eccezione è Ada. Ciò è dovuto alle caratteristiche stesse del linguaggio che, anziché essere costituito da pochi costrutti componibili, ne comprende invece parecchi specializzati e, spesso, non necessari [12] [24].

2.1 Modello di propagazione delle eccezioni

Per poter intercettare e trattare le eccezioni è necessario un meccanismo che consenta di trasmettere informazioni fra l'unità del programma in cui la condizione eccezionale viene intercettata e quella destinata a trattarla. Inoltre, per le ragioni già esposte, è opportuno che tale meccanismo si conformi ad un modello realizzativo, che verrà indicato come "modello di propagazione delle eccezioni".

Si noti che, in questo scritto, con la dizione "unità di programma" si indica una parte di programma distinguibile sintatticamente dalle altre e che realizza un'astrazione di tipo procedurale (7).

Si supponga allora che l'unità A chiami l'unità B. Come conseguenza viene generata un'attivazione di B (8). Se l'attivazione di B non termina in modo normale si dice che la chiamata a B ha "sollevato" un'eccezione e poiché, in generale, B non è in grado di trattare tale condizione eccezionale, si limiterà a "segnalare" il fatto a un'unità di programma C incaricata del trattamento e chiamata, d'ora innanzi, "handler" dell'eccezione.

A seconda delle limitazioni imposte a C, si hanno due modelli:

1. PROPAGAZIONE AD UN LIVELLO ("one level propagation" o "single level propagation"): se si suppone che C coincida con A o, più precisamente, che l'attivazione di C coincida con l'attivazione di A che hachiamato B (fig. 1);

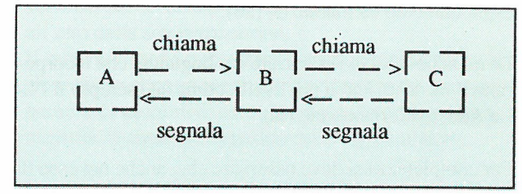

Fig. 1: Modello di propagazione ad un livello

2. PROPAGAZIONE A PIU' LIVELLI ("multilevel propagation" o "free propagation"): in caso contrario. Anche in quest'ultimo caso, tuttavia, si impone usual - mente una limitazione sulla scelta di C: si suppone cioè che di C esista almeno un'attivazione e che da questa sia derivata, attraverso una catena di chiamate, l'attivazione di B che ha segnalato l'eccezione (fig. 2).

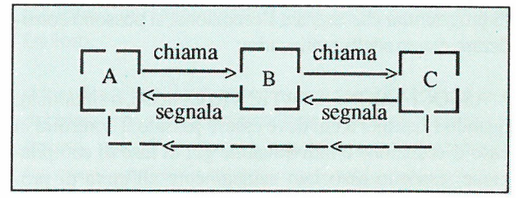

Fig. 2: Modello di propagazione a più livelli

(1) Anche se questo punto di vista è stato sostenuto da Goodenough in un lavoro fondamentale sulle eccezioni [13].
(2) Ad esempio i monitor del Concurrent Pascal [15].
(3) Ad esempio le coroutine del Simula realizzate da oggetti di tipo classe [25].

(4) A riprova di ciò basti pensare ai concetti che esso incorpora: multitasking, puntatore, area, offset, variabile basata; senza contare che le funzioni intrinseche UNSPEC e ADDR, accoppiate a puntatori liberi contenenti indirizzi di memoria, consentono addirittura di indirizzare la memoria centrale.

(5) Come esempio si possono ricordare le difficoltà incontrate nella realizzazione dei primi compilatori dell'Algol 60, nonostante il linguaggio fosse stato definito con una precisione sconosciuta prima d'allora (1960-1963). Solo quando fu noto ai progettisti il modello realizzativo (dovuto a Dijkstra) basato sul run-time stack, la realizzazione si semplificò enormemente.
(6) Tornando alla realizzazione dei compilatori non è un mistero che, mentre per alcuni linguaggi si sono potuti trovare semplici modelli realizzativi (a stack per l'Algol, a stack e heap per il Pascal, a stack nidificati interagenti per il Simula), per altri non sono noti modelli "puliti": ad esempio per il PL/I e per Ada. Non è da escludere che, in entrambi i casi, l'enfasi posta sull'universalità del linguaggio abbia messo in ombra le modalità di realizzazione.

(7) Ulteriori precisazioni esulano dallo scopo di questo scritto e possono essere trovate, ad esempio, in [17]. Unità di programma nel senso suddetto sono, ad esempio, le procedure del Pascal, i blocchi dell'Algol, le subroutine del Fortran, ecc. Quale definizione adottare per un certo linguaggio è una decisione che terrà conto (sperabilmente) sia delle caratteristiche del linguaggio stesso, sia del modello di propagazione scelto.
(8) La dizione "viene generata un'attivazione di B" è usata di proposito al posto di "viene attivata B" per insistere sul fatto che diverse attività della stessa unità potrebbero coesistere sia in vera concorrenza su una macchina multiprocessor, sia in una quasi - concorrenza gestita dalle capacità multitasking del sistema operativo oppure dal programma stesso.

L'inserimento di un meccanismo di propagazione in un liguaggio moderno non dovrebbe evidentemente essere in contrasto con la possibilità di realizzare programmi ben strutturati e modulari. Ad esempio un'unità A che chiami un'unità B dovrebbe conoscere che cosa fa B e quali eccezioni B può segnalare, perché ciò fa parte dell'astrazione che B realizza. Ma, in omaggio al principio di information-hiding, A non dovrebbe conoscere "come" B opera né quali eccezioni possono essere segnalate da altre unità invocate da B.
E' evidente che tale obiettivo può essere conseguito solo se si adotta il modello di propagazione ad un livello e se si prevede che tutte le uscite eccezionali di un'unità siano esplicitamente dichiarate (9) [26].

Le manchevolezze riscontrate nei linguaggi che incorporano il meccanismo a più livelli, come ad esempio il PL/I e Ada, confermano ciò (10).

Per completezza si deve osservare che, anche nel caso di propagazione ad un solo livello, si possono presentare difficoltà nella definizione di un modello coerente, qualora il linguaggio ammetta la possibilità di procedure ricorsive o di classi (11). Una possibile soluzione potrebbe essere, nel caso di una procedura o di una classe invocata in modo ricorsivo, di far terminare solo l'ultima invocazione o istanza [23] (v. § 4).

3. ASSOCIAZIONE DEGLI HANDLER ALLE UNITA' DI PROGRAMMA

Per quanto riguarda l'associazione di un handler all'unità di programma che segnala l'eccezione, si possono considerare due modelli realizzativi:

1. ASSOCIAZIONE STATICA (o lessicale, o sintattica): quando l'handler a cui deve essere passato il controllo in caso d'eccezione è individuabile già in fase di compilazione, essendo associato staticamente all'unità di programma che può segnalare l'eccezione;

2. ASSOCIAZIONE DINAMICA: quando la selezione dell'handler dipende dal cammino seguito fino al momento in cui si è verificata l'eccezione; in altre parole la stessa eccezione, verificatasi nella stessa unità di programma, può essere gestita da handler diversi.

(9) A meno che non si adotti una pura gestione di default (v. § 5.3).
(10) A proposito di Ada vale forse la pena di osservare che vengono continuamente proposte modificazioni alla definizione del linguaggio [32] [35] e in particolare ai suoi meccanismi sulle eccezioni [12]. Il suo manuale di riferimento, peraltro, si mantiene piuttosto vago sull'argomento.
(11) Nel senso delle classi del Simula.

Il modello statico è l'unico che consente una programmazione strutturata e modulare: inoltre esso permette la selezione dell'handler con metodi validi ed efficienti, appunto perché già adottabili in fase di compilazione [26]. Fra questi si ricordano quelli basati sulle "branch table" e sulle "handler table" (v. § 5.1).

Si può osservare che una rigorosa applicazione del modello statico richiede necessariamente che tutte le uscite eccezionali siano dichiarate esplicitamente, qualora si adotti il modello di propagazione a più livelli (12). Ciò non toglie che ciò sia desiderabile anche nel caso di propagazione ad un solo livello, a meno che non si adotti una pura gestione di default (v. § 5.3).

Infine si può ricordare che il modello dinamico [14], che conta ancora alcuni adepti, è quello adottato, con risultati non molto felici, nel PL/I.

4. ESISTENZA DELL'UNITA' SEGNALANTE DOPO LA SEGNALAZIONE

Per quanto riguarda la sorte dell'unità segnalante, sono possibili due modelli realizzativi:

1. CON TERMINAZIONE ("termination model" o "escape model"): se l'unità segnalante cessa di esistere dopo la segnalazione;

2. CON PROSEGUIMENTO ("resumption model" o "notify model"): se, dopo la gestione dell'eccezione, l'esecuzione dell'unità segnalante viene fatta proseguire.

Il modello con proseguimento, benché attentamente considerato e suggerito da Goodenough [13], pone importanti problemi relativi all'interdipendenza fra le diverse unità di programma, poiché altera le relazioni gerarchiche fra le unità stesse, rendendole mutuamente dipendenti.
L'inserimento di tale meccanismo in un linguaggio non sembra attualmente opportuno, non essendo ancora stata trovata una soluzione soddisfacente di tali problemi.
Per inciso si nota che nel PL/I è incorporata una sorta di modello con proseguimento, benché in forma piuttosto confusa e incoerente (13).

Attualmente il modello favorito è pertanto quello a terminazione (adottato, ad esempio, in Ada) che, nonostante la maggiore semplicità, permette di affrontare quasi tutte le situazioni. Senza contare che, anche qualora si intendesse adottare il modello a proseguimento, occorrerebbe quasi certamente prevedere anche l'altro modello.

(12) Come controesempio si consideri che in Ada, mancando tali dichiarazioni esplicite, la selezione degli handler è possibile solo in esecuzione, pur essendo essi associati staticamente alle unità di programma (associazione "quasi-statica"). Per una discussione dettagliata sull'argomento si veda [16].
(13) La definizione del linguaggio PL/I è addirittura precedente al succitato articolo di Goodenough.

5. GESTIONE DELLE ECCEZIONI

Un evento eccezionale può evidentemente essere gestito in modo più o meno complesso. A questo scopo sono state proposte e realizzate strategie basate sia su gestioni di default, sia su gestioni programmate. Qualunque strategia deve comunque affrontare problemi che rientrano nella seguente lista:

- SEGNALAZIONE dell'eccezione;
- RECOVERY di uno stato consistente del programma;
- MASCHERAMENTO dell'eccezione.

(13) La definizione del linguaggio PL/I è addirittura precedente al succitato articolo di Goodenough.

5.1 La segnalazione

Per quanto riguarda la segnalazione, supponendo di adottare il modello di associazione statico con propagazione ad un solo livello, due metodi particolarmente efficienti per la selezione dell'handler appropriato sono quelli basati sulle "branch table" e sulle "handler table".

Il primo metodo, che cerca di ottimizzare la velocità di esecuzione, associa ad ogni invocazione di un'unità di programma una tabella di salto contenente un elemento per ogni eccezione che può essere sollevata nell'unità (eventualmente un solo elemento se si adotta un'unica gestione di default).

Il secondo, che cerca di minimizzare l'occupazione di memoria, associa invece ad ogni handler una tabella contenente sia le regole di visibilità dell'handler da parte delle varie unità di programma, sia una lista delle eccezioni che l'handler può gestire. Questo metodo è da preferirsi qualora il numero di handler sia molto minore del numero di invocazioni (ciò che normalmente si verifica).

Entrambi i metodi sono stati sperimentati in diverse versioni del linguaggio CLU [26]; l'ultimo è quello adottato nel linguaggio CHILL [18].

5.2 La recovery di stato e il recovery cache

E' evidente che la sola segnalazione, benché utile nelle fasi di debugging [4], non garantisce un programma robusto, cioè che tolleri eccezioni, poiché il verificarsi di un evento eccezionale, nell'accezione adottata in questo scritto (v. § 1.1), comporta l'impossibilità di stabilire a priori l'entità del danno da essa provocato.
Ciò fornisce ulteriore sostegno alle argomentazioni in favore del non proseguimento dell'unità in cui l'eccezione si è verificata (v. § 4) e suggerisce l'opportunità di ripristinare, in caso d'eccezione, uno stato consistente del programma, prima di intraprendere qualsiasi altra azione.

Per risolvere il problema si può utilizzare il recovery cache, un meccanismo che consente di tenere automaticamente traccia dei valori delle variabili di programma sufficienti a ricostruire, qualora in un'unità si verifichi un'eccezione, lo stato che il programma aveva all'atto della sua invocazione.
Il recovery cache può essere considerato come una versione automatica ed eventualmente ottimizzata, rispetto al numero di variabili di stato del programma, di più antiche tecniche di checkpoint gestite dal programmatore.

5.3 La gestione di default e i recovery block

Sul recovery cache è basata la tecnica dei "backward recovery block", messa a punto dal Reliability Research Group dell'Università di Newcastle upon Tyne [2].

Si tratta di dotare il linguaggio di programmazione di un nuovo costrutto, il recovery block, costituito da un'unità di programma, eventuali unità alternative e un test di accettazione.

Le varie unità del blocco devono realizzare lo stesso algoritmo, pur mantenendosi il più possibile indipendenti fra loro.

Il test di accettazione deve essere unico per tutte le unità del blocco ed essere indipendente dalla loro costituzione interna, poiché costituisce una misura del successo del funzionamento della nuova astrazione (il recovery block appunto) considerata come un tutt'uno.

L'invocazione del recovery block provoca l'esecuzione dell'unità principale e del test di accettazione. Se quest'ultimo fallisce, o se è stata segnalata un'eccezione da un blocco a livello inferiore, viene ricostruito lo stato inizaile del blocco mediante il recovery cache. Se esistono unità alternative viene chiamata la prima della lista; in caso contrario il blocco segnala l'eccezione al chiamante. Ovviamente anche per l'unità alternativa viene eseguito lo "stesso" test di accettazione e così via.
E' evidente che questo meccanismo giunge addirittura a mascherare completamente l'eccezione nel caso in cui una delle unità alternative riesca a superare il test.
Per completezza vanno citate le critiche rivolte a questa

tecnica, ritenuta di difficile inserimento in linguaggi preesistenti [23] e limitata, poiché consente una pura gestione di default, nel senso che tutte le eccezioni che avvengono all'interno di un blocco di recovery sono indistinguibili e trattate allo stesso modo.

La prima obiezione appare non molto fondata: nel 1984 Ron Kerr, uno dei propositori del metodo, ne ha incorporato una versione funzionante nel linguaggio Simula; proposte simili sono inoltre state fatte per Ada [12]. Da affrontare è piuttosto il problema di determinare con certezza se il test di accettazione relativo ad un blocco consente di catturare tutti i casi in cui il comportamento delle unità del blocco stesso si discosta dalle specifiche. Le conoscenze attuali [8] sembrano tuttavia sufficienti per la realizzazione di un sistema in grado di risolvere il problema.

La seconda obiezione è, a giudizio dell'autore, più fondata. Va tuttavia notato che, benché il punto di vista adottato possa essere considerato troppo drastico, soprattutto durante le fasi di sviluppo e di debugging, esso è nettamente orientato all'ottenimento di programmi fault-tolerant ed è perfettamente consistente con l'opinione di chi ritiene che raramente il verificarsi di una vera eccezione fornisce informazioni sulla sua causa e che pertanto possono non aver senso tentativi di gestione basati su tale presunzione.

5.4 La gestione programmata

Una gestione completamente programmata delle eccezioni non è evidentemente proponibile per programmi che debbano possedere requisiti di robustezza. Una soluzione che contempera le diverse esigenze di trattamento potrebbe perciò essere quella che unisce alla gestione programmata una gestione di default di quelle eccezioni per le quali non sono previsti handler espliciti.

I meccanismi di questo tipo attualmente inseriti in alcuni linguaggi di programmazione sono però piuttosto insoddisfacenti. Fra i candidati per futuri sviluppi si possono ricordare gli schemi "sequel based" (14) [LIND84] ed "event based" [30].

Le maggiori promesse sembrano però venire da una ridefinizione del "backward recovery block" che consenta anche la gestione programmata. Tale ridefinizione è tuttora in corso da parte degli stessi propositori del metodo.

(14) La sequela è un'unità di programma che, al termine dell'esecuzione, trasferisce il controllo alla fine dell'unità di programma nella quale è dichiarata come unità locale.

Infine si nota che sono anche possibili meccanismi che adottano una "forward recovery", cioè che ripristinano uno stato successivo a quello che il programma aveva all'atto dell'invocazione dell'unità che ha sollevato l'eccezione. Tali meccanismi sono evidentemente utilizzabili solo per una gestione programmata.

6. DISABILITAZIONE DELLE ECCEZIONI

Non c'è dubbio che, se si accettano le definizioni presentate nei paragrafi 1.1 e 1.2, la definizione del linguaggio di programmazione non dovrebbe consentire al programmatore di disabilitare le eccezioni. I progettisti di CHILL e di CLU, ad esempio, hanno adottato questo punto di vista (impossibilità della disabilitazione).

In realtà nei linguaggi in cui è consentita, la disabilitazione non sembra essere altro che un espediente per supplire a carenze che avrebbero dovuto ricevere ben altra attenzione. Come esempi basti pensare al PL/I o ad Ada, in cui essa è spesso l'unico mezzo che consente al compilatore di ottimizzare il codice prodotto, salvo naturalmente tollerare comportamenti impredicibili qualora l'eccezione disabilitata avvenga ugualmente.

Diverso è il caso di un linguaggio ben progettato per il quale sia possibile, in particolari situazioni, verificare che determinate eccezioni non possono mai sorgere in certe parti di un programma. In questo caso dovrebbe però essere il compilatore ad effettuare la verifica e a non generare il codice necessario a rilevare e gestire l'eccezione.

7. BIBLIOGRAFIA

[1] ADA: LANGUAGE, COMPILERS AND BIBLIOGRAPHY, Cambridge University Press, pagg. 11.1-11.12, Cambridge, 1984

[2] T. Anderson, R. Kerr, RECOVERY BLOCKS IN ACTION, 2nd International Conference on Software Engineering, Proceedings, Atlanta, 1976

[3] W.F. Appelbe, K. Hanse, A SURVEY OF SYSTEMS PROGRAMMING LANGUAGES: CONCEPTS AND FACILITIES, Software Practice and Experience, 15(2), 1985

[4] A. Butti, A. Lora, G.C. Macchi, THE FSM AND MESSAGE COMMUNICATION: IMPACT ON SOFTWARE TESTING, IEEE International Conference on Communications - ICC'84, Proceedings, Amsterdam, 1984

[5] G. Castelli, ALCUNI LINGUAGGI A CONFRONTO, Note di Software, 12, 1979

[6] INTRODUCTION TO CHILL, CCITT Working Party XI/3, COM XI, 389-E, Ginevra, 1980

[7] F. Cristian, A RECOVERY MECHANISM FOR MODULAR SOFTWARE, 4th International Conference on Software Engineering, Proceedings, Monaco di Baviera, 1979

[8] F. Cristian, EXCEPTION HANDLING AND SOFTWARE FAULT TOLERANC, IEEE Transactions on Computers, C-31(6), 1982

[9] F. Cristian, REASONING ABOUT PROGRAMS WITH EXCEPTIONS, FTCS 13th Annual International Symposium Fault Tolerant Computing, Proceedings, Milano, 1983

[10] F. Cristian, A RIGOROUS APPROACH TO FAULT-TOLERAND PROGRAMMING, 4th Jerusalem Conference on Information Technology, Proceedings, Gerusalemme, 1984

[11] N. Cocco, S. Dulli, UNA STRUTTURA PER LA GESTIONE DELLE ECCEZIONI ORIENTATA ALLA PROGRAMMAZIONE MODULARE, Congresso Annuale AICA, Proceedings, Padova, 1982

[12] M. Di Santo, L. Nigro, W. Russo, PROGRAMMING RELIABLE AND ROBUST SOFTWARE IN ADA, FTCS 13th Annual International Symposium Fault Tolerant Computing, Proceedings, Milano, 1983

[13] J. Goodenough, EXCEPTION HANDLING, ISSUES AND A PROPOSED NOTATION, Communications of the ACM, 18(12), 1975

[14] P. Grandi, THE WAY TO HANDLE EXCEPTIONS, Note di Software, 13/14, 1980

[15] P.B. Hansen, THE PROGRAMMING LANGUAGE CONCURRENT PASCAL, IEEE Transactions on Software Engineering, 1(2), 1975

[16] J.C. Heliard, COMPILING ADA, in "Methods and Tools for Compiler Construction", pagg. 375-398, Cambridge University Press, Cambridge, 1984

[17] E. Horowitz, FUNDAMENTALS OF PROGRAMMING LANGUAGE, pagg. 273-293, Springer-Verlag, Berlino, 1983

[18] R. Impagnatiello, COSTRUTTI PER LA GESTIONE DELLE ECCEZIONI IN CHILL E ADA, Congresso Annuale AICA, Proceedings, Padova, 1982

[19] D. Jarret, SOFTWARE FAULT TOLERANCE STAVES OFF THE ERRORS THAT BESIEGE MICROPROCESSOR SYSTEMS, Electronic Design, 32(16), 1984

[20] R. Kerr, THOUGHTS ON EXCEPTION HANDLING IN SIMULA, Simula Newsletter, 11(4), 1983

[21] R. Kerr, ERROR HANDLING BY RECOVERY BLOCKS, Simula Newsletter, 12(1), 1984

[22] J.C. King, SYMBOLIC EXECUTION AND PROGRAM TESTING, Communications of the ACM, 19(7), 1976

[23] J.L. Knudsen, EXCEPTION HANDLING - A STATIC APPROACH, Software Practice and Experience, 14(5), 1984

[24] S. Krogdahl, K.A. Olsen, ADA, AS SEEN FROM SIMULA, Software Practice and Experience, 16(8), 1986

[25] G. Lamprecht, INTRODUCTION TO SIMULA 67, Wieweg & Sohn, Braunschweig, 1981

[26] B.H. Liskov, A. Snyder, EXCEPTION HANDLING IN CLU, IEEE Transactions on Software Engineering, SE-5(6), 1979

[27] D. Loveman, ADA RESOLVES THE UNUSUAL WITH EXCEPTIONAL HANDLING, Electronic Design, January 22, 1981

[28] M. D. MacLaren, EXCEPTION HANDLING IN PL/I, Sigplan Notices, 12(3), 1977

[29] L.R. Nackman, R.H. Taylor, A HIERARCHICAL EXCEPTION HANDLER BINDING MECHANISM, Software Practice and Experience, 14(10), 1984

[30] R. Pooley, EVENT BASED EXCEPTION HANDLING SCHEME, Simula Newsletter, 11(4), 1983

[31] I.C. Pyle, THE ADA PROGRAMMING LANGUAGE, pagg. 64-74, PrenticeHall, 1981

[32] R.L. Schwartz, P.M. Melliar-Smith, THE FINALIZATION OPERATION FOR ABSTRACT TYPES, 5th International Conference on Software Engineering, Pro-

ceedings, San Diego, 1981

[33] C.H. Smedema, P. Medema, M. Boasson, THE PROGRAMMING LANGUAGES PASCAL MODULA CHILL ADA, pagg. 99-101/136-138, Prentice-Hall, 1983

[34] P. Wegner, PROGRAMMING LANGUAGES - THE FIRST 25 YEARS, IEEE Transactions on Computers, C-25(12), 1976

[35] P. Wegner, ON THE UNIFICATION OF DATA AND PROGRAM ABSTRACTION IN ADA, 10th ACM Symposium on Principles of Programming Languages, Proceedings, Austin, 1983

UN CONVEGNO AICA SULLA ELABORAZIONE PARALLELA

Paolo Stofella
Advanced Computing Systems, srl
Milano

Le ricerche volte alla realizzazione di strumenti per il calcolo automatico sono costantemente sollecitate da una recente richiesta, proveniente da svariati settori del mondo scientifico ed industriale, in termini di potenza computazionale. Questi stimoli continui hanno fatto sì che venisse accellerato, in tempi recenti, il processo di esaurimento delle possibilità tecnologiche offerte dallo schema di calcolo di tipo Von Neumann dirottando l'attenzione dei ricercatori verso paradigmi computazionali alternativi ed in particolare verso il calcolo parallelo che sembra rappresentare la via più promettente per la realizzazione di elaboratori di altissima potenza.

In linea con questa tendenza, la ricerca italiana si orienta alla sperimentazione di nuove proposte nell'ambito del calcolo parallelo; nei giorni 17 e 18 dello scorso novembre si é tenuto, presso la sede A.I.C.A. di Milano un convegno internazionale sulla *Elaborazione Parallela*, al quale sono intervenuti numerosi esponenti di università e centri di ricerca italiani e stranieri, fornendo una panoramica completa sullo stato dell'arte e sulle tendenze di sviluppo in questo settore, sia dal punto di vista dell'hardware che da quello del software.

L'intervento introduttivo è stato affidato al Prof. Dadda del Politecnico di Milano, il quale si é sofferamto in particolare sugli aspetti del calcolo automatico che si legano al settore della ricerca scientifica. Attraverso una panoramica sulle esigenze computazionali legate a varie discipline, dall'astrofisica alla chimica, dalle scienze atmosferiche alla fisica delle particelle, sono state delineate le nuove esigenze nella simulazione di sistemi fisici, la quale sempre *più si affianca alla teoria ed all'esperimento come strumento di indagine della natura.*

La relazione del Prof. Vanneschi dell'Università di Pisa, dedicata alle architetture parallele general purpose, ha fornito un quadro estremamente completo sulle macchine parallele oggi esistenti ed un'analisi critica delle loro prestazioni in termini di efficenza e scalabilità.

Data una macchina parallela con n processori, si dice Banda di elaborazione il numero medio di operazioni effettuate al secondo distinguendo tra Banda di picco, nel caso ideale, e Banda effettiva nel caso reale. Si dice efficienza il rapporto tra banda effettiva e banda ideale, saclabilità il rapporto tra banda effettiva con n processori e banda effettiva con un solo processore. Nel seguito sono brevemente descritte le architetture analizzate nel corso dell'intervento:

-*array processor*: caratterizzati da un numero elevato di PE (Processing Element) connessi in una matrice generalmente toroidale. Si tratta in genere di macchine di tipo SIMD (Single Instruction stream Multiple Data stream); l'efficenza si avvicina a 1 quando non vi sia un carico di

comunicazione tra i PE troppo elevato.

- *pipeline*: architetture con processore singolo ed unità esecutiva parallela con modalità SISD (Single Instuction stream Single Data stream).
Esempi tipici di questa architettura sono i supercalcolatori della classe Cray. Particolarmente adatti alla elaborazione vettoriale, questi elaboratori mostrano drastici cali di efficenza quando il codice vettoriale non sia una elevata percentuale del codice totale di un programma.

- *data-flow*: la prima proposta per un'architettura parallela, molto elegante ma poco commerciale, utilizzava uno stile di programmazione funzionale pura (senza side-effect) in cui la computazione veniva rappresentata da un grafo; rami distinti del grafo potevano così essere computati in parallelo. L'overhead di gestione a runtime del parallelismo diveniva però estremamente pesante, rendendo di fatto l'architettura data-flow poco efficente.

- *architetture multiprocessor*: con n processori collegati per mezzo di crossbar o bus di comunicazione, con memoria privata o condivisa e modalità MIMD (Multiple Instruction stream Multiple Data stream). L' indice di scalabilità tende a peggiorare molto quando si aumenti il numero di processori, per problemi di traffico quando sia utilizzato un bus di comunicazione e per problemi di controllo di accessi multipli alla memoria condivisa.

- *architetture* VLIW (Very Long Instruction Word): architetture parallele in cui un programma viene suddiviso in istruzioni indipendenti e parallelizzabili, composte di molte operazioni, a livello statico. Vi è somiglianza con le macchine data-flow, ma in questo caso si evita il carico di gestione della parallelizzazione a runtime.

- *sistemi* MIMD *a parallelismo massiccio*: sistemi basati su componenti VLSI che cercano di sfruttare questa tecnologia evitando i problemi dei sistemi multiprocessor a memoria condivisa. La comunicazione avviene attraverso canali point-to-point e non esiste memoria condivisa, ottenendo una buona scalabilità all'aumentare del numero di processori. Rappresntante più significativo di questa classe di macchine é il transputer, un processore con coprocessore matematico, memoria RAM e interfacce di comunicazione integrati su singolo chip. Elaboratori basati su transputer, come la Compuitng Surface di Meiko o PiNa, una macchina nata dalla collaborazione tra le Università di Pisa e Napoli, presentano caratteristiche che le rendono competitive, a seconda del numero di transputer connessi, con un insieme di elaboratori che va dalle workstation fino ai supercomputer.

Altri interventi hanno delineato le caratteristiche degli elaboratori paralleli attraverso analoghe tassonomie, segnatamente quello del Prof. Cantoni dell'Università di Pavia, con un accento particolare per i problemi legati alla visione artificiale.

Per quanto riguarda le architetture di tipo special purpose, la relazione del Prof. Stefanelli del Politecnico di Milano ha fornito una descrizione di alcune proposte industriali, in particolare di macchine a modello SIMD per la elaborazione d'immagini, e di un prototipo di macchina parallela MSIMD (Multiple SIMD) a struttura piramidale per elaborazione d'immagini, nato dalla collaborazione tra numerose Università italiane, denominato PAPIA.

Numerosi interventi, nel corso della conferenza hanno toccato gli aspetti software legati all'utilizzo degli elaboratori paralleli; uno di questi, nell'esposizione del Prof. Boari del'Università di Bologna, ha fornito una panoramica completa sui paradigmi di programmazione parallela, secondo la classica suddivisione dei linguaggi di programmazione in imperativi, funzionali logici e ad oggetti.

Per quanto riguarda i linguaggi imperativi paralleli si riscontra la tendenza, manifestata da linguaggi come Occam, CSP, Ada ed ECSP, all'utilizzo di strutture di comunicazione sincrona basate su canali tra processi, in alternativa ad una gestione a memoria condivisa; tale tendenza si lega in massimo parte al lavoro teorico di Hoare sulla comunicazione dei processi sequenziali .

I linguaggi di programmazione funzionale, logica ed object oriented, paradigmi emergenti nell'ingegneria del software mostrano interessanti caratteristiche quando se ne ipotizzil'utilizzo su piattaforme di calcolo parallele. Il formalismo funzionale, caratterizato da una robusta fondazione matematica e da costrutti esmplici e potenti, é dotato di un altissimo grado di parallelismo implicito quando non siano permessi side effect (linguaggi funzionali puri). Appartengono a questa classe linguaggi come Lisp, FP e SASL.

Analogamente al paradigma funzionale, i linguaggi logici ben si prestano ad una implementazione su macchine parallele. Facendo riferimento al linguggio Prolog ed alle versioni Parallel Prolog e Parlog, esistono due modalità di esecuzione parallela: nel parallelismo OR vengono valutate in parallelo più clausole Prolog alternative mentre nel parallelismo AND sono dimostrati concorrentemente i sottogoal di una stessa clausola. In entrambi i casi viene sfruttato il parallelismo implicito nel formalismo, senza l'esigenza di fornire costrutti aggiuntivi, come avviene per i linguaggi imperativi.

Un alto grado di parallelismo implicito é associato anche allo stile di programmazione a oggetti. Si distinguono, in questo caso, linguaggi ad oggetti passivi e ad oggetti attivi, nei quali ad ogni oggetto è associato un processo concorrente. Alla prima classe appartengono molti dei linguaggi ad oggetti utilizzati su macchine sequenziali, come SmallTalk ed i Flavor di Lips. I linguggi ad oggetti attivi (ConcurrentSmallTalk, PO, Orient84) si presentano ad una forma di parallelizzazione del tutto naturale, nella quale ogni oggetto diviene una unità esecutiva indipendente associata ad un Processing Element del sistema.

L'esigenza di poter disporre di strumenti e linguaggi di alto e altissimo livello per la programmazione parallela è stata più volte ribadita nel corso del convegno. Un accento particolare é stato posto sul tema degli strumenti per l'esplicitazione automatica del parallelismo intrinseco in un programma sequenziale, sebbene i tentativi compiuti in passato in quest'area (compilatori parallelizzanti) non abbiamo fornito risultati esalanti, soprattutto nel caso del parallelismo di grana medio-grossa, nel quale é assolutamente necessaria una conoscenza strutturale dell'algoritmo per ottenere uno shema di parallelizzazione efficace

Uno spazio di riguardo è stato riservato alle tematiche connesse con le reti neuronali e alle macchine denominate neurocomputer. Questo paradigma di calcolo, come descritto dal Prof. Mauri nell'intervento sulle attività italiane nel settore e dal Prof. Treleaven nella relazione dal titolo "Neural Computing", conosce in tempi recenti un rinnovato interesse, dovuto in buona parte ai progressi tecnologici che permettono di realizzare supporti hardware in grado di simulare modelli neuronali con tempi di risposta plausibili. Si tratta di modelli particolarmente dispendiosi in termini di risorse computazionali che sono caratterizzati da un elevato parallelismo implicito; naturale dunque associare ai neurocomputer, macchine per la simulazione di reti neuronali, un'architettura parallela.

Tra le proposte accademiche ed industriali in questo settore sono stati citati MARK III, il primo neurocomputer commerciale, associato a un host della famiglia VAX, la scheda ANZA Plus per Personal Computer, NNETS, neurocomputer sviluppato dalla NASA sulla base di quaranta transputer INPMOS.

Gli interrogativi nel settore del calcolo neuronale sono ancora numerosi, primo tra tutti quello che riguarda la possibilità di costruire applicazioni di reale complessità, superando la dimensione di *toy problem* che ha caratterizzato gli esempi applicativi fino ad oggi presenti in lettura. Questioni di notevole interesse teorico sono, dice il Prof. Mauri, la validità, in relazione ai nuovi paradigmi neuronali, della tesi di Church sull'universalità della macchina di Turing e la possibilità da parte di una rete neuronale di riprodurre meccanismi simbolici.

Parte del convegno è stata dedicata alla presentazione di lavori origianli sviluppati su elaboratori paralleli. La presentazione di un'applicazione di simulazione fluidodinamica, implementata su una macchina per automi cellulari, di applicazioni di riconoscimento di immagini, di fisica e chimica computazionale e di progettazione strutturale hanno occupato parte della seconda giornata di lavoro, fornendo un quadro efficace delle possibilità e delle problematiche di realizzazione che si legano alla produzione di software su piattaforme di calcolo parallele.

In conclusione, il convegno di Milano ha permesso di definire lo stato dell'arte e le tendenze di sviluppo nel settore, mettendo in particolare evidenza la necessità di operare un salto di qualità nella produzione di strumenti software adeguati all'utilizzo di risorse di calcolo ad architettura parallela ed il crescente interesse per il paradigma di computazione neuronale. Confortante, in quest'ottica, é il fatto che le metodologie di sviluppo del software ed i paradigmi computazionali che vanno recentemente affermandosi nella cultura informatica siano dotati di un grado di parallelismo implicito in generale superiore agli approcci più tradizionali, permettendo alle macchine parallele di proporsi non solo come risposta alle esigenze di maggiore potenza di calcolo, ma anche come piattaforme ideali per la realizzazione di nuove soluzioni computazionali.

EDITO DA CALEIDOGRAF S.N.C.
VIA L. DA VINCI N.4/6 TEL. 039 - 587541
22058 OSNAGO (CO)

REALIZZATO DALLA COMPUTER AREA S.R.L. SEZ. D.T.P.
VIA A. VOLTA, 27 20058 VILLASANTA (MI)
TEL. 039 - 306081